KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Laimonas Mikelionis

Laiko sąnaudų, skaičiuojamų su programavimo aplinkos įskiepiais, ryšių su programų inžinerijos metrikomis tyrimas

Magistro projektas

Vadovas
Doc. dr. Eimutis Karčiauskas

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

Laiko sąnaudų, skaičiuojamų su programavimo aplinkos įskiepiais, ryšių su programų inžinerijos metrikomis tyrimas

Magistro projektas
Programų sistemų inžinerija (kodas 621E16001)

Vadovas
Doc. dr. Eimutis Karčiauskas

Recenzentas
Dr. Jaroslav Karpovič

Projektą atliko
Laimonas Mikelionis
IFM-4/2 gr. studentas

KAUNAS, 2016

KAUNO TECHNOLOGIJOS UNIVERSITETAS

Informatikos fakultetas

Laimonas Mikelionis

Programų sistemų inžinerija, 621E16001 (ISCED 51252)

„Laiko sąnaudų, skaičiuojamų su programavimo aplinkos įskiepiais, ryšių su programų inžinerijos metrikomis tyrimas“

AKADEMINIO SĄŽININGUMO DEKLARACIJA

2016 m. gegužės 24 d.
Kaunas

Patvirtinu, kad mano, Laimono Mikelionio, baigiamasis projektas tema „*Laiko sąnaudų, skaičiuojamų su programavimo aplinkos įskiepiais, ryšių su programų inžinerijos metrikomis tyrimas*“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

Aš suprantu, kad išaiškėjus nesąžiningumo faktui, man bus taikomos nuobaudos, remiantis Kauno technologijos universitete galiojančia tvarka.

Laimonas Mikelionis

Turinys

Paveikslų sąrašas

Mikelionis, Laimonas. *Laiko sąnaudų, skaičiuojamų su programavimo aplinkos įskiepiais, ryšių su programų inžinerijos metrikomis tyrimas*. Magistro baigiamasis projektas, vadovas doc. dr. Eimutis Karčiauskas; Kauno technologijos universitetas, Informatikos fakultetas.

Mokslo kryptis ir sritis: Programų sistemų inžinerija

Reikšminiai žodžiai: *programavimo laikas, programų sistemų inžinerija, programos gyvavimo ciklas, programų sistemų inžinerijos kodo metrikos*.

Kaunas, 2016.

SANTRAUKA

Vis toliau plečiantis ir tobulėjant informacinių technologijų sričiai susiduriama su projekto valdymo ir laiko planavimo problemomis. Dabartinės naudojamos sistemos turi daug spragų, laiko skaičiavimo atžvilgiu, o programinės įrangos užsakovai, nori žinoti, kiek laiko buvo kurtas jų produktas. Taip pat programinės įrangos užsakovai vis dažniau pateikia programinės įrangos kodo metrikas kaip vieną iš funkcinių reikalavimų specifikacijų.

Norint išspręsti šią problemą ir išpildyti užsakovo pageidavimus reikalinga sistema, kuri sugeba sekti laiką, sugaištą rašant programinį kodą, ir gebanti paskaičiuoti sukurto kodo metrikas. Naudojant tokią sistemą būtų sekamas programavimo laikas, skaičiuojamos programinio kodo metrikos ir gerinama programinės įrangos kokybė tiek iš funkcinės, tiek iš kokybės perspektyvos.

Šio darbo metu yra kuriama programinė įranga, kaip programavimo aplinkos papildinys. Ši programinė įranga geba skaičiuoti realų ir teorinį programavimo laiką bei programinio kodo metrikas. Šiame dokumente pateikiama magistrinio darbo analizė, aptariamos realizavimui reikalingos technologijos, priimtini sprendimai. Dokumente pateikta ir analizė jau egzistuojančių, panašaus pobūdžio sistemų. Dokumento projektinėje dalyje pateikiama programinės įrangos projektavimo ir realizavimo seka. Taip pat, darbo metu sukauptų duomenų analizė bei koreliacijos, tarp programavimo laiko ir kodo metrikų, rezultatai.

Mikelionis, Laimonas. *Research of Time Costs, Acquired With a Programming Environment Plugins, and Another Source Code Metrics*. Master‘s degree, supervisor assoc. prof. Eimutis Karčiauskas. The Faculty of Informatics, Kaunas University of Technology.

Research area and field: Software engineering

Key words: *code metrics, programing time, software engineering, software development life cycle*

Kaunas, 2016.

SUMMARY

While the scope of Information Technologies further grows and improves, the problems of project management and time development often occurs. Nowadays systems used for time development have a variety of gaps, but the customers of software want to know how much time is being spent on their specific product. Moreover, software metrics is one of the functional requirements specifications required from software customers.

Seeking to solve such problems and satisfy customer’s needs, the system which could count the time spent on writing program's code and able to count metrics of code needs to be created. Usage of this type of system would allow to follow time development, counting program code metrics and improving the quality of software, taking into account from functional to quality perspective.

During the time of this work, the software as the plugin for integrated development environment is being created. This software is able to count real and theoretical development time, and metrics of software code. This document analyzes the research topic, provides the analysis of technologies that are necessary for realization and gives the idea of made decisions. This document also provides the analysis of existing systems with similar formats. The design part of the paper, provides sequences of realization and software management. Moreover, the results accumulated during the master has been provided with analysis of correlations between development time and software metrics.

1. Įžanga

Darbo objektas – programinės inžinerijos kodo metrikų analizė ir jų ryšys su laiku, sugaištu rašant programinį kodą.

Darbo tikslas – ištirti programinės kodo metrikas, atlikti jų panaudojimo analizę. Išmatuoti realaus programavimo laiko sąnaudas ir gauto laiko ryšį su programinės įrangos metrikomis.

Tyrimo uždaviniai:

- Išanalizuoti programų inžinerijoje taikomų kodo metrikų panaudojamumą.
- Sukurti programinę įrangą, skaičiuojančią programavimo laiką ir kodo metrikas.
- Panaudojus sukurtą programinę įrangą atlikti eksperimentinius skaičiavimus.
- Atlikti sukauptų duomenų analizę.
- Pagal sukauptus duomenis atlikti analizę ir išsiaiškinti programavimo laiko ir kodo metrikų koreliaciją.

Tyrimo metodika – pirmiausia reikia suprasti ir išanalizuoti programinės įrangos metrikų panaudojamumą ir naudą. Panaudojus sukurtą programinę įranga išanalizuoti sąveiką tarp laiko sąnaudų kuriant programinę įrangą ir programinės įrangos kodo metrikų.

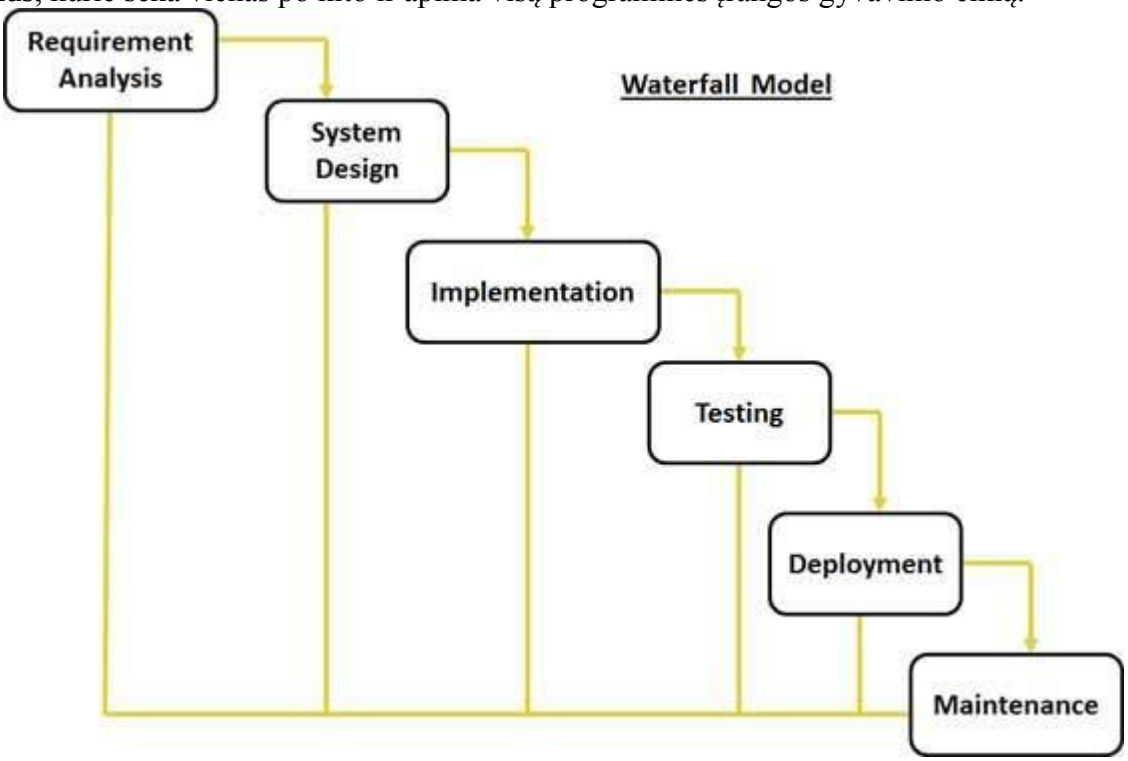
Darbo struktūra – darbą sudaro kelios pagrindinės dalys. Pirmiausia programinės įrangos kodo metrikų analizė: skaičiavimo metodika, panaudojamumas, reikšmė. Kita dalis apžvelgia sukurtą programinę įrangą, kuri skirta laiko ir kodo metrikų skaičiavimui bei atvaizdavimui. Tyrimų ir rezultatų dalis aprašo ir grafiškai pateikia sąveiką tarp programinio kodo metrikų ir laiko, sugaišto gauti tas metrikas iš programinio kodo (rašymo metu).

2. Programų sistemos inžinerijos tikslai ir metodai

Programų sistemos inžinerija – tai sritis, kuri apima visą programinės įrangos gyvavimo ciklą: programinės įrangos projektavimą, programavimą, testavimą bei palaikymą. Programinės įrangos gyvavimo ciklą sudaro šie komponentai.

- Komunikacijos
- Reikalavimų sudarymo
- Plano sudarymo
- Sistemos analizės
- Programinės įrangos projektavimo
- Programavimo
- Testavimo
- Integracijos
- Realizavijos
- Palaikymo
- Programinės įrangos panaikinimo („mirties“) [1]

Šie komponentai vaizduojami kaip dažnai naudojamas programinės įrangos kūrimo proceso valdymo *krioklio* modelis. Šis modelis yra pirmasis proceso modelis. Jis parodo programinės įrangos kūrimo žingsnius, kurie seka vienas po kito ir apima visą programinės įrangos gyvavimo ciklą.



1 pav. Krioklio modelis [2]

Yra ir daug kitų proceso valdymo modelių, tokių kaip dabar dažnai naudojamas AGILE[1], tačiau apskritai programų sistemų inžinerija apima šiuos modelius ir programinės įrangos gyvavimo ciklą.

Šio darbo metu labiausiai kreipiamas dėmesys į realizacijos dalį. Realizacijos metu gautas sistemos funkcionalumas realizuojamas programinio kodu laikantis sistemos reikalavimų specifikacijos. Programavimas užima labai didelę dalį programinės įrangos gyvavimo cikle. Užsakovas nori žinoti apie programavimo trukmę, o tai labai sunku nuspėti neturint konkrečių žinių apie projektą ir nežinant savo galimybių programavimo atžvilgiu. Šiame darbe tiriamas laiko

[1] Modelis pateiktas prieduose

ir kodo metrikų santykis gali patikslinti projekto pabaigos datą. Realizavimo metu labai svarbūs pasirinkimai tiek programavimo kalbos, tiek programavimo aplinkos.

2.1.Programinės įrangos kokybės įvertinimas

Programinės įrangos kokybė – vienas svarbiausių programinės įrangos išorinių kriterijų. Tai programinės įrangos kokybės įvertinimas, kuris neturi tikslo apibrėžimo, kokius programinės įrangos kodo ar funkcionalumo parametrus turi atitikti programinė įranga. Kokybė gali būti suprantama kaip tinkamumo naudojimui laipsnis arba kaip reikalavimų atitikimas ir defektų nebuvimas[3].

Programinės įrangos kūrimo kokybę galima suskirstyti į tris pagrindines grupes:

- Funkcinę kokybę
- Struktūrinę kokybę
- Proceso kokybę

Iš visų šių grupių labiau panagrinėkime struktūrinę kokybę. Ši programinės įrangos kokybės rūšis labiausiai tinkama nuspręsti paties programinio kodo kokybę – testuojamumą, palaikomumą, saugumą ir kt[4]. Užtikrinti programinės kokybę yra visos programinės įrangos kūrėjų darbas, labai svarbus atsižvelgiant į kliento poreikius. Automatiniam programinio kodo kokybės užtikrinimui gali padėti tokię integruoti įrankiai kaip statinė kodo analizė. Taip pat programinio kodo metrikų skaičiavimas integruotas ir į tokią programavimo aplinką kaip „Microsoft Visual Studio“. Papildomi įrankiai, tokie kaip „ReSharper“ gali padėti surasti dar daugiau klaidų programos kode jo nekompiliuodamas, tačiau šis įrankis yra mokamas.

3. Literatūros ir metodologijos analizė

3.1.Programavimo aplinkos – IDE (angl. Integrated development tools)

Programavimo aplinka – tai specializuota programa, skirta kurti programinę įrangą tam tikromis programavimo kalbomis. Tai dažniausiai būna programa, kurioje yra kodo redagavimo priemonės, derinimo ir kompiliatoriaus įrankiai, taip pat sistemos instaliavimo ir paleidimo priemonės bei įrankiai.

Kuriant paprastesnes sistemas, tokias kaip internetiniai puslapiai, galima naudoti ir kitokias programas, tokias kaip Notepad++. Ši programa yra lengvai įdiegiama, tačiau yra nepatogi dirbant su HTML, PHP, CSS, Javascript kalbomis. Specializuotos programos turi draugišką vartotojui sąsają, integruotus karkasus, kurie leidžia lengviau rašyti kodą, jį derinti, testuoti ir publikuoti.

Programavimo aplinkos seniau labai skyrėsi nuo naudojamų dabar. Pradinės programavimo aplinkos kaip *Emacs* buvo paprasti teksto redagavimo įrankiai, tačiau buvo plačiai naudojami dėl savo praplėčiamumo, realaus laiko atvaizdavimo ir geros dokumentacijos. Keičiantis laikams atsirado labiau specifikuotos programavimo aplinkos, tokios kaip Eclipe ar NetBeans IDE. Šios programavimo aplinkos turi daugelio kalbų palaikymą, teksto paryškinimo funkciją, kuri priklauso nuo programavimo kalbos ir kitus įrankius padedančius rašyti programinį kodą. Vėliau buvo sukurta programavimo aplinka Intelij Idea, kuri apjungė Eclipse ir NetBeans IDE programavimo aplinkų funkcionalumą ir jį stipriai patobulino.

Programavimo aplinką sudaro keletas skirtingų komponentų. Pagrindiniai iš jų yra: teksto redagavimo priemonė (angl. code editor), vartotojo sąsaja (angl. GUI – Graphical User Interface), kompiliatorius (angl. compiler), programos derinimo priemonės (angl. debugger), testavimo priemonės.

Kodo editorius skirtas surasti sintaksės, logikos klaidas, padėti suprasti kiekvienos programavimo kalbos specifinius žodžius, simbolius ir kt. Kiekvienas kodo interpretatorius turi bazines funkcijas, o viskas kita jau yra specializuota programavimo kalbai, kuriai jis skirtas.

Vartotojo sąsaja – tai naudojamos programinės įrangos langas. Nuo naudojamos sistemos priklauso ir vartotojo sąsaja. Kiekviena programa turi specifinį išdėstymą, tačiau dauguma programų, skirtų programavimui, turi tas pačias bazines funkcijas. Svarbiausi vartotojo sąsajos

elementai yra kodo redagavimas, derinimas, kompiliavimas. Kitos funkcijos priklauso nuo programavimo kalbos.

Kompiliatorius – vienas svarbiausių programavimo aplinkos komponentų. Jis skirtas paversti kodą, parašytą aukšto lygio programavimo kalba, į kompiuteriui suprantamą kalbą, dažniausiai dvejetainį kodą, kurį gali apdoroti procesorius. Pats kodo vertimas į dvejetainį vadinamas kompiliavimu.

Kodo derinimo priemonė – tai komponentas, leidžiantis vykdymo metu stabdyti kodą norimoje vietoje, taip pat matyti tam tikrus parametrus realiu laiku bei peržiūrėti klaidas. Kiekvienos programinės aplinkos, skirtos programavimui, kodo derinimo priemonė yra skirtinga, turinti savą funkcionalumą.

Statinė kodo analizė – funkciją realiu laiku sekanti rašomą kodą ir gebanti sekti galimas klaidas programos kode. Šio komponento tikslas – nekompiliuojant kodo rasti kiek įmanoma daugiau klaidų programos kode.

3.2.NetBeans IDE

NetBeans IDE yra atviro kodo programavimo aplinka, skirta JAVA, PHP, C++,HTML ir kitoms programavimo kalboms. Pati NetBeans programinė įranga yra realizuota naudojant JAVA programavimo kalbą ir veikia dažniausiai naudojamose operacinėse sistemose, nes naudoja JVM (angl. Java virtual machine) technologiją, kuri sukurtą programinę įrangą paleidžia ne pačioje operacinėje sistemoje, o virtualioje mašinoje. Taip atsiribojama nuo operacinės sistemos apribojimų. Kadangi nėra priklausomybės nuo operacinės sistemos, sukurta programinė įranga gali būti vykdoma ir kuriama tiek Solaris, Mac OS, Windows arba Linux operacinių sistemų aplinkoje.

Taip pat NetBeans IDE leidžia naudoti papildomus įskiepius, papildinius, kurie praplečia programinės įrangos galimybes bei palengvina programavimo darbus. Didžioji dalis papildinių yra sukelti į NetBeans interneto puslapį, kur savo papildinius (įskiepius) gali įkelti bet kas. Negalima aklai pasitikėti ten esančiomis programomis, nes tai gali pakenkti programavimo aplinkai.

3.3.Eclipse IDE

Kita plačiai naudojama programavimo aplinka – Eclipse IDE. Eclipse – tai atviro kodo integruota kūrimo aplinka. Pagal dizainą Eclipse nėra išbaigta. Ją sudaro įskiepiai, kurių kiekvienas gali turėti savus langus ar dalį programos lange. Visa Eclipse programinė įranga paremta įskiepių modeliu. Kiekvienas iš įskiepių skirtas skirtingam veiksmui atlikti. Vieni gali kompiliuoti kodą, kiti derinti, testuoti ir t.t. Pasak kūrėjų, programinės įrangos funkcionalumą riboja tik vaizduotė, nes panaudojus tinkamus įskiepius su Eclipse programavimo aplinką, kuriamai programinei įrangai, apribojimai smarkiai sumažėja. Net patys įskiepiai Eclipse programai gali būti kuriami naudojant Eclipse IDE. Nors pati Eclipse palaiko daug programavimo kalbų, bet dažniausiai ji siejama su JAVA programavimo kalba kaip viena geriausių programavimo aplinkų šiai kalbai. Taip pat labai daug šių dienų programuotojų ją naudoja mobiliųjų aplikacijų kūrimui Android aplinkoje, kuri taip pat paremta JAVA programavimo kalba ir yra populiariausia išmaniųjų telefonų operacinė sistema. Tai labai lanksti ir patogi naudoti integruota kūrimo aplinka.

3.4.Įskiepiai

Kiekvienai programinei įrangai retkarčiais reikia tam tikrų atnaujinimų, kurie praplečia programinės įrangos galimybes. Įskiepis leidžia programinei įrangai, naršyklei ar kt. padaryti tai, ko sistema negali padaryti pati. Yra įskiepių, kurie, pavyzdžiui, leidžia atvaizduoti tam tikrus dokumentus naršyklėje arba praplečia esamų dokumentų ar failų palaikomumą. Nepaisant to, ne kiekvienas įskiepis yra naudingas sistemai. Yra įskiepių, kurie gali būti skirti duomenų vogimui ar kitiems neteisėtiems veiksmams. Kiekvieną įskiepį ar papildinį galima įsidiegti tik įsitikinus, jog jis nekels problemų sistemoje ir neatlikinės neteisėtų veiksmų. Įskiepiai dažniausiai turi skaitmeninį parašą, kuris įrodo, jog jie pavojaus nekelia.

Įskiepiai gali būti skirstomi į kelias grupes. Vieni skirti programinei įrangai, esančiai kompiuteriuose, kiti – naršyklėms. Naršyklėms skirti įskiepiai vadinami „add-on“. Keletas tokių įskiepių pavyzdžių būtų Macromedia Flash Player arba Apple QuickTime player, skirti multimedijos failų peržiūrai naršyklėse. Stacionariuose kompiuteriuose esančiai programinei įrangai skirti įskiepiai, tokie kaip Eye Candy. Anksčiau minėtas įskiepis praplečia Adobe Photoshop programinės įrangos naudojamų filtrų galimybes. Įskiepiai labai palengvina programavimo, darbo su programa ar naršymo internete procesą. Tai labai paprastos ir patogios naudoti programėlės.

Dažniausiai įskiepiai yra kuriami internetiniams puslapiams, paremtiems turinio valdymo sistema. Turinio valdymo sistemos pavyzdžiai yra WordPress, Joomla, Drupal ir kt. Įskiepio apibūdinimas taip pat labai priklauso ir nuo turinio valdymo sistemos. WordPress pagrindu paremti internetiniai puslapiai naudoja įskiepius. Drupal sistemų kūrėjai nenaudoja žodžio „įskiepis“. Jie labiau linkę papildinius vadinti moduliais. Nepaisant to, tai reiškia tuos pačius įskiepius kaip ir WordPress sistemos. Nepaisant didelės gausos įskiepių, esančių internete ir skirtų skirtingoms turinio valdymo sistemoms, įskiepius reikia rinktis atsargiai, nes kitaip jie gali tik pakenkti jūsų internetiniam tinklapiui.

Kodėl verta kurti įskiepius turinio valdymo sistemoms? Turinio valdymo sistemos labai pakeitė internatinių puslapių kontingentą, jų pagalba yra daug lengviau ir paprasčiau kurti internetinius puslapius, taip leidžiant eiliniams vartotojams pateikti informaciją internete. Turinio valdymo sistemos (angl. CMS), tokios kaip WordPress bei Drupal, leidžia lengvai kurti blog‘us bei internetines parduotuves. Visa tai įskiepių dėka. Įskiepiai leidžia kurti papildomus filtrus, metodus, galinčius apdoroti veiksmus, kurių pagal nutylėjimą pati turinio valdymo sistema apdoroti negali. Dažniausiai turinio valdymo sistemos, paremtos PHP programavimo kalba, kuri yra labai dinamiška ir leidžia kurti puslapius, turinčius draugišką vartotojo sąsaja. Vartotojo sukurto įskiepio įdiegimas į sistemą taip pat labai paprastas, nes visos turinio valdymo sistemos turi labai paprastą ir patogią naujų įskiepių įdiegimo bei valdymo sąsają.

3.5.Mongo DB duomenų bazė

Mongo DB duomenų bazė – nereliacinė duomenų bazė. Labai plačiai plintanti, nes yra greita, nėra griežtų apribojimų schemai ir yra lengvai praplečiama. Duomenų bazė duomenis saugo JSON (JavaScript object notation) formatu. Tai labai patogu, jei sistemos duomenys yra siunčiami tinkle, nes JSON formatas yra tiesiog tam tikro formato simbolių eilutė, kurioje nereikia specializuoti specialių formatų ir tikrinti, ar formatai, ženklai yra leidžiami. Visi duomenys saugomi kaip simbolių eilutės ir nereikia rūpintis formatais.

Duomenų bazė yra lengvai keičiama ir praplečiama, kadangi nėra griežtos schemos, todėl vienas įrašas duomenų bazėje gali saugoti 10 atributų, kitas - 5, kitas gali 25. Skirtingai nuo SQL duomenų bazės, kurioje turi būti saugoma visa eilutė, net jei kai kurie atributai nėra nustatyti, bet jie vis tiek turi būti saugomi vien tam, kad būtų taisyklingai išsaugomi duomenys. Taip pat SQL duomenų bazėje turi būti nustatyti duomenų formatai ir jie negali būti keičiami. Mongo DB duomenų bazės populiarėja dėl dinamiškumo, greitaveikos ir paprasto naudojimo. Daug didelių projektų dažnai naudoja Mongo DB duomenų bazę arba kitą nereliacinę duomenų bazę.

3.6.Projekto valdymo sistema

Kuriant sistemą, yra būtina/siūloma naudoti projekto valdymo sistemą, kad būtų lengviau kurti projektą, sekti darbus, struktūrizuoti projekto kūrimo procesą. Tam panaudota internetinė sistema Pivotal Tracker. Ji pasirinkta dėl lengvo, paprasto naudojimo, mažiems projektams yra nemokama, lengva pridėti naujus darbus, lengva kitiems vartotojams sekti projekto eigą. Jei kuriant projektą yra naudojama versijų valdymo sistema, Pivotal Tracker gali būti prijungiama prie šios sistemos ir visos kodo versijos gali būti matomos Pivotal Tracker sistemoje.

Sistema lengva ir paprasta naudotis. Lengva sekti projekto eigą, darbus ir kt.

3.7.Versijų kontrolė

Kuriant programinę įrangą būtina naudoti versijų valdymo sistemą, kuri leistų sekti kodo pakeitimus, naujas versijas. Taip pat leistų lengvai pabandyti pridėti naujus elementus ir naują funkcionalumą, o jiems nepasiteisinus, galima būtų viską grąžinti atgal ir bandyti vėl. Šiam tikslui galima naudoti tiek Git sistemą, tiek SVN. Patogiau naudoti Tortoise SVN programinę įrangą, nes ji integruojama į Windows OS aplinką. Kad būtų galima naudoti šią sistemą, reikia turėti serverį, kuriame kodas būtų talpinamas arba panaudoti tai, ką siūlo kitos sistemos. Ir taip pat reikia turėti Tortoise SVN klientą, kuris būtų jau naudojamas kaip klientas operacinėje sistemoje. Sistema padeda kurti programinę įrangą, sekti naujausius realizuotus funkcionalumus ir kodo pakeitimus.

3.8.Netbeans IDE ir Eclipse IDE palyginimas

Eclipse IDE skirtumas nuo Netbeans IDE didelis. Eclipse turi nedidelį savo funkcionalumą, o visas kitas funkcijas leidžia įskiepiai. Tie įskiepiai praplečia Eclipse IDE funkcionalumą, priklausomai nuo įskiepių ir platformos, su kuria norima dirbti. Jei Eclipse IDE yra konfigūruojama taip, kad dirbtų su Android operacine sistema, tai reikia prijungti Android emuliatorius, taip pat Android SDK ir kt. Tai reikia tam, kad sukonfigūruotų ir paruoštų darbui aplinką. Taip pat Eclipse IDE naudoja daugiau resursų, bet Netbeans IDE skurdi savo funkcionalumu.

Eclipse IDE ypač populiari taip pažengusių programuotojų ir labiau paplitusi tarp Android programuotojų. Tačiau pradedantiesiems programuotojams labiau rekomenduojama naudoti Netbeans IDE dėl paprastos vartotojo sąsajos ir greito veikimo.

3.9.Java ir Mongo DB

Mongo DB dažniausiai naudojama dirbant su .NET, Node.js, Angular.js platformomis. Retai kada naudojama dirbant su Java aplikacijomis, tačiau tai labai patogu naudoti Java aplikacijose. Mongo DB greičiau veikia nei SQL, todėl galima pagerinti greitaveiką ir taip lėtoje Java aplinkoje. Java ir Mongo Db duomenų bazė bendrauja naudojant Mongo Db klientą, ir visa informacija keliauja BSON formato duomenimis. Tai reiškia, kad dažniausiai naudojamos simbolių eilutės ir tai lengva naudoti kitose kalbose. Java yra labai struktūrizuota programavimo kalba ir jai reikia tiksliai žinoti objekto tipą, jei norima prieiti prie duomenų. .NET aplinkoje objektai gali būti aprašomi bendriniais vardais naudojant „var“ žodį ir kintamojo tipas bus nustatytas tik priskyrus tam tikrą reikšmę. Java kalboje duomenų tipas turi būti žinomas, kitaip bus daug klaidų ir „nulaužimų“.

4. Alternatyvų analizė

Šiame skyriuje apžvelgiama panašių projektų funkcionalumas bei panaudojamumas. Atrinkta keletas labiausiai funkcionalumą atitinkančių programų ir jos aptariamos toliau.

4.1.WakaTime

WakaTime – programinės įrangos platforma, leidžianti skaičiuoti programinės įrangos darbo laiką ir kai kurias kitas programinio kodo savybes. Sistema yra realizuota kaip internetinė platforma konkrečiam vartotojui. Ši sistema turi daug sukurtų įskiepių skirtingoms platformoms bei programavimo aplinkoms. Visa informacija naudojant internetinių paslaugų platformą (angl. web service) yra siunčiama į WakaTime tinklapio duomenų bazę. Visą informaciją galima gauti per WakaTime internetinį puslapį: https://wakatime.com/. Informaciją pavaizduota grafikais ir lentelėmis. Kad būtų galima skaičiuoti duomenis, reikia atlikti tam tikrus žingsnius:

- Prisiregistruoti prie tinklapio,
- Gauti API raktą,
- Gautą raktą įvesti į pasirinktos platformos įskiepio prisijungimus,
- Galima pradėti darbą, nes sistema veikia automatiškai.

WakaTime veikimui reikalinga Python programinio kodo bazė, nes WakaTime sistema paremta Python programavimo kalba. Sistema sugeba saugoti failo kelią sistemoje, laiką dirbtą su sistema.

Žino, kokia programavimo kalba programuota, seka visus pakitimus laiko skaičiavime ir kt. Duomenis galima eksportuoti arba importuoti.

Sukurta WakaTime programinė įranga yra mokama, todėl naudojimasis ja kainuoja. Kainos svyruoja priklausomai nuo programinės įrangos norimo funkcionalumo ir naudojimosi laiko. Sistema sugeba sekti gaunamus duomenis ir priklausomai nuo turimos licencijos tipo, atvaizduoja tam tikro laikotarpio istorija. Taip pat sistemą turi visa dokumentaciją funkcionalumo integravimui į kitas, jau egzistuojančias sistemas

4.2.Project Time Tracker

Project Time Tracker yra ne sistema, tačiau vieno programuotojo sukurti metodai, kurie leidžia gauti laiką, dirbtą su klase. Šie programinio kodo pavyzdžius galima panaudoti kuriant sistemą laiko skaičiavimui. Kodas paremtas failo keitimo algoritmu, kuris seka visus failo pakeitimus ir priima tai kaip programavimo laiką. Visą informaciją galima rasti nurodytu adresu: https://blogs.oracle.com/geertjan/entry/project_time_tracker.

5. Programų sistemos inžinerijos metrikos

5.1. Programinės įrangos metrikos

Programinės įrangos metrikos plačiai naudojamos tiek tarp programinės įrangos kūrėjų, tiek tarp sistemos vartotojų. Šios metrikos – tai kiekybinis programinės įrangos įvertinimas, tačiau gali būti naudojamos kaip vienas iš reikalavimų programinei įrangai. Pvz., kliento reikalavimas – programinės įrangos išeities kodo palaikomumo indeksas turi būti daugiau nei 85. Tai atvejis kada programinės įrangos metrikos gali būti panaudojamos kaip reikalavimas.

Dažniausiai programinės įrangos metrikos skirstomos į 3 pagrindines grupes:

- Produkto metrikos;
- Proceso metrikos;
- Projekto metrikos;

Kiekviena grupė turi tam tikras specifines metrikas, kurios tinka konkrečiai grupei, kitos metrikos gali būti priskirtos kelioms grupėms. Šios metrikos gali būti panaudotos nustatyti programinės įrangos kokybę. Kokybės metrikos labiausiai siejamos su produkto ir proceso metrikomis. Tačiau kokybę gali labai stipriai veikti parametrai iš projekto grupės: programuotojų skaičius, įgūdžiai, programinės įrangos kūrimo tvarkaraštis, įmonės dydis, struktūra. Nepaisant to, programinės įrangos kokybė turi būti apžvelgiama iš visos programinės įrangos gyvavimo ciklo perspektyvos [5]. Taip galima susieti tiek projekto kūrimo dalį, su procesais ir galutinio produkto parametrais bei metrikomis.

Proceso metrikos

Proceso metrikos apibūdina programinės įrangos kūrimo procesų metrikas. Jos gali būti panaudotos siekiant pagerinti programinės įrangos kūrimo ir palaikymo procesus. Vienos iš tokių metrikų gali būti defektų pašalinimas programinės įrangos kūrimo metu arba programinės įrangos problemų taisymo atsako laikas. Naudojant AGILE paremtą programinės įrangos kūrimo modelį, proceso metrikas galima pritaikyti kiekvienos iteracijos metu. Jei naudojama, pvz., klaidų tankumo metrika, galima kiekvienos iteracijos metu išmatuoti klaidų tankumą ir taisyti programinę įrangą iki iteracijos pabaigos arba kitos iteracijos metu.

Projekto metrikos

Projekto metrikos skirtos apibūdinti programavimo procesą viso gyvavimo ciklo metu. Tokios metrikos kaip programuotojų skaičius, kaina, programavimo tvarkaraštis ir produktyvumas geriausiai tinka apibūdinti programuotojų komandos darbą ir projekto gyvavimo ciklą.

Produkto metrikos

Produkto metrikos skirtos apibūdinti galutinės programinės įrangos charakteristikas. Tokie dydžiai kaip programos dydis, greitaveika, sudėtingumas ir kokybės lygis priklauso produkto metrikų grupei. Tai leidžia spręsti apie galutinio produkto kiekybinį įvertį.

5.2. Programinės įrangos kodo metrikos

5.2.1. Kodo eilučių skaičius

Labai dažnai naudojama programinės įrangos metrika yra kodo eilutės (angl. Lines of Code (LOC)). Naudojant šią metriką galima įvertinti programinės įrangos dydį eilučių skaičiumi, tačiau yra labai sunku nuspręsti, kurios eilutės yra tinkamos, kurios – ne. Anksčiau, kai programavimas vykdavo asembleriu, buvo labai lengva skaičiuoti programos kodo dydį eilučių skaičiumi, nes kiekviena kodo eilutė reiškė vieną instrukciją. Atsiradus aukšto lygio programavimo kalboms, tapo labai sunku skaičiuoti programinio kodo eilutes, nes skirtingai kalbai skirtinga eilutė reiškė skirtingus loginius veiksmus programos viduje. Todėl vienas su vienu ryšys, kaip buvo assemblerio laikais, nutrūko. Norint teisingai skaičiuoti programinio kodo eilutes, reikia teisingai nuspręsti kas bus skaičiuojam, ką priskirti kaip programinio kodo eilutę, o ko ne. Tai galima padaryti keliais būdais:

- Skaičiuoti tik vykdomas eilutes;
- Skaičiuoti vykdomas eilutes ir duomenų priskyrimus;
- Skaičiuoti vykdomas eilutes, duomenų priskyrimus ir komentarus;
- Skaičiuoti eilutes įvedimo ekrane;

Skirtingi programinio kodo eilučių skaičiavimo metodai naudojami skirtingose programinėse įrangose skirtose paskaičiuoti programinės įrangos metrikas. Vienos gali įtraukti komentarus ir duomenų priskyrimo sakinius. Kitos gali juos ignoruoti. Dėl to turi būti pasirinkta programinė įranga, kurios skaičiavimo algoritmai labiausiai atitinka įmonės politiką, kaip turi būti kuriamas kodas ir kokios eilutės išeities kode yra svarbios.

Tokiose kalbose kaip BASIC, PASCAL ir C kelios loginės (ar vykdomosios) kodo eilutės gali būti aprašomos kaip viena fizinė eilutė. Taip gali būti ir kitose kalbose, tačiau dėl lengvesnio programinės įrangos koregavimo ir programavimo kultūros, programinis kodas turi būti lengvai skaitomas ir suprantamas žmonėms, kurie programinės įrangos nekūrė, o ją tik palaiko. Arčiausias santykis tarp loginių ir fizinių eilučių būtų FORTRAN programavimo kalbos. Ir santykis būtų artimas 1. Skirtumas tarp programinės įrangos fizinių ir loginių eilučių gali būti ir 5. Vadinasi, viena fizinė eilutė atlieka 5 logines operacijas. Tai labai didelis skirtumas ir negalima vienareikšmiškai skaičiuoti, kad viena fizinė eilutė yra viena loginė eilutė (instrukcija). Pavyzdžiui, turint programinį kodą:

for (i = 0; i < 100; i++) printf("hello"); / How many lines of code is this? */*

Pateiktas kodas turi:

- 1 fizinę eilutę,
- 2 logines eilutes,
- 1 komentarų eilutę.

Pagal kodo standartus bei lengviau skaitomą ir palaikomą kodą, ši eilutė gali būti perrašyta taip:

```
/* Now how many lines of code is this? */
for (i = 0; i < 100; i++)
{
   printf("hello");
}
```

Pateiktas kodas turi:

- 5 fizines eilutes,
- 2 logines eilutes,
- 1 komentarų eilutę.

Tai parodo, kaip skirtingas programinio kodo eilučių skaičiaus skaičiavimo algoritmas gali pakeisti gaunamus duomenis. Matome, kad tiek pat loginių operacijų ir komentarų galima sutalpinti į 1 arba į 5 eilutes. Tai parodo, koks netikslus yra programinio kodo eilučių skaičiavimo metodas.

5.2.2. Ciklomatinis sudėtingumas

Ciklomatinis sudėtingumas – viena labiausiai naudojamų programinės įrangos metrikų. Ją išrado Thomas J. McCabe 1976 m. Tai viena tiksliausiai paskaičiuojamų programinės įrangos metrikų ir yra plačiai naudojama programinėje įrangoje, skirtoje skaičiuoti sukurtos ar kuriamos programinės įrangos metrikas. Ciklomatinis sudėtingumas – tai programinės įrangos išeities kodo visų galimų grafo kelių skaičius. Turint programinį kodą, galima sudaryti grafą, kuris nusako visus galimus kelius sukurtai programinei įrangai. Ciklomatinis sudėtingumas turi ir savo matematinę išraišką. Ją galima apibūdinti taip:

$$M = E - N + 2P$$

M – ciklomatinis sudėtingumas,
E – briaunų skaičius grafe,
N – viršūnių skaičius grafe,
P – sujungtų komponentų skaičius.

Tai parodo, jog paskaičiuoti ciklmatinį sudėtingumą programai nėra sunku. Sunkiausia yra sudaryti grafą, kuris atvaizduotų visus galimus programinės įrangos veiksmus ir iš to būtų galima gauti viršūnių, briaunų ir sujungtų komponentų skaičių. Pavyzdžiui, toliau pateiktai programai galima paskaičiuoti ciklomatinis sudėtingumą. Matome, kad programa turi du sąlyginius sakinius bei du ciklus. Šiuo atveju galima panaudoti kitą Ciklomatinis sudėtingumo matematinę išraišką:

$$M = \#IFS + \#Loops + 1$$

#IFS – sąlyginių sakinių skaičius,
#Loops – ciklų skaičius programoje.

```
void sort( int*a, intn )
{
    int i, j, t;
      if( n < 2 ) return;
          for( i=0 ; i < n-1; i++ ) {
              for( j=i+1 ; j < n ; j++ ) {
```

```
            if( a[i] > a[j] ) {
            t= a[i];
            a[i] = a[j];
            a[j] = t;
            }
          }
      }
  }
```

Pateiktoje programoje matome, kad yra 2 sąlyginiai sakiniai bei 2 ciklai, todėl šios funkcijos ciklomatinis sudėtingumas M = 2 + 2 +1 = 5. Ciklomatinis sudėtingumas šiuo atveju yra vidutinis. McCabe rekomenduoja, kad maksimali ciklomatinio sudėtingumo reikšmė būtų 10. Jei programinės įranga sudėtingesnė, reikėtų bandyti ją skaidyti arba supaprastinti.

Funkcinio programavimo atveju ciklomatinis sudėtingumas gali būti apskaičiuojamas lengvai, kadangi kiekviena operacija seka viena po kitos. Tačiau objektinio programavimo atveju tai padaryti sunkiau, nes operacijos skaidomos funkcijomis, klasėmis ir objektais. Tokiu atveju geriausia skaičiuoti ciklomatinį sudėtingumą atskiroms funkcijoms. Pagal McCabe, ciklomatinis sudėtingumas neturi viršyti 10, todėl jei funkcija ar metodas viršija šį skaičių, programinė įranga turi būti skaidoma arba prastinama priklausomai nuo situacijos. Tai leis ne tik supaprastinti programinį kodą, bet ir pagerinti programinės įrangos palaikomumą.

5.2.3. Palaikomumo indeksas

Palaikomumo indeksas (MI) – tai išvestinė programų metrika, susidedanti iš kelių Halstead‘o metrikų ir turinti kelias skaičiavimo formules. Ši metrika leidžia apskaičiuoti ir nustatyti programinės įrangos palaikomumo indeksą, pagal kurį galima spręsti, ar laikomasi programavimo taisyklių, lygiavimo, komentarų rašymo ir kt. Palaikomumo indeksas gali būti skaičiuojamas tiek visai sistemai, tiek atskiriems moduliams. Visai sistemai turi būti imamos vidutinės atskirtų modulių reikšmės. Palaikomumo indeksas turi kelias skaičiavimo galimybes, bet visoms reikalinga eilučių kiekio metrika. Viena iš funkcijų, kaip paskaičiuoti palaikomumo indeksą, pateikta toliau:

$$MI = 171 - 3.42 \times \ln E - 0.23 \times CC - 16.2 \times \ln LOC$$

čia MI – palaikomumo indeksas (angl. Maintainability index), E – Halstead‘o pastangos, CC – ciklomatinis sudėtingumas, LOC – eilučių kiekis. Vėliau tokios metrikos, kaip palaikomumo indeksas, skaičiavimas buvo integruotas į vieną populiariausių programavimo aplinkų „Microsoft Visual Studio“.

Turint paskaičiuotą palaikomumo indeksą programos kodui, galima spręsti apie šios programos prižiūrimumą naudojantis toliau pateikta lentele.

Lentelė 1. Palaikomumo indekso vertinimo lentelė

MI reikšmė	**Programos įvertinimas**
> 85	geras prižiūrimumas
65 - 85	vidutinis prižiūrimumas
0 - 65	blogas prižiūrimumas
< 0	labai blogas programos kodas (nestruktūrizuotas, nekomentuotas)

5.3.Halstead‘o metrikos

Šiuolaikinė programinė įranga, skirta tirti programinio kodo metrikas, naudoja daug įvairių algoritmų ir įvairių modelių skaičiuoti metrikas. Tačiau didžiąją dalį šių algoritmų sudaro modeliai ir funkcijos, kurias atrado Maurice Howard Halstead 1977 m. Šių metrikų skaičiavimas paremtas keliais baziniais parametrais, kurie yra gaunami iš programinės įrangos kodo. Halstead‘o teigimu, „kompiuterinė programa yra algoritmo realizacija, algoritmą nusakant kaip operatorių ir operandų skaičiumi“[6]. Kai kurios metrikos buvo panaudotos kaip bazinis elementas kitų metrikų skaičiavimų algoritmams sukurti. Norint paskaičiuoti Halstead‘o metrikas, reikia panaudoti šiuos 4 parametrus:

- n_1 – unikalių operatorių skaičius,
- n_2 – unikalių operandų skaičius,
- N_1 – operatorių pasirodymo skaičius,
- N_2 – operandų pasirodymo skaičius.

Naudojant šiuos parametrus, galima apskaičiuoti kitas Halstead‘o metrikas. Taip pat kai kurios metrikos įeina ir į kitų metrikų skaičiavimo formules, kaip pavyzdžiui, palaikomumo indeksas.

5.3.1. Programos ilgis

Programos ilgis yra dviejų Halstead‘o bazinių elementų suma – operatorių pasirodymo skaičius ir operandų pasirodymo skaičius. Ši metrika žymima raide N.

$$N = N_1 + N_2.$$

5.3.2. Programos žodynas

Programos žodynas yra suma Halstead‘o bazinių elementų – unikalių operatorių ir operandų suma. Tai parodo, kiek yra unikalių operatorių ir operandų programos tekste. Žymima n. Apskaičiuojama pagal:

$$n = n_1 + n_2.$$

5.3.3. Programos apimtis

Halstead‘as aprašo programos apimtį kaip „tinkamą priemonę bet kokiai realizacijai bet kokiam algoritmui“ arba kaip „numerį reikalingų palyginimų sugeneruoti programai“[6]. Kitas šios metrikos apibūdinimas būtų „minimalus skaičius bitų, reikalingų dekoduoti programą“[7]. Programos ilgis, reikalingas išspręsti tą pačią problemą, priklauso nuo pasirinktos programavimo kalbos. Ši metrika yra žymima raide V. Apskaičiuojama pagal toliau pateiktą formulę, kurioje yra panaudota programos žodyno metrika:

$$V = N * \log_2 n.$$

5.3.4. Minimali (potenciali) programos apimtis

Potenciali programos apimtis parodo glaustą programos versiją, reikalingą suprogramuoti kodą, skirtą problemai išspręsti. Kadangi minimaliai problemai išspręsti reikalinga bent viena funkcija, o funkcija turi pavadinimą ir skliaustus. Tai jau yra du operatoriai (pvz., func()) [7]. Taip pat reikia pridėti ir kitus galimus operatorius ir operandus. Ši metrika žymima V^* , o funkcija skaičiavimui pateikta toliau.

$$V^* = (2 + n_2) * \log_2(2 + n_2).$$

5.3.5. Programos lygis

Programos lygio metrika parodo santykį taip minimalios programos apimties ir realios programos apimties. Tai netiesiogiai leidžia nuspręsti pasirinktos programavimo kalbos, kuri bus

naudojama išspręsti problemą, sudėtingumą. Ši metrika žymima raide L (angl. level) ir apskaičiuojama kaip santykis:

$$L = \frac{V^*}{V}$$

5.3.6. Programos sudėtingumas

Programos sudėtingumas taip pat viena iš Halstead‘o metrikų. Ši metrika yra atvirkščias dydis programos lygiui. Žymima raide D (angl. difficulty) ir apskaičiuojama pagal šią formulę:

$$D = \frac{1}{L} = \frac{n_1}{2} \times \frac{N_2}{n_2}.$$

5.3.7. Programavimo pastangos

Programavimo pastangos – labai abstrakti metrika, tačiau turinti savo skaitinę reikšmę ir skaičiavimo formulę. Norint gauti pastangas, reikalingas sukurti programinę įrangą, imamas santykis tarp programos lygio ir programos žodyno arba sudėtingumo ir žodyno sandauga. Žymima raide E (angl. effort) ir apskaičiuojama pagal šią formulę:

$$E = \frac{V}{L} = D \times V$$

5.3.8. Programavimo laikas

Programavimo laikas labai svarbus kuriant programinę įrangą, nes užsakovas/klientas dažnai nori žinoti, kiek laiko gali užtrukti programinės įrangos kūrimas ir nori žinoti programinės įrangos kūrimo pabaigos laiką (angl. deadline). Halstead‘o metrika leidžia paskaičiuoti laiką, per kurį programuotojas turėjo kūrė programinę įranga. Kadangi ši metrika, kaip ir kitos, paskaičiuojamos iš bazinių parametrų, vadinasi, laiką galima paskaičiuoti tik jau sukūrus programinę įranga. Programavimo laiko metrika yra programavimo pastangų ir Stroud‘o skaičiaus santykis. Šis skaičius tiksliausiai apibūdinamas kaip „(...) elementarių veiksmų skaičius, kurį žmogaus smegenis gali atlikti per sekundę“[6]. Programų inžinerijoje šis skaičius žymimas S ir turi konkrečią reikšmę, kuri lygi 18. Ši metrika žymima raide T (angl. time) ir apskaičiuojama pagal šią formulę:

$$T = \frac{n_1 * N_2 * N * \log_2 n}{2 * n_2 * S} = \frac{E}{18}$$

Kaip pavyzdį šiai metrikai galima pateikti programos kodo pavyzdį (C programavimo kalba):

```
main()
{
   int a, b, c, avg;
   scanf("%d %d %d", &a, &b, &c);
   avg = (a + b + c) / 3;
   printf("avg = %d", avg);
}
```

Apskaičiuojami baziniai parametrai programai. Randame unikalius operatorius ir operandus. Unikalus operatoriai: main, (), {}, int, scanf, &, =, +, /, printf. Unikalūs operandai: a, b, c, avg, "%d %d %d", 3, "avg = %d". Pagal tai n1 = 10, n2 = 7. Apskaičiuojame N1 = 16, N2 = 15. Turėdami bazinius komponentus galime apskaičiuoti reikalingą laiką programinei įrangai sukurti naudodami S (Stroud‘o numerį), lygų 18. Tada:

$$T = \frac{1{,}3557}{18} = 75.3\ s$$

Šiai programinei įrangai sukurti reikėjo 1 min ir 15.3 s. Visa tai apskaičiuojama pagal bazinius elementus naudojant Halstead‘o metodus.

6. Laiko numatymo problema

Kuriant programinę įrangą užsakovas visada reikalauja pateikti programinės įrangos kūrimo pabaigos datą, tai yra numatyti kūrimo laiką. Taip pat vieni programinės įrangos užsakovai gali pateikti ir tam tikras metrikas, kurios turi tenkinti sukurtą programinę įrangą. Jei užsakovas tobulins sistemą pats, arba bus daromi pakeitimai programos kode pateikiama metrika gali būti palaikomumo indeksas. Aukšta šios metrikos reikšmė leidžia lengvai prižiūrėti/taisyti/tobulinti programos kodą. Taip pat tam tikruose konkursuose ar užduotyse yra nurodomas konkretus kodo eilučių skaičius, kuris negali būti viršijamas. Iš to matyti, kad programinės įrangos metrikos yra plačiai naudojamos, tačiau kitaip įvardijamos.

Žiūrint iš užsakovo perspektyvos, būtina žinoti programinės įrangos pristatymo (programavimo darbų pabaigos) datą, kad būtų galima organizuoti darbą įmonėje. Žiūrint iš programuotojo perspektyvos yra labai sunku šitą numatyti, nes jei nėra duomenų, kurie statistiškai leistų nuspręsti programinės įrangos kūrimo terminus, naujos programos terminus tiksliai numatyti praktiškai neįmanoma. Patyrę programuotojai jau gali apytiksliai numatyti terminus, kada programinė įranga bus baigta, tačiau ir tai nėra tikslu. Turint statistinę informaciją, kiek laiko buvo kurtas panašus projektas, galima didinti terminų tikslumą.

Laiko numatymo (angl. estimate) problemai spręsti galima sukurti programinę įrangą, kuri skaičiuotų tikslų laiką, kiek buvo dirbta su konkrečia klase prie konkretaus projekto. Projekto pabaigoje galima pateikti klientui ataskaitą, kiek kuri klasė/modulis ir visas projektas „kainavo“ laiko, o tai yra ir pinigai (iš kliento pusės).

Taip pat turint laiką konkrečiam projektui ir to projekto metrikų reikšmes galima gauti statistiką apie programavimo „greitį“ – kodo eilučių skaičių per laiko tarpą. Šis statistinis dydis leistų nuspėti kitų projektų programavimo laiką pagal kodo eilutes, ciklomatinį sudėtingumą ir kitas programų inžinerijos metrikas.

7. Programų inžinerijos metrikų koreliacijos tyrimo programinė įranga

Norint suskaičiuoti programinės įrangos kūrimo laiko kaštus, reikia sukurti programinę įrangą, kuri be papildomo vartotojo veiksmų sektų laiką, kurį užtruko programinės įrangos kūrėjas kurdamas konkretų funkcionalumą. Tam panaudojami įskiepiai (angl. plugins), kurie leidžia programos fone atlikti veiksmus.

Šie įskiepiai kuriami NetBeans IDE programavimo aplinkai. Netbeans IDE platforma turi funkcionalumą, kuris leidžia kurti įskiepius tai pačiai Netbeans IDE platformai. Jis kuriamas kaip eilinis Java projektas, tiktai kitoks turi būti projekto tipas. Viskas prasideda nuo projekto sukūrimo, tipas pasirenkamas Netbeans module ir toliau nurodomi tam tikri duomenys, skirti būtent įskiepio kūrimui. Taip pat įskiepis gali būti ir tam tikro potipio – tai gali būti mygtukas, gali turėti ir įvedimo laukus, sąsajas. Taip pat Netbeans IDE turi labai tvarkingą, aukšto funkcionalumo lygio instaliavimo platformą, kuri leidžia ne tik sekti naujas versijas, bet ir rūpinasi, ar įskiepiai pasirašyti skaitmeniniu parašu. Tai leidžia vartotojui ramiai instaliuoti pasirinktus įskiepius ir matyti reikalingas licencijas bei papildomą programinę įrangą.

Kuriant įskiepį pirmiausia reikia atlikti techninės architektūros analizę, kad išsiaiškintume tinkamiausius komponentus bei technologijas, skirtas sukurti programinę įrangą. Pirmiausia pasirenkama programavimo aplinka. Kadangi įskiepis kuriamas NetBeans IDE programavimo aplinka, su šia programavimo aplinka kuriamas ir įskiepis. Tai leidžia lengvai testuoti, derinti bei kuri programinę įrangą (įskiepį). Pati NetBeans IDE realizuota Java programavimo kalba, todėl kuriamas įskiepis turi būti taip pat realizuotas šia kalba.

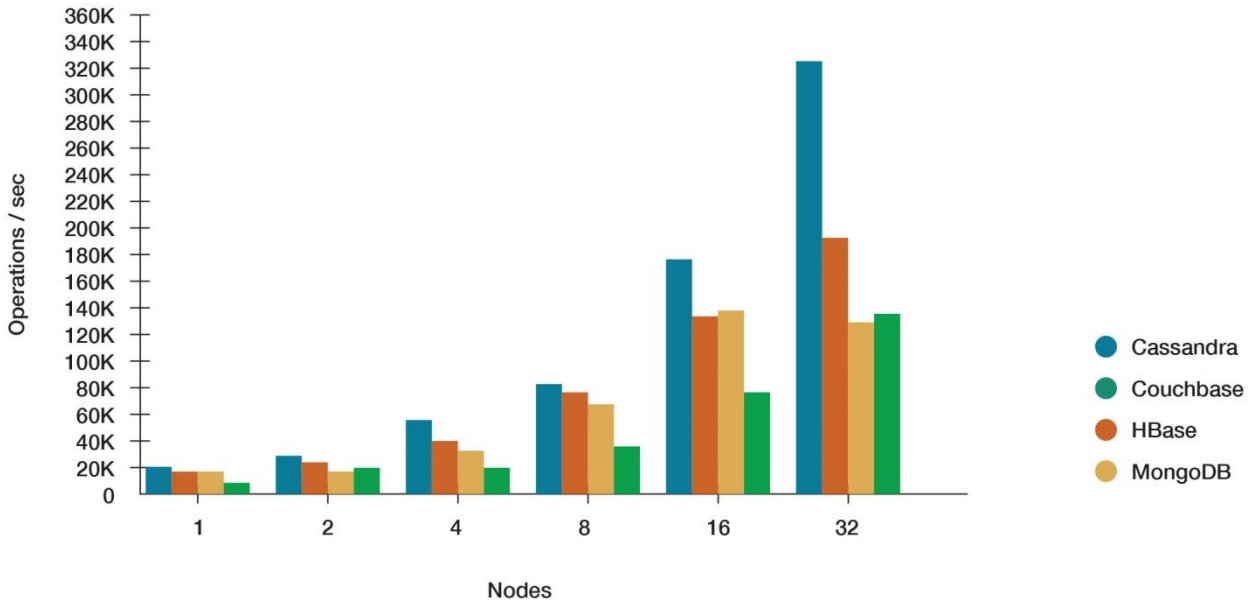
Duomenų bazės pasirinkimas yra platesnis. Pirmiausia galima rinktis tarp duomenų bazių tipų – SQL arba NoSQL. SQL duomenų bazės yra labiau paplitusios ir daugiau naudojamos visose programinėse įrangose. SQL (angl. Structured Query Language - struktūrizuota užklausų kalba) yra taip pat programavimo kalba, skirta aprašyti ir manipuliuoti duomenimis reliacinių duomenų bazių valdymo sistemose. Tai reiškia, kad duomenys yra griežtai struktūrizuoti ir yra ryšiai tarp duomenų bazių lentelių. SQL duomenų bazės turi ir didelių trūkumų. Norint praplėsti nustatytą struktūrą reikia perdaryti duomenų bazės modelį, pakeisti, papildyti, ištrinti tam tikrus laukus ar lenteles. Tai sumažina duomenų bazės praplečiamumą ir panaudojamumą.

NoSQL duomenų bazės – tai duomenų bazės be ryšių ir konkrečių struktūrų. Viena populiariausių NoSQL tipo duomenų bazių modelių – MongoDB. Mongo DB duomenų bazė – nereliacinė duomenų bazė. Labai plačiai plintanti, nes yra greita, nėra griežtų apribojimų schemai ir yra lengvai praplečiama. Duomenų bazė duomenis saugo JSON (JavaScript object notation) formatu. Tai labai patogu, jei sistemos duomenys yra siunčiami tinkle, nes JSON formatas yra tiesiog tam tikro formato simbolių eilutė, kurioje nereikia specializuoti specialių formatų ir tikrinti, ar formatai, ženklai yra leidžiami. Visi duomenys saugomi kaip simbolių eilutės ir nereikia rūpintis formatais.

Duomenų bazė yra lengvai keičiama ir praplečiama, nes nėra griežtos schemos, todėl vienas įrašas duomenų bazėje gali saugoti 10 atributų, kitas 5, kitas gali 25. Skirtingai nuo SQL duomenų bazės, kur turi būti saugoma visa eilutė, ir jei kai kurie atributai nėra nustatyti, jie vis tiek turi būti saugomi vien tam, kad būtų taisyklingai išsaugoti duomenys. Taip pat SQL duomenų bazėje turi būti nustatyti duomenų formatai ir jie negali būti keičiami. Mongo DB duomenų bazės populiarėja dėl dinamiškumo, greitaveikos ir paprasto naudojimo. Daug didelių projektų dažnai naudoja Mongo DB duomenų bazę arba kitą nereliacinę duomenų bazę.

Mongo DB dažniausiai naudojama dirbant su .NET, Node.js, Angular.js platformomis. Retai kada naudojama dirbant su Java aplikacijomis, tačiau tai MondoDB labai patogu naudoti ir Java aplikacijose. Mongo DB greičiau veikia nei SQL, todėl galima pagerinti greitaveiką ir taip lėtoje Java aplinkoje. Java ir Mongo DB duomenų bazė bendrauja naudojant Mongo DB klientą, ir visa informacija perduodama BSON (binary JSON) formato duomenimis. Tai reiškia, kad naudojamos simbolių eilutės ir tai lengva panaudoti kitose kalbose. Java yra labai struktūrizuota programavimo kalba ir jai reikia tiksliai žinoti objekto tipą, jei norima prieiti prie duomenų. .NET aplinkoje objektai gali būti aprašomi bendriniais vardais naudojant žodį „var“ ir kintamojo tipas bus nustatytas tik priskyrus tam tikrą reikšmę. Java kalboje duomenų tipas turi būti žinomas.

Toliau pateikiama tam tikrų NoSQL duomenų bazių greitaveikos tyrimas.



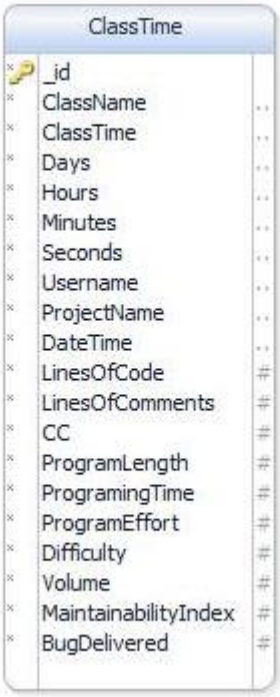
2 pav. NoSQL tipo duomenų bazių palyginimas[8]

Aukščiau pateiktoje diagramoje matomas populiarių NoSQL duomenų bazių palyginimas. Esant vienam moduliui, visų duomenų bazių operacijų laikas per sekundę yra daugmaž vienodas. Didejant modulių skaičiui išryškėja Cassandra duomenų bazės greitaveikos privalumai. Tačiau projekto įgyvendinimui pasirinkta Mongo DB duomenų bazė dėl patirties ja naudojantis ir išsamios bei detalios dokumentacijos.

7.1. Duomenų bazės modelis

Kuriamai programinei įrangai turi būti sudarytas tam tikras duomenų bazės modelis, kuris aprašo duomenų struktūrą. Kadangi naudojama NoSQL duomenų bazė, jokių ryšių tarp lentelių nėra. Tai leidžia nesirūpinti duomenų vientisumu, kas turi būti užtikrinta SQL duomenų bazėse. NoSQL duomenų bazė gali būti praplečiama tiesiog kuriant programinę įrangą. Jei vietoj 5 saugomų reikšmių bus saugomos 6, duomenų bazė automatiškai prasiplės papildomu lauku, kurio pavadinimas nurodomos programiniame kode. Taip pat NoSQL duomenų bazėje, skirtingai nuo SQL duomenų bazės, lentelės vadinamos kolekcijomis (angl. collections).

NoSQL duomenų bazė gali turėti kiek tik reikia laukų, tačiau kuriant įrašą, kuris NoSQL duomenų bazėje vadinamas dokumentu, visada bus vienas privalomas laukas, kuris yra identifikatorius konkretaus dokumento - *_id*. Šis laukas yra visada unikalus ir veikia kaip pirminis raktas (angl. primary key) SQL duomenų bazėje.

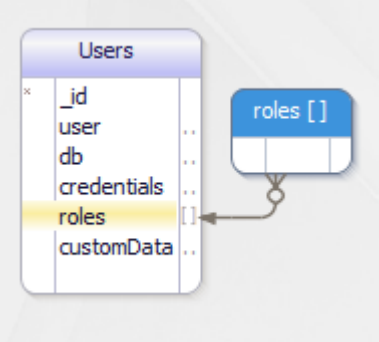


3 pav. Duomenų bazės modelis

_id	– unikalus MongoDB identifikatorius
ClassName	– klasės su kuria dirbama pavadinimas
ClassTime	– datetime tipo parametras
Days	– dienos dirbtos su šia klase
Hours	– valandos dirbtos su šia klase
Minutes	– minutės dirbtos su šia klase
Seconds	– sekundės dirbtos su šia klase
Username	– vartotojo vardas kuris dirbo su klase
ProjectName	– projekto kuriam priklauso klasė pavadinimas
DateTime	– datetime tipo parametras
LinesOfCode	– kodo eilučių klasėje skaičius (metrika)
LinesOfComments	– komentarų klasėje skaičius (metrika)
CC	– ciklomatinis sudėtingumas (metrika)
ProgramLength	– programos ilgis (metrika)
ProgramingTime	– programavimo laikas pagal Halstead‘o metrikas (metrika)
ProgramEffort	– programavimo pastangos (metrika)
Difficulty	– programos sudėtingumas (metrika)
Volume	– programos apimtis (metrika)
MaintainabilityIndex	– palaikomumo indeksas (metrika)
BugDelivered	– klaidos, kurios bus pristatytos produkte pagal programinį kodą (metrika)

Kaip parodyta aukščiau pateiktame paveiksle duomenų bazę sudaro viena lentelė. Taip yra todėl, kad nereliacinėje duomenų bazėje dokumentas gali būti saugomas dokumente. Tai reiškia, kad tai, ką reikėtų aprašyti 2-3 lentelėmis SQL tipo duomenų bazėje, galima aprašyti vienu dokumentu NoSQL duomenų bazėje.

Nors pateiktas duomenų bazės modelis yra tik viena lentelė, tačiau ji skirta saugoti visą reikalinga informaciją tiek pirmam įskiepiui, skirtam skaičiuoti programavimo laiką ir programinės įrangos kodo metrikas, tiek antram, skirtam atvaizduoti surinktą informaciją. Prisijungimas prie duomenų bazės vyksta naudojantis Mongo DB klientu. Tam, kad būtų autorizuotas vartotojas, naudojama numatytoji lentelė vartotojams saugoti. Toliau pateikiama vartotojų lentelės schema.

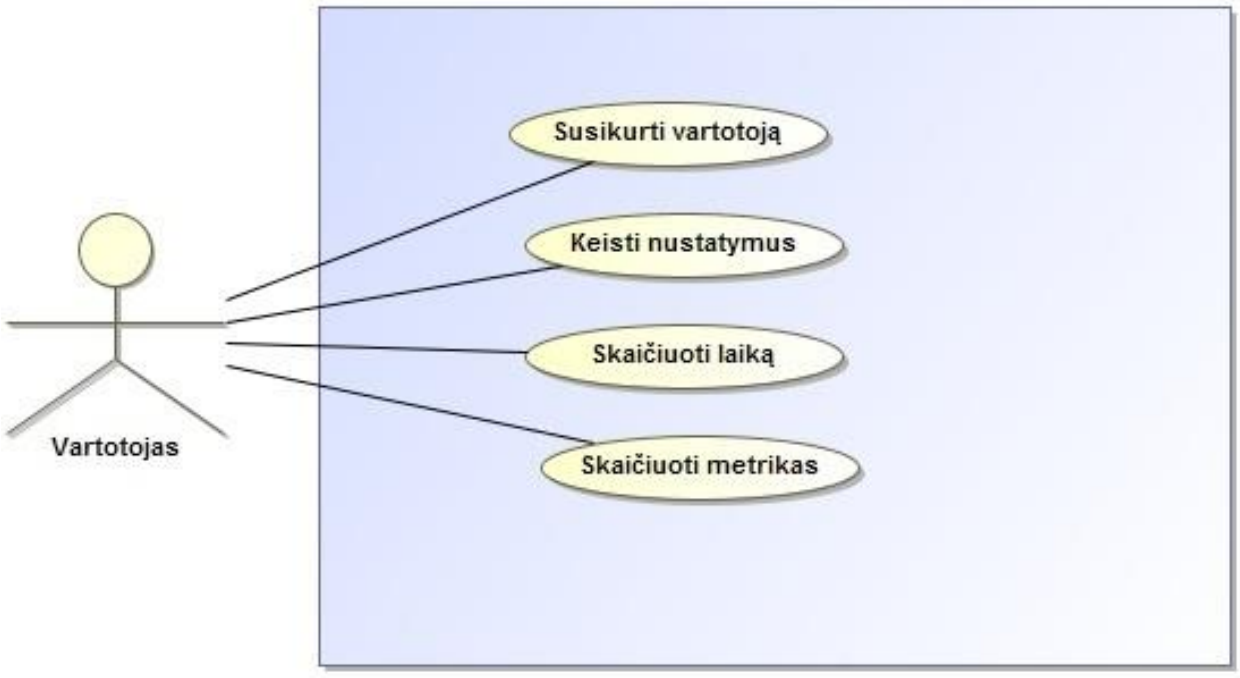


4 pav. Vartotojų duomenų bazės lentelės modelis

Kaip minėta ankščiau, vartotojų lentelė yra numatytoji duomenų bazėje, todėl jos modelis yra nekeičiamas. Galima matyti, jog vienas vartotojas gali turėti daug rolių duomenų bazėje. Todėl tai pateikta kaip skirtingų rolių priklausomybė masyvui.

7.2.Panaudos atvejų modelis skaičiavimo įskiepiui

Panaudos atvejų modelis svarbus sistemos kūrimo etapas. Jo metu pagal užsakovo reikalavimus sudaromas modelis, kokius veiksmus gali atlikti vartotojas su sistema ir kaip tai veikia sistemą. Panaudos atvejai – tai veiksmai, kuriuos lems vartotojas. Laiko ir programinės įrangos metrikų skaičiavimo įskiepio panaudos atvejų modelis pateiktas toliau.



5 pav. Panaudos atvejų modelis (1 įskiepis)

1. PANAUDOJIMO ATVEJIS: Susikurti vartotoją	
Vartotojas/Aktorius:	Neregistruotas vartotojas
Aprašas:	Procesas, kurio metu vartotojas susikuria vartotoją duomenų bazėje ir gauna priėjimą prie jos
Prieš sąlyga:	Vartotojas neturi priėjimo prie duomenų bazės
Sužadinimo sąlyga:	Sistemos paleidimas (užkraunamas įskiepis)
Po sąlyga:	Vartotojas registruotas duomenų bazėje ir turi prieigą prie duomenų (skaityti/rašyti)

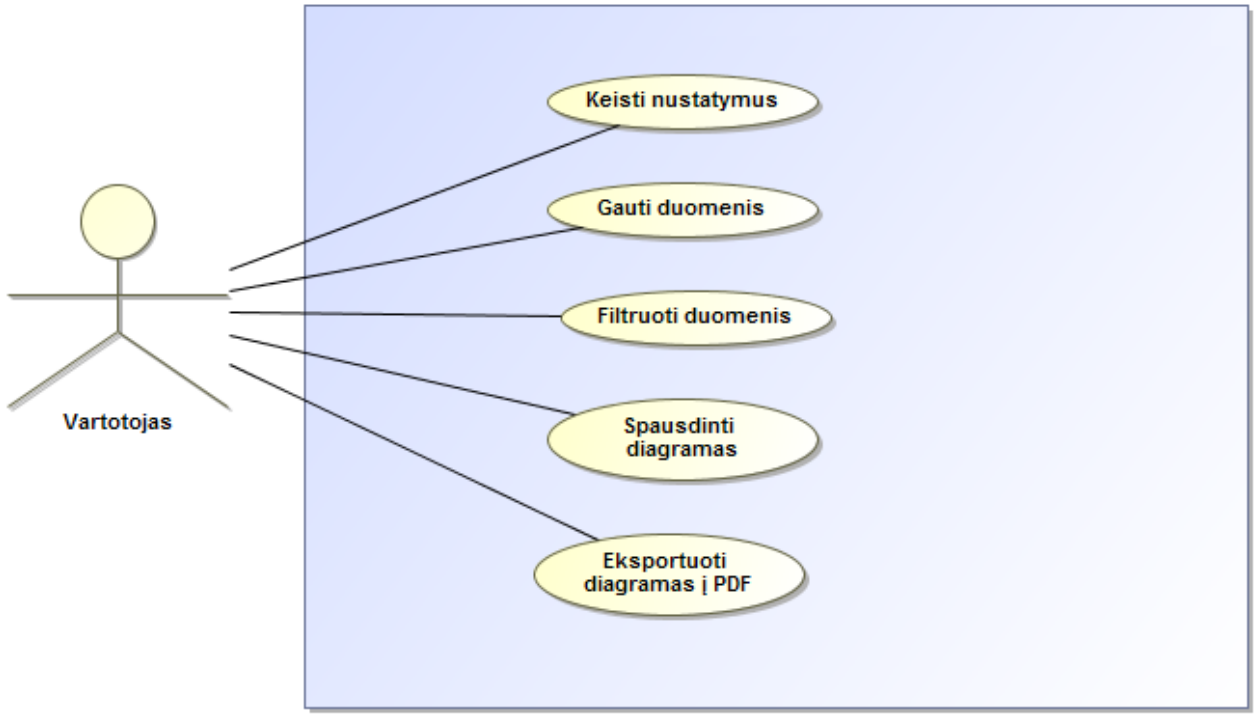
2. PANAUDOJIMO ATVEJIS: Keisti nustatymus	
Vartotojas/Aktorius:	Neregistruotas vartotojas, registruotas vartotojas
Aprašas:	Procesas, kurio metu vartotojas keičia prisijungimo, projekto tipo ir kitus nustatymus
Prieš sąlyga:	Vartotojas neturi pasirinktų nustatymų arba jie parinkti numatytieji
Sužadinimo sąlyga:	Atidaromas konfigūracijos langas
Po sąlyga:	Vartotojas turi specifinius sistemos nustatymus

3. PANAUDOJIMO ATVEJIS: Skaičiuoti laiką	
Vartotojas/Aktorius:	Registruotas vartotojas
Aprašas:	Procesas, kurio metu sistema skaičiuoja vartotojo darbo (programavimo) laiką
Prieš sąlyga:	Vartotojas neturi atidarytų klasių ir/ar projektų
Sužadinimo sąlyga:	Atidaroma pasirinkta klasė
Po sąlyga:	Programavimo laikas pradedamas skaičiuoti ir saugoti

4. PANAUDOJIMO ATVEJIS: Skaičiuoti metrikas	
Vartotojas/Aktorius:	Registruotas vartotojas
Aprašas:	Procesas, kurio metu sistema skaičiuoja vartotojo suprogramuoto kodo metrikas
Prieš sąlyga:	Vartotojas neturi atidarytų klasių ir/ar projektų
Sužadinimo sąlyga:	Uždaroma pasirinkta klasė
Po sąlyga:	Programinio kodo metrikos skaičiuojamos ir saugomos

7.3.Panaudos atvejų modelis atvaizdavimo įskiepiui

Atvaizdavimo įskiepis turi kitokias funkcijas nei skaičiavimo, tačiau naudoja bendrą duomenų bazės lentelę ir bendrus duomenis. Tai leidžia greitai ir efektyviai sekti duomenų pokyčius duomenų bazėje. Atvaizdavimo įskiepio tikslas – pateikti susistemintus, surinktus duomenis vartotojui, kur jis galėtų matyti visą savo darbo eigą, programinės įrangos kodo metrikas bei sugaištą laiką programuojant. Panaudos atvejų modelis pateikiamas toliau.



6 pav. Panaudos atvejų modelis (2 įskiepis)

1. PANAUDOJIMO ATVEJIS: Keisti nustatymus	
Vartotojas/Aktorius:	Neregistruotas vartotojas, registruotas vartotojas
Aprašas:	Procesas, kurio metu vartotojas keičia prisijungimo, filtravimo, diagramų tipo nustatymus
Prieš sąlyga:	Vartotojas neturi pasirinktų nustatymų arba jie parinkti numatytieji
Sužadinimo sąlyga:	Atidaromas konfigūracijos langas
Po sąlyga:	Vartotojas turi specifinius sistemos nustatymus

2. PANAUDOJIMO ATVEJIS: Gauti duomenis	
Vartotojas/Aktorius:	Registruotas vartotojas
Aprašas:	Procesas, kurio metu vartotojas gauna duomenis iš duomenų bazės
Prieš sąlyga:	Vartotojas neturi jokių duomenų vaizdavimo lange
Sužadinimo sąlyga:	Paspaudžiamas mygtukas, kuris gauna duomenis iš duomenų bazės
Po sąlyga:	Vartotojas gali manipuliuoti duomenimis, nes jie jau yra duomenų bazėje

3. PANAUDOJIMO ATVEJIS: Filtruoti duomenis	
Vartotojas/Aktorius:	Registruotas vartotojas
Aprašas:	Procesas, kurio metu vartotojas gali filtruoti duomenis pagal keletą parametrų taip atsirinkdamas tuos, kurių jau reikia
Prieš sąlyga:	Vartotojas turi visus bendrus duomenis
Sužadinimo sąlyga:	Pasirenkami filtro parametrai
Po sąlyga:	Vartotojas mato filtruotus duomenis

4. PANAUDOJIMO ATVEJIS: Spausdinti diagramas	
Vartotojas/Aktorius:	Registruotas vartotojas
Aprašas:	Procesas, kurio metu vartotojas gali matyti duomenis diagramų pavidalu

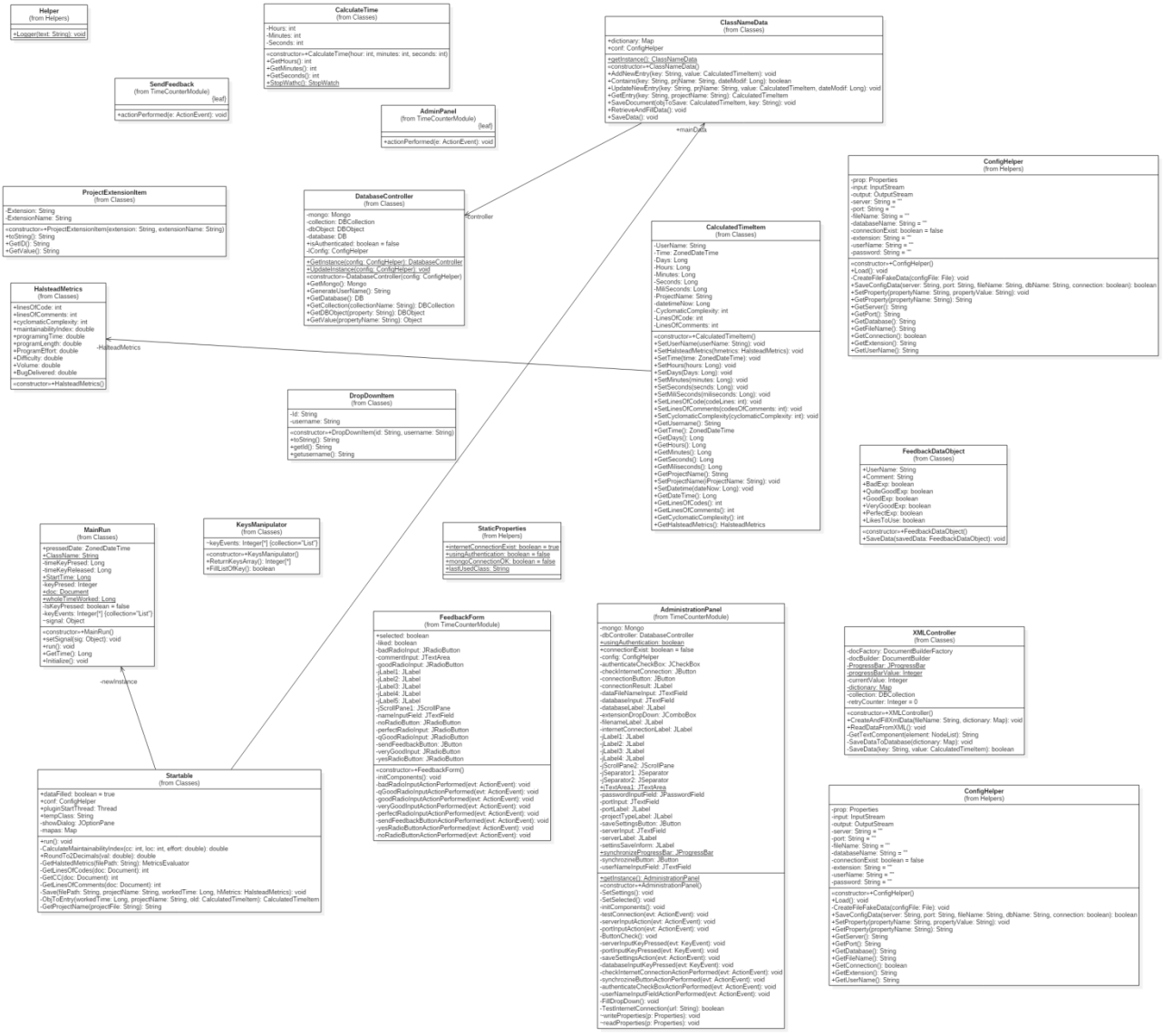
Prieš sąlyga:	Vartotojas filtruoja duomenis
Sužadinimo sąlyga:	Pasirenkamas diagramos tipas
Po sąlyga:	Vartotojas mato duomenis diagramų pavidalu

1. PANAUDOJIMO ATVEJIS: Eksportuoti duomenis į PDF dokumentą	
Vartotojas/Aktorius:	Registruotas vartotojas
Aprašas:	Procesas, kurio metu duomenys diagramų pavidalu eksportuojami į PDF dokumentą
Prieš sąlyga:	Vartotojas pasirenka duomenis, kuriuos nori eksportuoti
Sužadinimo sąlyga:	Pasirenkami duomenys ir paspaudžiamas eksportavimo mygtukas
Po sąlyga:	Vartotojas gauna PDF dokumentą su keliais variantais diagramų pagal pasirinktus duomenis

7.4.Klasių diagramos

7.4.1. **Klasių diagramos skaičiavimo įskiepiui**

Klasių diagrama laiko ir metrikų skaičiavimo įskiepiui pateikiama toliau. Joje parodytas klasių susietumas, metodai, kintamųjų matomumas kitų klasių atžvilgiu.



7 pav. Skaičiavimo įskiepio klasių diagrama

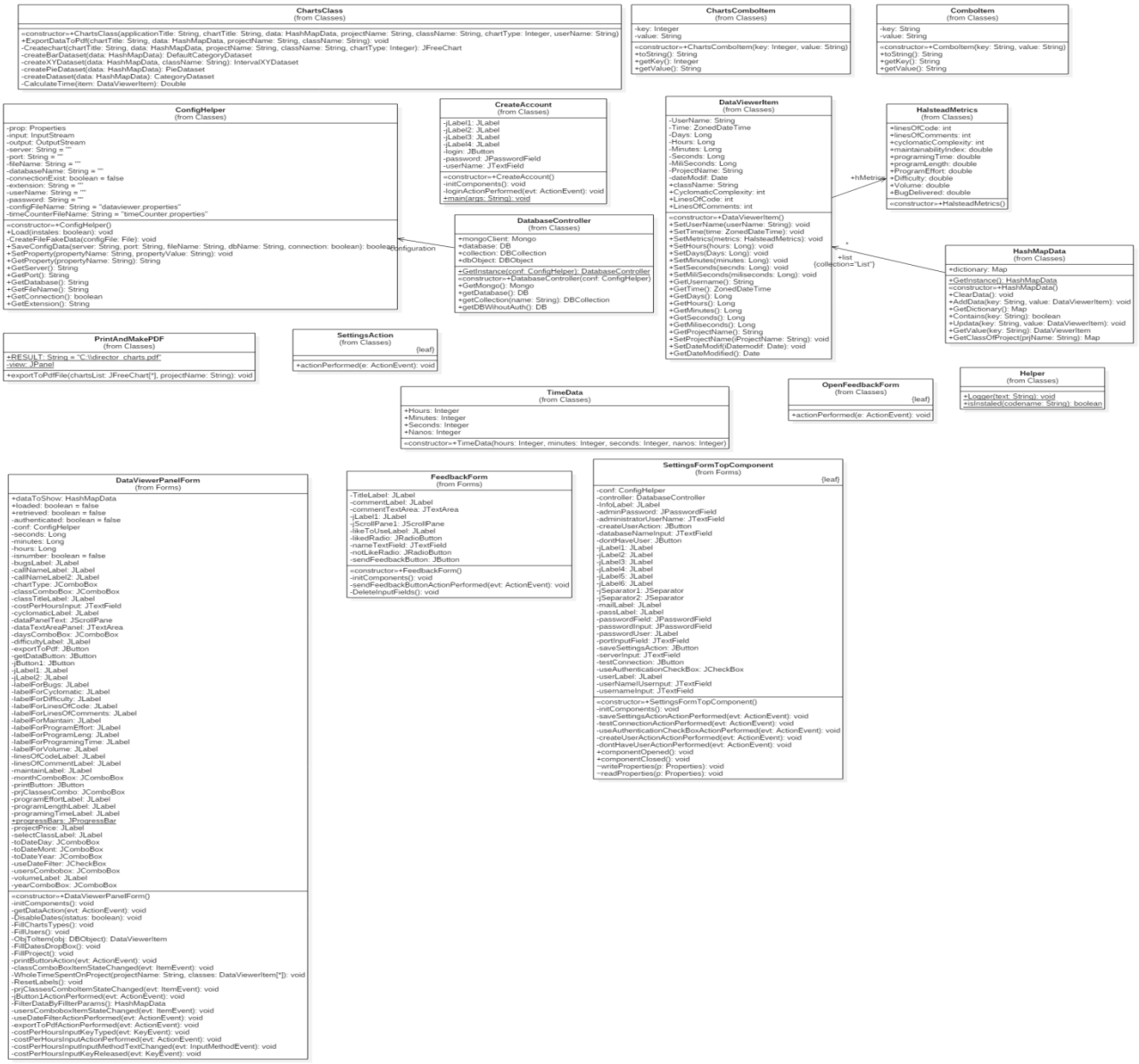
Aukščiau pateiktoje klasių diagramoje pateiktos pagrindinės klasės naudojamos programinėje įrangoje. Trečių šalių klasės nepateikiamos šioje diagramoje.

Iš klasių diagramos matome, kad trūksta ryšių. Naudojant atvirkštinę technologiją ir Star UML programinę įrangą iš klasių sugeneruota UML klasių diagrama. Pagrindinė klasė naudojama skaičiavimui yra Startable. Ji atsakinga tiek už visko užkrovimą, tiek už laiko ir metrikų skaičiavimą. Startable klasė naudoja MainRun klasę, kurioje atskiroje gijoje skaičiuojamas programavimo laikas, kurį programuotojas sugaišo rašydamas programinį kodą. Klasės uždarymo metu Startable klasė gauna iš MainRun klasės duomenis apie laiką ir toliau panaudodama klasės pavadinimą ir trečiųjų šalių bibliotekas skaičiuoja programinio kodo metrikas. Visa informacija, kaip matyti iš klasių diagramos, saugoma ClassNameData objekte, kuris ir yra siunčiamas į Mongo DB duomenų bazę. Tam reikalinga klasė DatabaseController.

Sąryšių trūkumas yra dėl to, jog panaudota programinė įranga nesugeba rasti klasės panaudojimo kitos klasės metode. Pvz. ConfigHelper klasė yra panaudota DatabaseController klasėje, tačiau sąryšio tarp jos nėra. Jei ConfigHelper klasės objektas būtų aprašytas prie klasės parametrų – ryšys egzistuotų.

7.4.2. **Klasių diagramos atvaizdavimo įskiepiui**

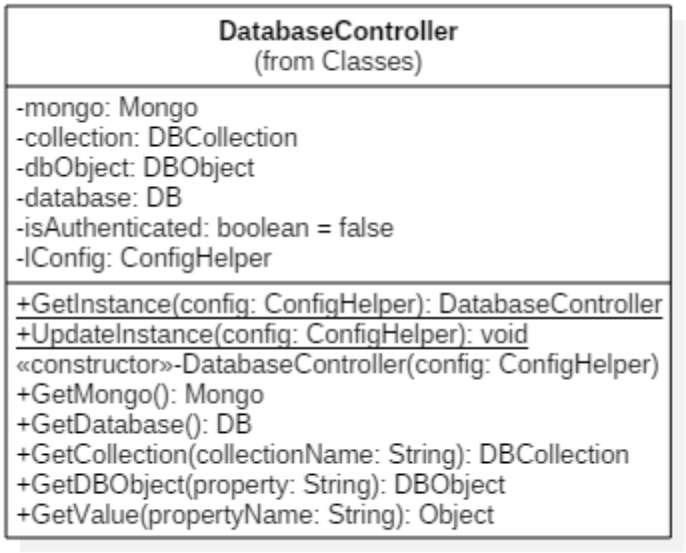
Toliau pateikiama klasių diagrama atvaizdavimo įskiepiui.



8 pav. Atvaizdavimo įskiepio klasių diagrama

Pateikta klasių diagrama sukauptų duomenų atvaizdavimui. Kadangi programinė įranga yra skirta atvaizdavimui, visi sukaupti duomenys vaizduojami programos lange. Tam naudojama DataViewerPanelForm klasė. Joje sudėti visi reikalingi elementai duomenų atvaizdavimui. Visi duomenys gaunami panaudojus DatabaseController klasę. Ji atsakinga už komunikaciją su duomenų baze. Tam naudojama ir ConfigHelper klasė, nes ji atsakinga už prisijungimo duomenų nuskaitymą iš failo. Panaudojus ConfigHelper klasę sukuriamas objektas, kuris panaudojamas ryšiui su duomenų baze. Visi duomenys, gauti iš duomenų bazės, sudedami DataViewerItem objektų sąrašą. Šis sąrašas panaudojamas visų filtro sąrašų ir kitų duomenų užpildymui. ChartsClass klasė naudojama sugeneruoti diagramas iš filtruotų duomenų, o PrintAndMakePDF klasė panaudojama, kad sugeneruotų PDF dokumentą. Šis dokumentas turi visų tipų diagramas pagal filtruotus duomenis.

Sistemoje labai svarbi programinės įrangos komunikacija su duomenų baze. Abu realizuoti įskiepiai naudoja tą pačią duomenų bazę. Tokiu atveju galima panaudoti bendrą duomenų bazės komunikacijos sistemą. Detalesnė duomenų bazės komunikacijos ir valdymo klasė *DatabaseController* pateikiama žemiau.



9 pav. *DatabaseController* klasės klasių diagrama

Aukščiau pateikta diagrama parodo komunikacijos valdymo klasę su MongoDB duomenų baze. Pagrindiniai reikalingi kintamieji:

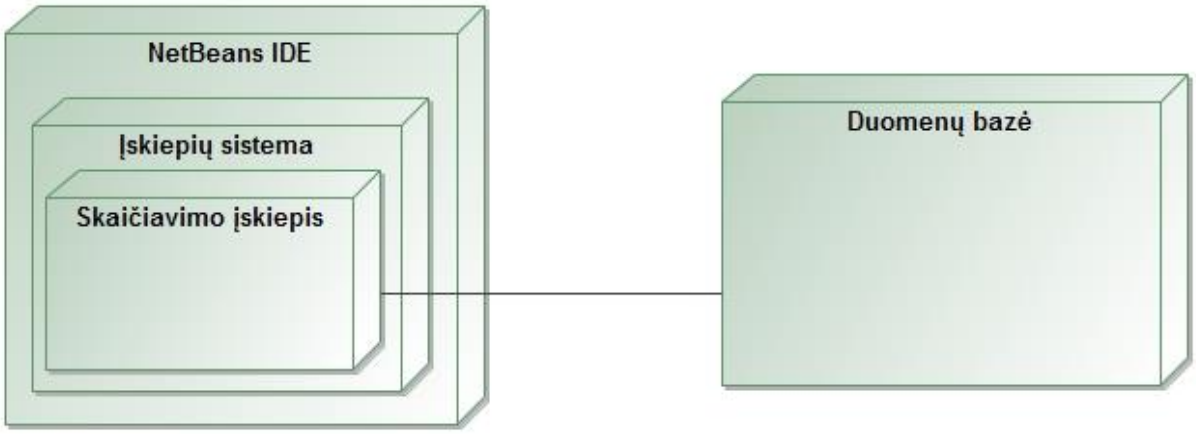
- Mongo – tai kliento bibliotekos objektas, reikalingas prisijungimui.
- Collection – kolekcija. NoSQL tipo duomenų bazėje nėra lentelių. Jos vadinamos kolekcijomis.
- dbObject – objektas, kuris yra nuskaitomas iš duomenų bazės ir paskui gali būti apdorojamas.
- Database – viename MongoDB serveryje gali būti kelios duomenų bazės. Šis objektas reikalingas gauti reikiamą duomenų bazę.
- isAuthenticated – lokalus kintamasis prieinamas iš išorės. Jis reikalingas žinoti, ar prie duomenų bazės sėkmingai prisijungta.
- lConfig – tai ConfigHelper tipo objektas. Jame saugoma visa reikalinga informacija apie prisijungimus. Ši informacija yra nuskaitoma iš .properties failo.

Metodai:

- GetInstance(config) – sukuria prisijungimą prie duomenų bazės. Kadangi MongoDB duomenų bazė priima tik vieną autentifikaciją, reikšminga panaudoti *Singleton* dizaino šabloną. Šio šablono panaudojimas garantuoja, kad visame kode bus tik vienas tokio tipo objektas su vienu prisijungimu prie duomenų bazės.
- GetMongo() – šis metodas skirtas gauti bazinį Mongo tipo objektą. Jis reikalingas tolesniems veiksmams.
- GetDatabase() – metodas grąžinantis konkrečią duomenų bazę kaip DB tipo objektą.
- GetCollection(collectionName) – metodas grąžinantis duomenų bazės kolekciją, jei tokia egzistuoja.
- GetDBObject(property) – šis metodas leidžia iš nuskaitytos duomenų bazės gauti įrašus.
- GetValue(propertyName) – šis metodas leidžia gauti iš duomenų bazės objekto, t.y. vieno įrašo, pasirinkto kintamojo reikšmę, pvz. palaikomumo indekso metrikos reikšmę konkrečiam įrašui.

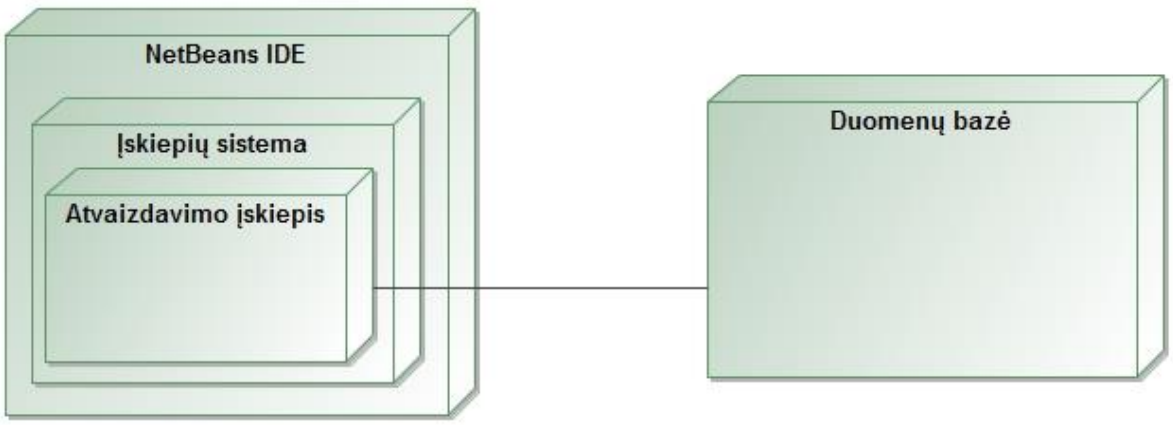
7.5.Architektūra

Kadangi programinė įranga yra kuriama kaip papildinys (įskiepis) egzistuojančiai programai, jos branduolys yra IDE (angl. Integrated development environment). Naudajamoje NetBeans IDE platformoje galima įdiegti ir kurti įskiepius. Pati NetBeans IDE platforma yra realizuota naudojantis JAVA programavimo kalba, todėl ir įskiepius kurti galima tik ja. Toliau pateikiama supaprastinta architektūros diagrama.



10 pav. Architektūros diagrama skaičiavimo įskiepiui

Pagal ankščiau pateiktą diagramą matome, kad programos įskiepis yra vienas iš sluoksnių NetBeans IDE sistemoje. Programavimo aplinka turi įskiepių sistemą, kurioje yra visi įskiepiai ir papildiniai sistemoje. Skaičiavimo įskiepis yra vienas iš sistemos elementų. Kadangi įskiepis duomenis siunčia į MongoDB duomenų bazę, todėl duomenys yra atskirti nuo pačios sistemos. Tai reiškia, kad duomenys gali būti siunčiami į nutolusią duomenų bazę (serverį) arba į veikiančią egzistuojančią MongoDB versiją lokaliame kompiuteryje. Skaičiavimo įskiepiui duomenys siunčiami į duomenų bazę. Kuriami arba atnaujinami. Atvaizdavimo įskiepiui reikalingas duomenų gavimas. Tačiau architektūra yra tokia pati.

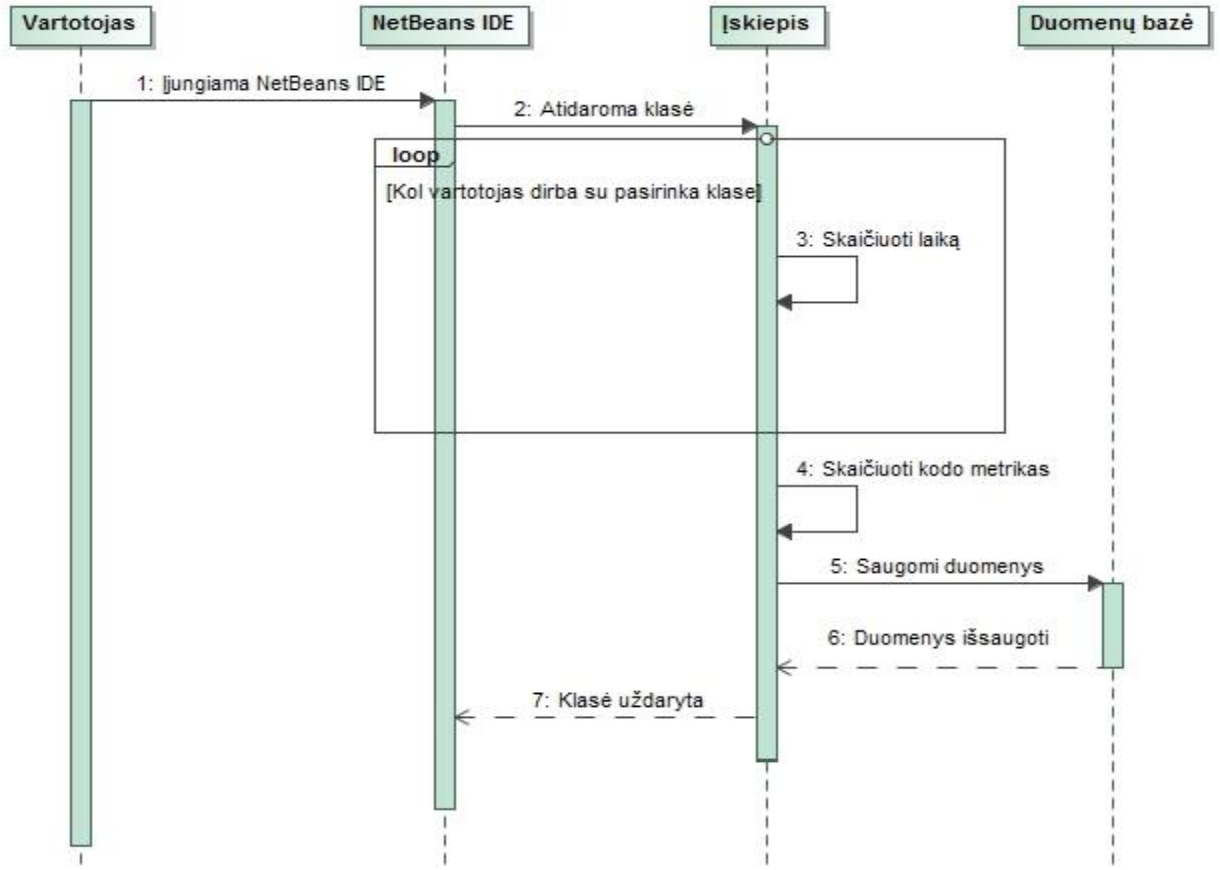


11 pav. Atvaizdavimo įskiepio architektūra

7.6.Sekų diagramos

7.6.1. Sekų diagrama skaičiavimo įskiepiui

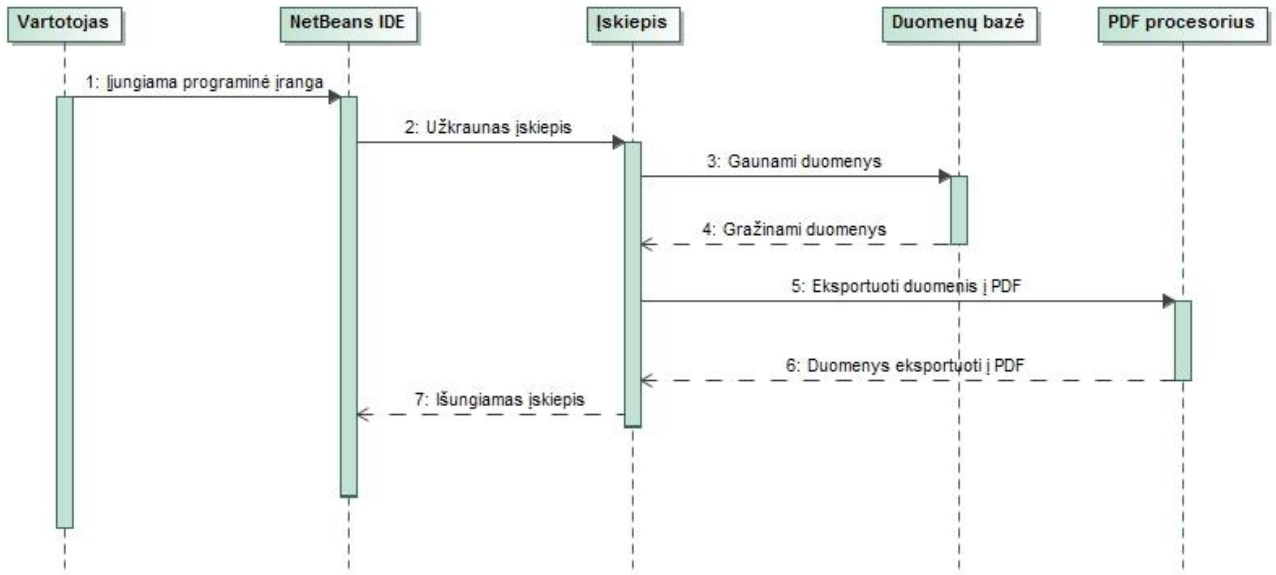
Sistema veikia turėdama tam tikrą seką. Vartotojas įsijungia programavimo aplinką NetBeans IDE ir automatiškai užkraunamas įskiepis. Kai įskiepis užkrautas vartotojas atsidaro projektą/klasę su kuria bus dirbama. Atidarymo metu paleidžiamas laikmatis, kuris skaičiuoja programuotojo darbą su konkrečia klase. Klasės uždarymo metu sustabdomas laikmatis ir gaunamas laikas darbo su klase – realaus kodo rašymo laiko. Uždarymo metu taip pat skaičiuojamos programinio kodo metrikos. Visa sekų diagrama pateikiama žemiau.



12 pav. Sekų diagrama skaičiavimo įskiepiui

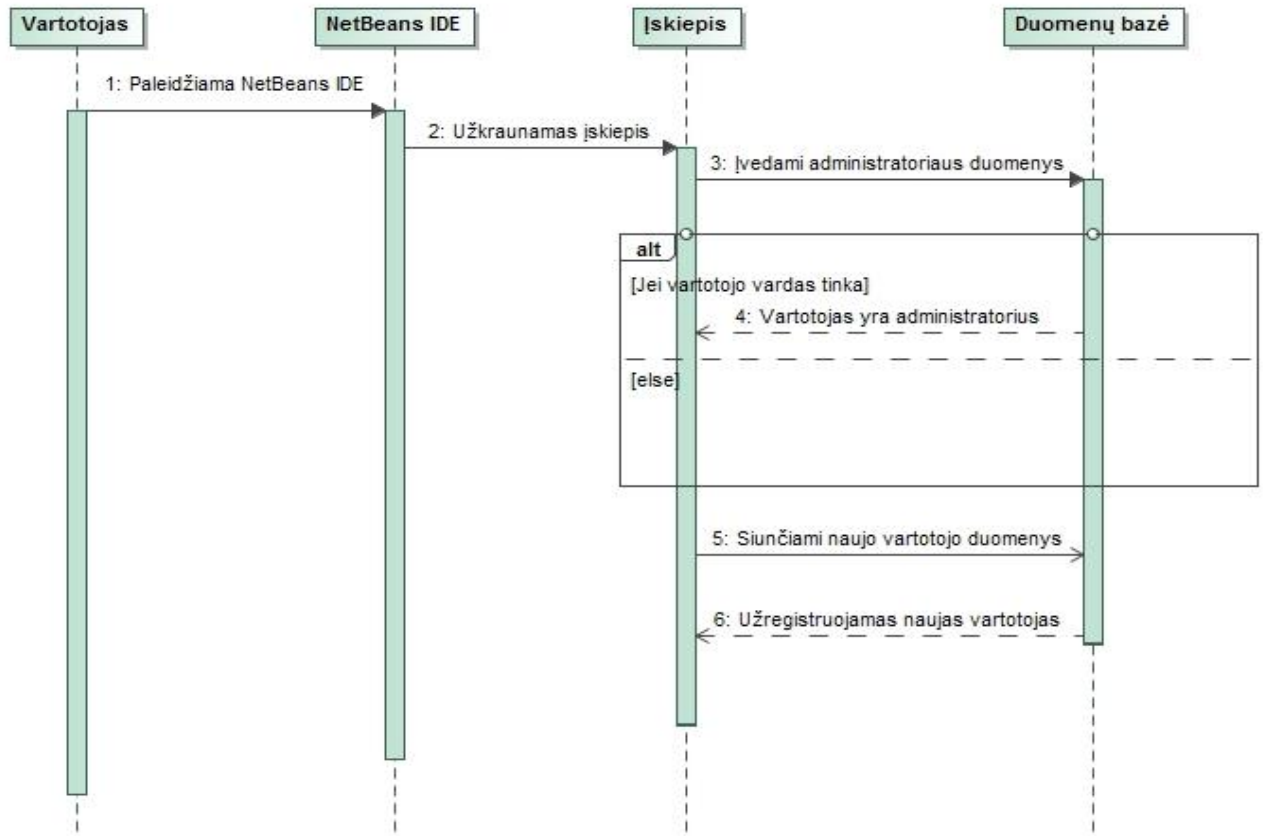
7.6.2. Atvaizdavimo įskiepio sekų diagrama

Atvaizdavimo įskiepis skirtas sukauptų duomenų analizei bei atvaizdavimui. Šis įskiepis turi dvi pagrindines funkcijas: atvaizduoti duomenis ir sukurti vartotoją duomenų bazėje. Vartotojo kūrimas reikalingas, jei norima sukurti naują vartotoją skaičiavimui. Tam reikalingas duomenų bazės egzistuojantis vartotojas, turintis rašymo teises.



13 pav. Sekų diagrama duomenų atvaizdavimui

Kadangi atvaizdavimo įskiepis turi funkciją sukurti vartotoją, toliau pateikiama vartotojo kūrimo sekų diagrama.



14 pav. Naujo vartotojo kūrimo sekų diagrama

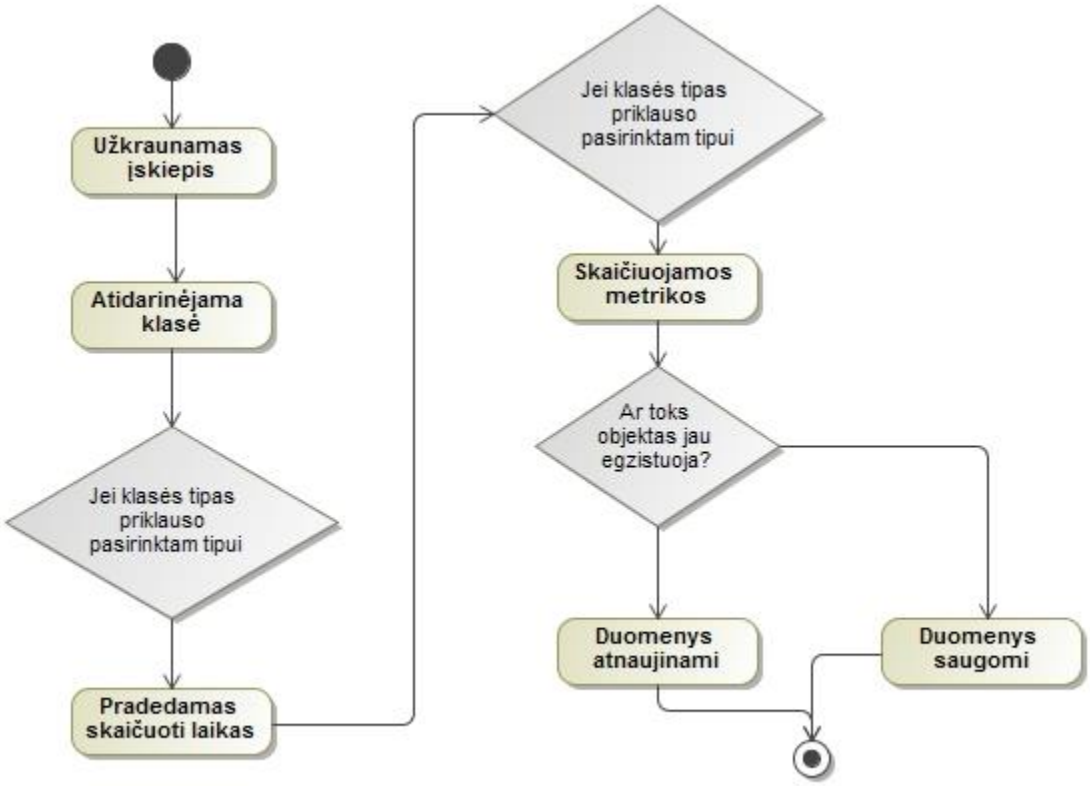
Vartotojui pasileidus NetBeans IDE programinę įranga užkraunamas sukurtas įskiepis. Jei vartotojas neegzistuoja arba norima sukurti naują, atidaromas naujo vartotojo kūrimo langas. Pirmiausia vartotojas, kuris naudojasi programine įranga, turi įvesti duomenų bazės naudotojo vardą ir slaptažodį. Įvestas vartotojas turi turėti rašymo teises į duomenų bazę. Kai administratoriaus vartotojas autentifikuojamas, įvedamas naujo vartotojo vardas ir slaptažodis. Siunčiama užklausa į

duomenų bazę, kad sukurtų naują vartotoją. Jei vartotojas neegzistuoja, sukuriamas naujas vartotojas duomenų bazėje.

7.7. Veiklos diagramos

7.7.1. Laiko skaičiavimo veiklos diagrama

Veiklos diagrama, skirta pavaizduoti vykstančių veiksmų seką sistemoje. Ji leidžia lengviau perprasti programinės įrangos vykdymo eiliškumą ir procesus. Veiklos diagrama gali parodyti ir sistemos architektūrinius sprendimus. Dažnai veiklos diagramos sudaromos kaip reikalavimas produkto kūrimui. Toliau pateikiama supaprastinta skaičiavimo įskiepio veiklos diagrama su baziniais elementais.

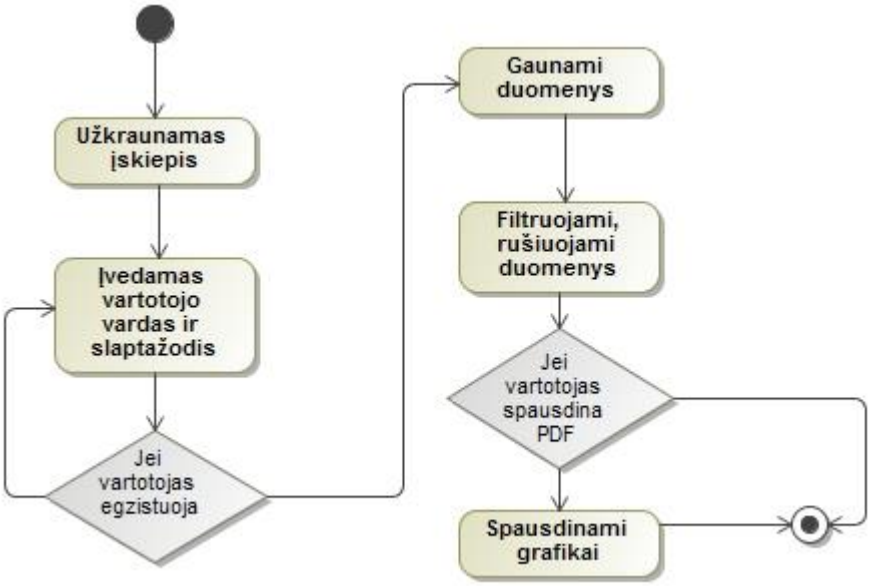


15 pav. Supaprastinta skaičiavimo įskiepio veiklos diagrama

Pateikta veiklos diagrama vaizduoja skaičiavimo įskiepio laiko skaičiavimo funkciją. Pirmiausia paleidimo metu užkraunamas įskiepis. Kadangi sistema turi nustatymų langą, jame pasirenkamas projekto tipas, kuris bus skaičiuojamas. Tai gali būti HTML, JAVA arba PHP. Jei nustatymuose pasirinktas klasės plėtinys sutampa su klase, kurią vartotojas atidaro, tada pradedamas skaičiuoti laikas. Po laiko skaičiavimo klasės uždarymo metu vėl tikrinama, ar plėtinys sutampa su nustatymuose pasirinktu plėtiniu. Jei viskas gerai, pradedamos skaičiuoti kodo metrikos. Prieš duomenų saugojimą atliekama egzistuojančio įrašo paieška. Jei toks įrašas egzistuoja, atnaujinami šio įrašo duomenys. Jei tokios įrašo nėra, sukuriamas naujas įrašas.

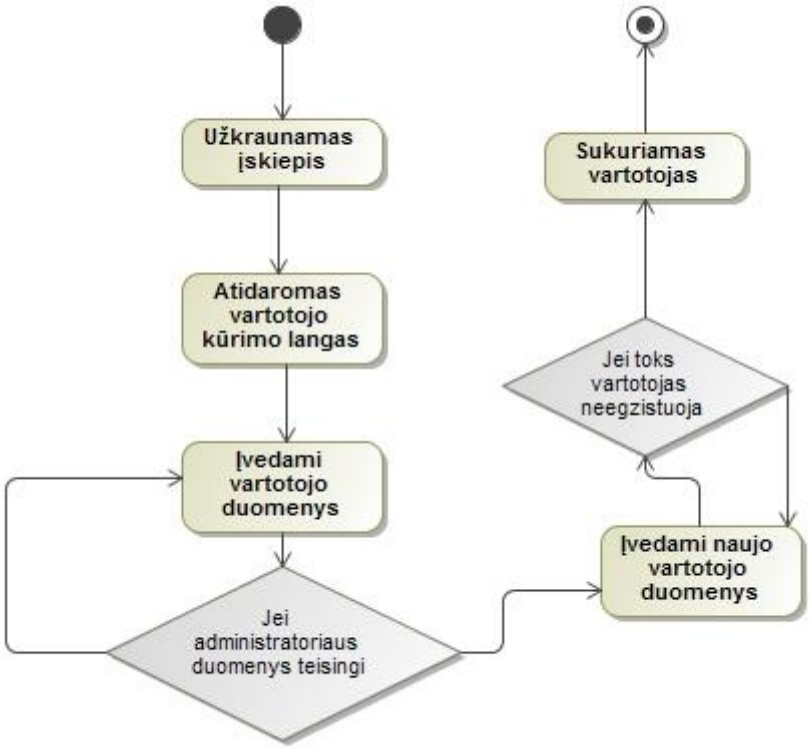
7.7.2. Veiklos diagrama atvaizdavimo įskiepiui

Atvaizdavimo įskiepis skirtas gauti ir pateikti duomenis vartotojui. Veiklos diagrama paprasta. Vartotojas gali filtruoti ir keisti parametrus. Esant poreikiui galima gauti grafikus ir juos eksportuoti į PDF dokumentą. Toliau pateikiama veiklos diagrama.



16 pav. Atvaizdavimo įskiepio veiklos diagrama

Vartotojų kūrimas gali būti atliekamas per atvaizdavimo įskiepį. Šiai funkcijai pateikiama veiklos diagrama.

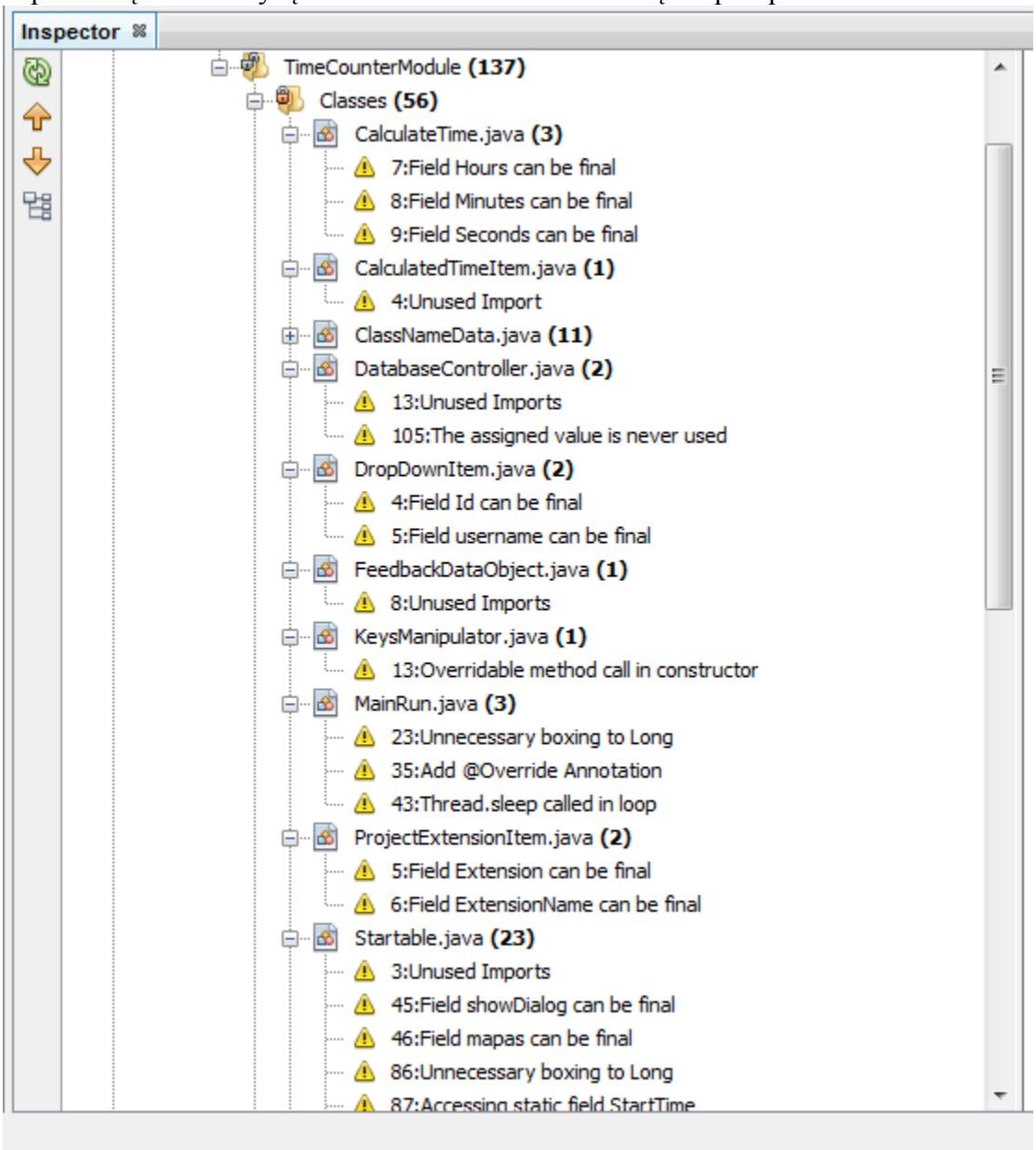


17 pav. Vartotojo kūrimo veiklos diagrama

Pirmiausia įjungiamas atvaizdavimo įskiepio langas. Atidarius vartotojo kūrimo langą pasirodo vartotojo vardo ir slaptažodžio įvedimo laukai. Įvedus administratoriaus tipo vartotojo prisijungimo duomenis tikrinama, ar vartotojas yra duomenų bazės administratorius. Jei vartotojas egzistuoja, įvedami nauji kuriamo vartotojo duomenys. Patikrinama, ar toks vartotojas yra. Jei jo nėra, jis sukuriamas.

7.8.Kokybinis įvertinimas

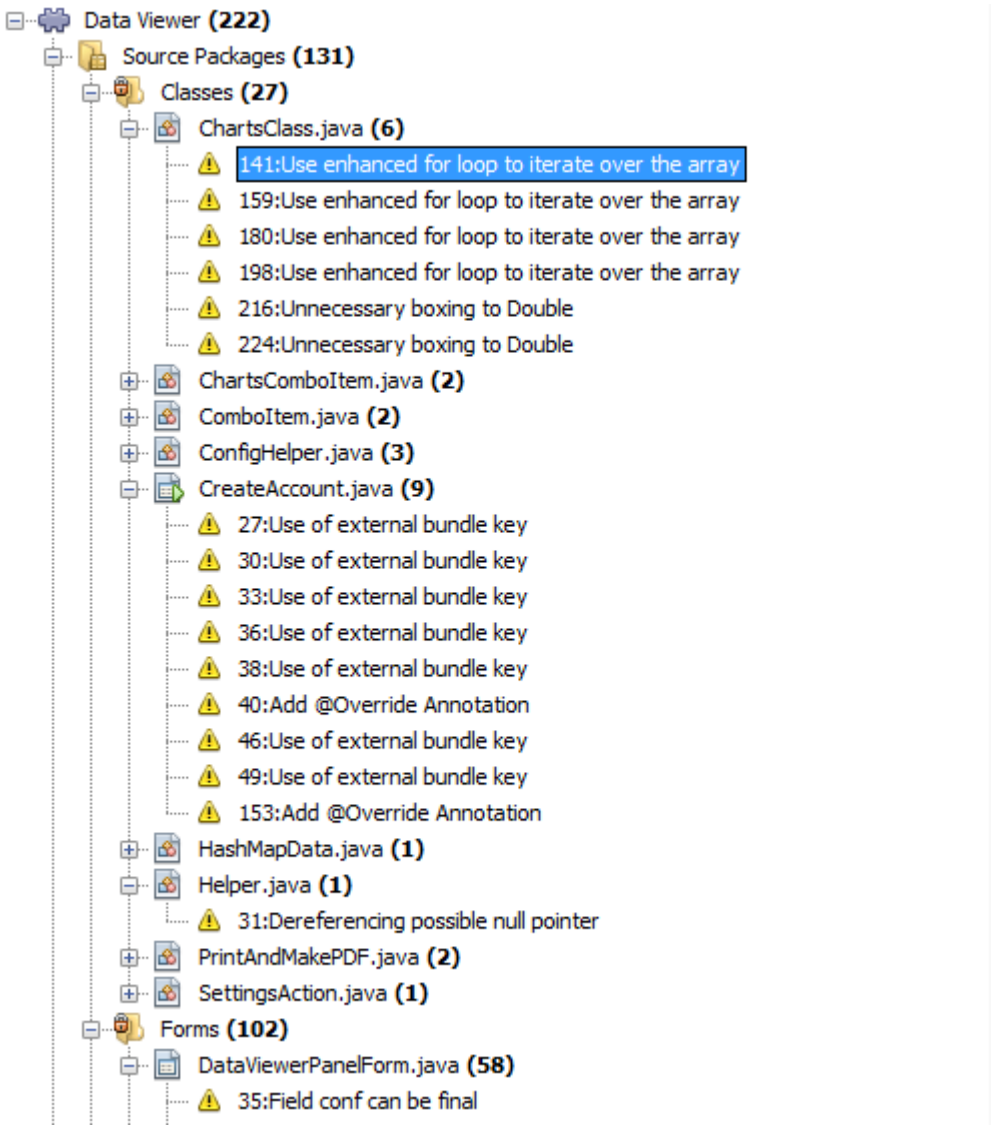
Sukurta programinė įranga realizuota JAVA programavimo kalba naudojant NetBeans IDE programavimo aplinką. Panaudojant programavimo aplinkos įrankius galima atlikti kodo statinę analizę, kad patikrinų kodo kokybę. Gauti rezultatai skaičiavimo įskiepiui pateikiami Toliau.



18 pav. Skaičiavimo įskiepio statinė kodo analizė

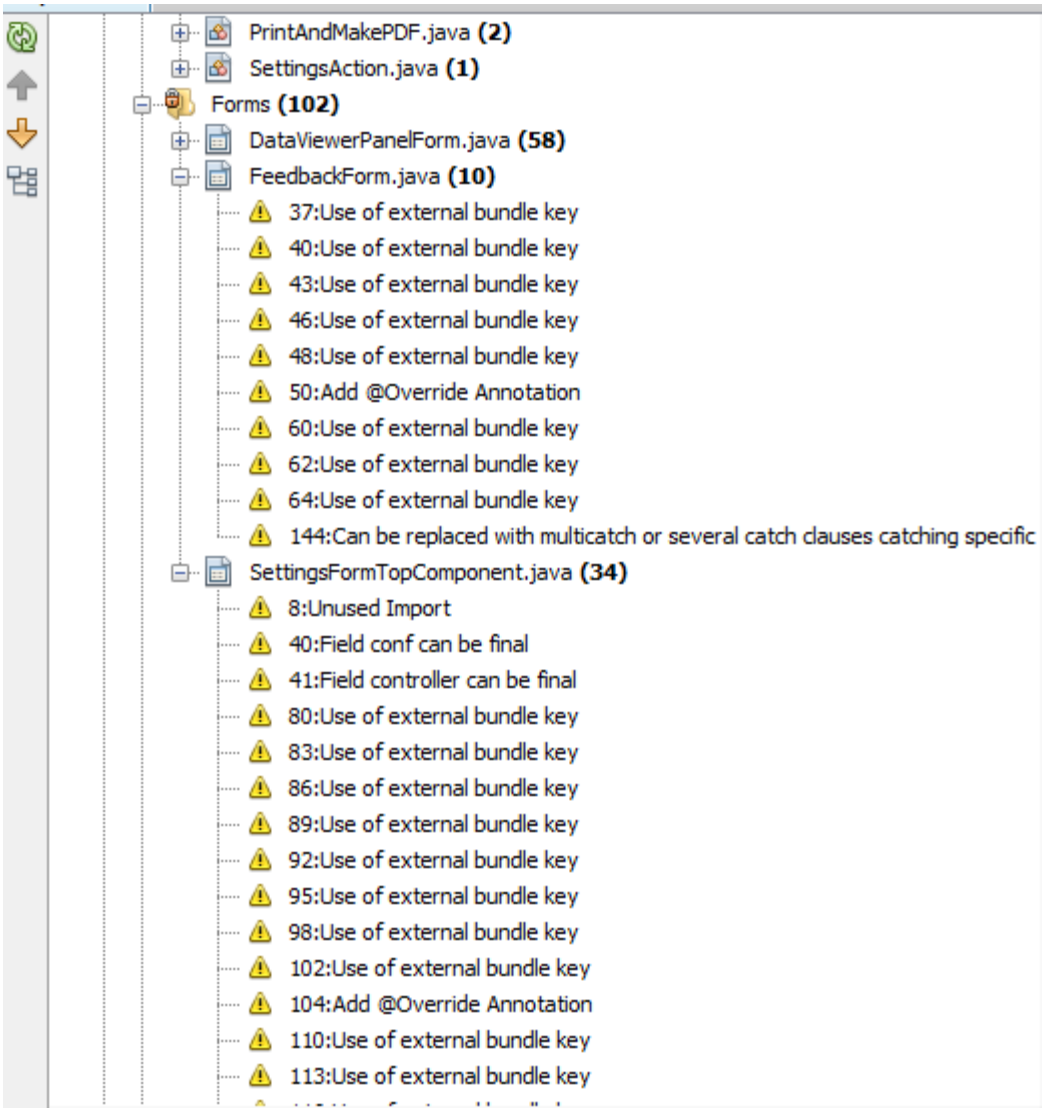
Pasinaudojus numatytaisiais programavimo aplinkos įrankiais galima matyti, kad klaidos programos kode yra nekritinės – tik įspėjimai. Pagrindinės problemos, kurios buvo surastos, tai nebenaudojami kitų klasių ar bibliotekų įterpimai (angl. include). Kitos klaidos: kai kurie kintamieji gali būti pažymėti kaip final tipo, reikšmės, kurios priskirtos, bet nepanaudotos. Tai nėra kritinė klaida, o labiau kodo pertekliškumas. Tai taip pat labai svarbu laikantis taisyklingos programavimo kultūros.

Atvaizdavimo įskiepiui taip pat labai svarbu atlikti kodo analizę. Žemiau pateikiami rezultatai.



19 pav. *Atvaizdavimo įskiepio statinės kodo analizės rezultatai*

Bendra kodo kokybė yra gera. Yra tik įspėjimai, kad kažkur galima pataisyti arba patobulinti kodą. Atvaizdavimo įskiepis daug darbo atlieka su duomenimis. Tam reikia naudoti ciklo sakinius, kad gautume gerus išfiltruotus duomenis. *Helper* klasėje yra viena galima *null* vieta, kurioje gali kažkas neveikti arba būti klaida. Reiktų atkreipti dėmesį į tokius pastebėjimus.



20 pav. Atvaizdavimo įskiepio statinės kodo analizės rezultatai(2)

Kadangi atvaizdavimo įskiepis paremtas daugiausiai formomis, kuriose atvaizduojami surinkti duomenys, daugiausiai klaidų ir yra minėtose formose. Klaidos nėra kritinės, labiau pastebėjimai, ką galima pakeisti. Tarp siūlymų yra ir panaudoti @Override anotaciją. Tai labai svarbu JAVA programavimo kalboje, ypač formų panaudojime. @Override notacija parodo, kad yra panaudojamas egzistuojantis JAVA metodas. Pavyzdžiui, jei turime metodą ToString() ir norime jį panaudoti, reikia uždėti @Override anotaciją.

8. Tyrimo metodologija

Tyrimo metu panaudota sukurta programinė įranga. Tačiau atlikti tam tikri pakeitimai programiniame kode. Kadangi sukurta programinė įranga turi didelį funkcionalumą, vartotojas gali keisti pagal save. Tačiau norint sukaupti duomenis tyrimui ir analizei reikia, kad duomenys būtų saugomi konkrečioje duomenų bazėje, kuri būtų prieinama tyrimo metu. Iš sukurtos programinės įrangos pašalinama vartotojo sąsaja. Joje būdavo galima keisti duomenų bazės prisijungimus bei projekto tipą, kuris bus skaičiuojamas. Kadangi viskas pašalinama, o tyrimas turi būti atliekamas su JAVA klasėmis, programiniame kode statiškai aprašomi klasių plėtiniai. Pakeista sistema skaičiuos programavimo laiką ir metrikas be jokios vartotojo sąsajos. Klasės uždarymo metu tik parodoma lentelė, kuri informuoja, kad skaičiuojamos programinės įrangos metrikos ir laikas. Ir visa tai saugoma duomenų bazėje.

Tyrimo metu didelis dėmesys skiriamas Halstead‘o metrikoms. Tam, kad galima būtų paskaičiuoti Halstead‘o metrikas, reikia panaudoti trečiųjų šalių biblioteką. Ji leidžia, nurodžius kelią iki klasės, kartu su klasės vardu paskaičiuoti visas galimas Halstead‘o metrikas. Kitos metrikos, tokios kaip kodo eilučių skaičius, ciklomatinis sudėtingumas, komentarų skaičius, palaikomumo indeksas skaičiuojamos atskirai. Viso tyrimo metu pagrindinis dėmesys skiriamas toliau išvardintoms metrikoms:

- Programavimo laikas (skaičiuojamas realus)
- Kodo eilučių skaičius – skaičiuojama visos eilutės atmetant tuščias eilutes
- Komentarų skaičius – komentarų skaičius kode. Skaičiuojama eilutėmis
- Ciklomatinis sudėtingumas – skaičiuojamas pagal parašytą kodą. Skaičiuojami sąlyginiai ir ciklo sakiniai visame programiniame kode.
- Programos ilgis – programos ilgis paskaičiuojamas panaudojus trečiųjų šalių bibliotekas.
- Programavimo laikas – teorinis programavimo laikas iš operandų ir operatorių skaičiaus. Pagal Halstead‘o aprašytą formulę.
- Programavimo pastangos – teorinės programavimo pastangos. Tai yra pastangos, reikalingos realizuoti algoritmą į programinį kodą.
- Sudėtingumas – Halstead‘o metrika, parodanti programinio kodo sudėtingumą.
- Apimtis – programinio kodo apimtis
- Palaikomumo indeksas – palaikomumo indeksas reikalingas spręsti apie kodo kokybę palaikymo atžvilgiu.
- Pristatomų klaidų skaičius – kiek klaidų gali būti programiniame kode. Šiam skaičiavimui naudojama supaprastinta formulė panaudojant papildomas bibliotekas.

Duomenys kaupiami į MongoDB duomenų bazę. Kad būtų užtikrintas duomenų bazės veikimas ir nebūtų nutraukiamas duomenų priėmimas, duomenų bazė paleista serveryje. Serverio darbo laikas yra 24/7. Kadangi duomenų saugojimui reikalingas vartotojo vardas ir slaptažodis, tam sukuriamas vartotojas be vartotojo įsiterpimo. Tai padaroma panaudojus kompiuterio MAC adresą. MAC adresas yra unikalus, todėl jis panaudojamas kaip vartotojo vardas ir slaptažodis. Programos kode yra ir administratoriaus vartotojo vardas bei slaptažodis. Prisijungiama su administratoriumi ir pridedamas naujas vartotojas pagal MAC adresą. Taip sukuriami unikalūs vartotojai.

Kartais gali būti poreikis matyti skirtingas programinio kodo metrikas pagal darbo su klase datą. Tam panaudojamas datos laiko elementas ir tai leidžia sekti duomenis. Skirtingų dienų darbo laikas saugomas kaip atskiras įrašas ir tai leidžia matyti progresą. Tačiau norint matyti galutines programinio kodo metrikas, reikia naudoti paskutinės dienos metrikas.

Atvaizdavimo įskiepis turi funkcionalumą, kuris rodo paskutinę saugotą reikšmę, jei nėra nustatyta filtravimo parametrų data. Tada matomos paskutinės ir realios programinės įrangos kodo metrikos.

9. Eksperimentinio tyrimo eiga

9.1.Eksperimento tikslas

Šio eksperimentinio tyrimo tikslas buvo nustatyti sukurtos programinės įrangos, skirtos sekti programavimo laiką ir suskaičiuotas programinio kodo metrikas, naudą programavimo kokybės ir projekto valdymo požiūriu. Įvertinti gautus duomenis, atlikti sukauptų duomenų analizę ir panaudojimo galimybes siekiant užtikrinti programinio kodo kokybę bei kodo metrikų sąveiką ir santykį su programavimo laiko kaštais.

9.2.Eksperimento uždaviniai ir sąveikos ryšiai

Sąveika tarp programavimo laiko ir paskaičiuotų programinio kodo metrikų:

- Programavimo laiko ir kodo eilučių santykis. Leidžia nustatyti programavimo greitį, kad ateityje leistų patikslinti projekto pristatymo laiką.
- Ciklomatinio sudėtingumo kitimas toje pačioje klasėje per laiką (datą). Tai leidžia patikrinti, ar klasei jos kūrimo metu galima pritaikyti kintančio sudėtingumo dėsnį, kuris parodytų, jog naujai kuriamo kodo sudėtingumas augo.
- Programavimo laiko, surinkto programinės įrangos pagalba, santykis su programavimo laiko paskaičiavimais iš programinio kodo pagal Halstead‘o metriką.
- Klaidų, kurios bus pristatytos sukurtai programinei įrangai, priklausomybė nuo realaus programavimo laiko ir teorinio, paskaičiuoto pagal Halstead‘o metrikas, programavimo laiko santykis.
- Klasių palaikomumo indekso tyrimas – kurios klasės gerai palaikomos, kurios ne.

9.3.Eksperimentinio tyrimo eiga programinės įrangos atžvilgiu

Eksperimentinio metu buvo modifikuota sukurta programinė įranga ir pakeista taip, kad rinktų anonimiškus duomenis iš vartotojų, kurie naudojasi sukurta programine įranga. Pirmiausia panaikinta vartotojo sąsaja, leidusi keisti prisijungimus prie duomenų bazės. Tai leido užtikrinti, kad duomenys bus siunčiami į duomenų bazę, kuri bus prieinama tik tyrėjui. Programinėje įrangoje pridėtas automatinis vartotojų generavimas pagal MAC adresą. Sukonfigūruota sistema pateikta NetBeans IDE įskiepių tinklapyje, kuriame įskiepiai prieinami viešai. Tai leido gauti duomenis iš atsitiktinių programuotojų ir užtikrinti duomenų anonimiškumą. Gauti duomenys surinkti į MongoDB duomenų bazę ir, naudojantis atvaizdavimo įskiepiu, duomenys surinkti analizei.

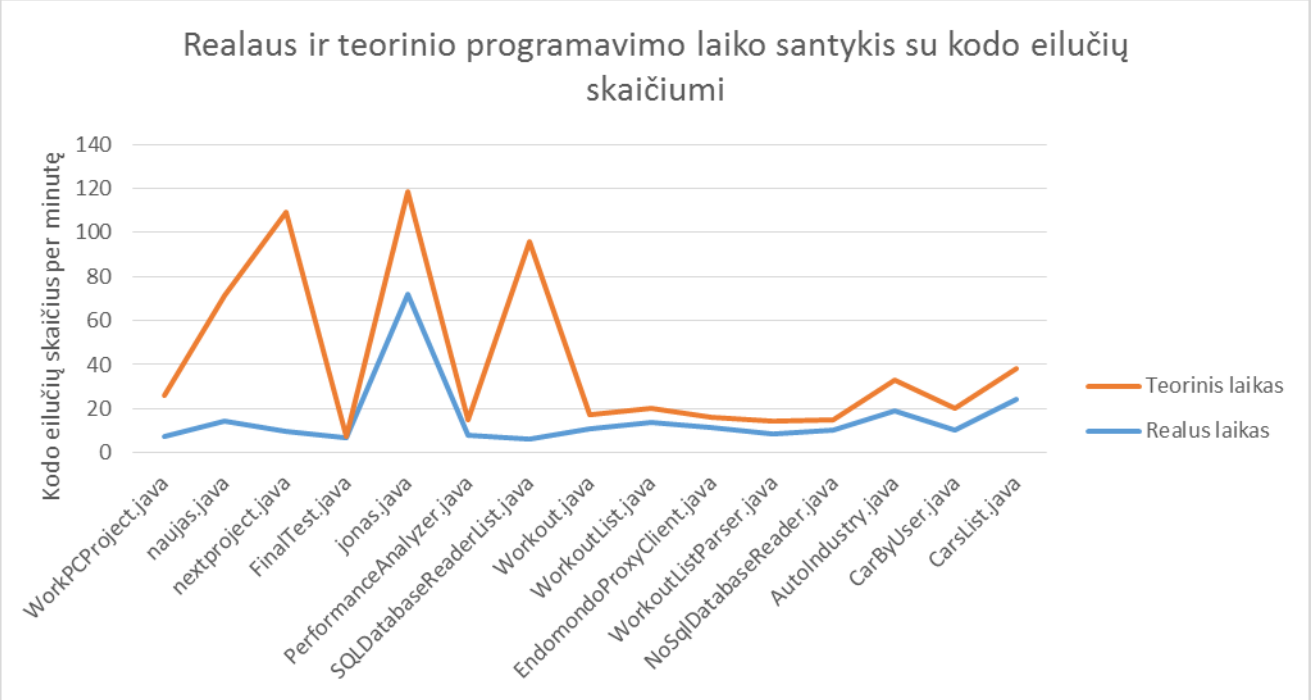
Įskipeių pasisiuntimo statistika pateikiama toliau:

- Laiko skaičiavimo įskiepis be programinio kodo metrikų – 145 parsisiuntimai. Laikotarpis: nuo 2016-01-24 iki 2016-05-13. Komentarai – problema dėl priėjimo prie failo teisių.
- Atvaizdavimo įskiepis – 148 parsisiuntimai. Laikotarpis: nuo 2016-01-24 iki 2016-05-13.Komentarų nėra.
- Laiko skaičiavimo įskiepis su programinio kodo metrikomis – 20 parsisiuntimų. Laikotarpis: nuo 2016-04-23 iki 2016-05-13. Komentarų nėra.

9.4.Eksperimentinio tyrimo rezultatai

Programinė įranga geba skaičiuoti tiek realų programavimo laiką, tiek teorinį, paskaičiuotą iš Halstead‘o metrikų. Pagal kodo eilučių skaičių ir programavimo laiko kaštus galima sužinoti programavimo greitį – kodo eilučių kiekį per laiko vienetą. Konkrečiu atveju imamas kodo eilučių

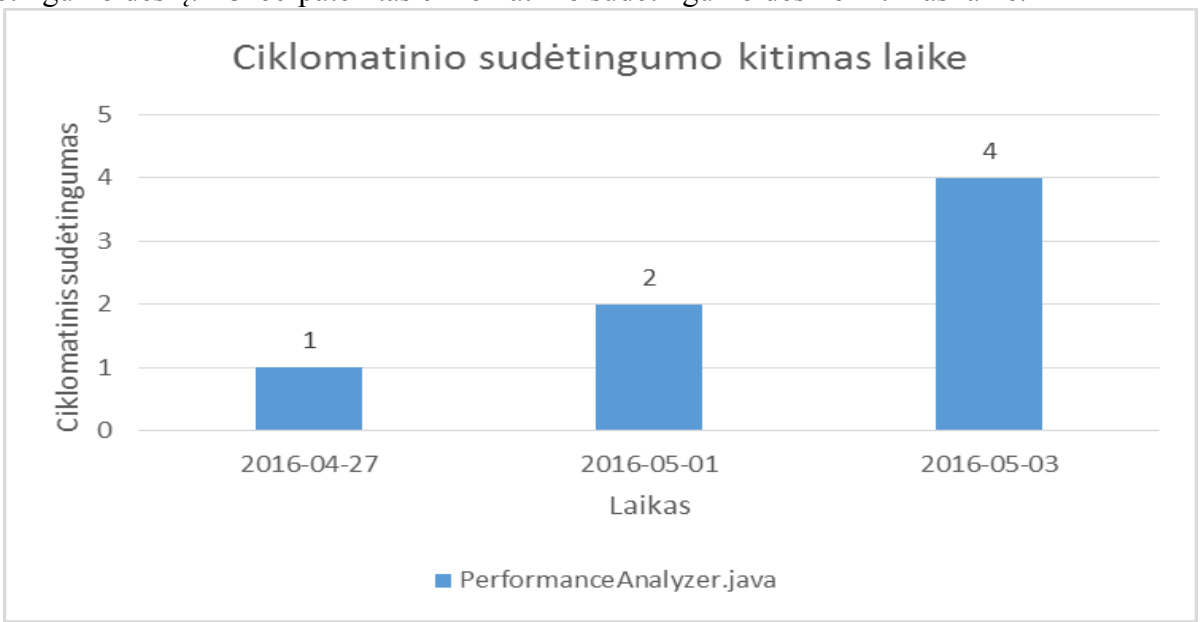
skaičius per viena minutę. Žemiau pateikiamas grafikas, kuriame pavaizduotas teorinis ir realus programavimo greitis.



21 pav. Realaus ir teorinio laiko santykis su kodo eilučių skaičiumi

Pateiktoje diagramoje pavaizduota realaus programavimo laiko, paskaičiuoto naudojant programinę įrangą, santykis su kodo eilučių skaičiumi. Santykis parodo realų programavimo greitį. Oranžine spalva pažymėtas teorinis programavimo greitis. Teorinis greitis – tai laiko, paskaičiuoto iš programinio kodo pagal Halstead'o programavimo laiko metriką, santykis su kodo eilučių skaičiumi. Jis parodo, koks programavimo laikas turėtų būti pagal operandus ir operatorius.

Ciklomatinis sudėtingumas parodo nepriklausomų kelių skaičių grafe . Tai leidžia spręsti apie programinio kodo realizavimo sudėtingumą. Ciklomatinis sudėtingumas sudarytas iš sąlyginių ir ciklo sakinių. Jei ciklomatinis sudėtingumas programuojant kinta, programiniam kodui, (klasei, funkcijai), galima pritaikyti *Augančio sudėtingumo* dėsnį. Šis dėsnis galioja programoms, kurios veikia savo aplinkoje ir evoliucionuoja kartu su aplinka. Augančio sudėtingumo dėsnis skelbia, kad evoliucionuojant programinei įrangai (ar programiniam kodui), sudėtingumas didėja, jei nėra atliekami sistemos palaikymo darbai. Iš surinktų duomenų rasta viena klasė, kuri atitinka augančio sudėtingumo dėsnį. Toliau pateiktas ciklomatinio sudėtingumo dėsnio kitimas laike.

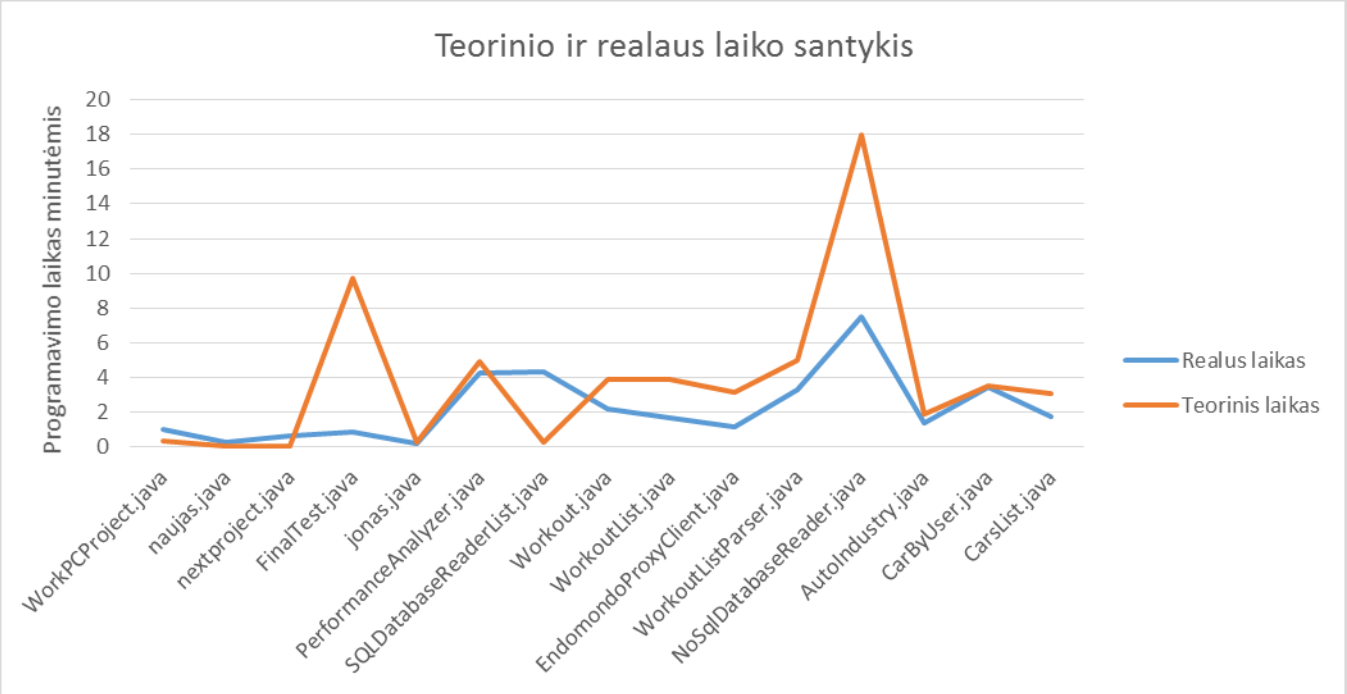


22 pav. Ciklomatinio sudėtingumo kitimas laike

Ši diagrama parodo klasės PerformanceAnalyzer.java ciklomatinio sudėtingumo kitimą laike. Jei kitas įrašas apie šią klasę turėtų nekintantį ciklomatinį sudėtingumą, būtų galima teigti, jog klasėje buvo atlikti palaikymo darbai ir sudėtingumas nebedidėjo. Tada Augančio sudėtingumo dėsnis nustoja galioti.

Turėdami programavimo laikus ir ciklomatinio sudėtingumo kitimą laike galima paskaičiuoti indeksą, kuris nurodytų ciklomatinio sudėtingumo augimo periodą. Tiksliam paskaičiavimui reikalingi dideli duomenų kiekiai, tačiau tam tikrus rėžius, kurie leistų nustatyti ciklomatinio sudėtingumo kitimo periodiškumą, galima paskaičiuoti iš turimų duomenų. Atitinkamai grafike pavaizduoti duomenys turi atitinkamas laiko reikšmes: 1 – 45s, 2 – 224s ir 3 – 254s. Laikas imamas sumuotas teigiant, jog programavimas buvo tęstinis ir laikas sumavosi. Tada apskaičiuojama kiek laiko buvo sugaišta rašant kodą su atitinkamu ciklomatinio sudėtingumo įverčiu. Gauti rezultatai parodo, jog laiko rėžiai, per kuriuos galima tikėtis ciklomatinio sudėtingumo augimo yra nuo 45s iki 112s. Vidutinis ciklomatinio sudėtingumo augimo periodas, pagal pateiktus duomenis, yra apie 73,5s.

Teorinis ir realus programavimo laikas yra labai skirtingas. Vienu atveju duomenų skaičiavimas gali būti paliktas (t.y. klasė atidaryta, o programuotojas nieko nerašo) ir laikas bus suskaičiuotas labai didelis, nors realaus kodo parašymo bus nedaug. Tačiau, net ir nerašant programinio kodo, laikas yra labai svarbus, nes programuotojai dažnai ieško informacijos, kaip išspręsti vieną ar kitą problemą, todėl tai reiškią programavimo laiką. Toliau pateikiama surinktų realių ir teorinių laiko kaštų santykio diagrama.

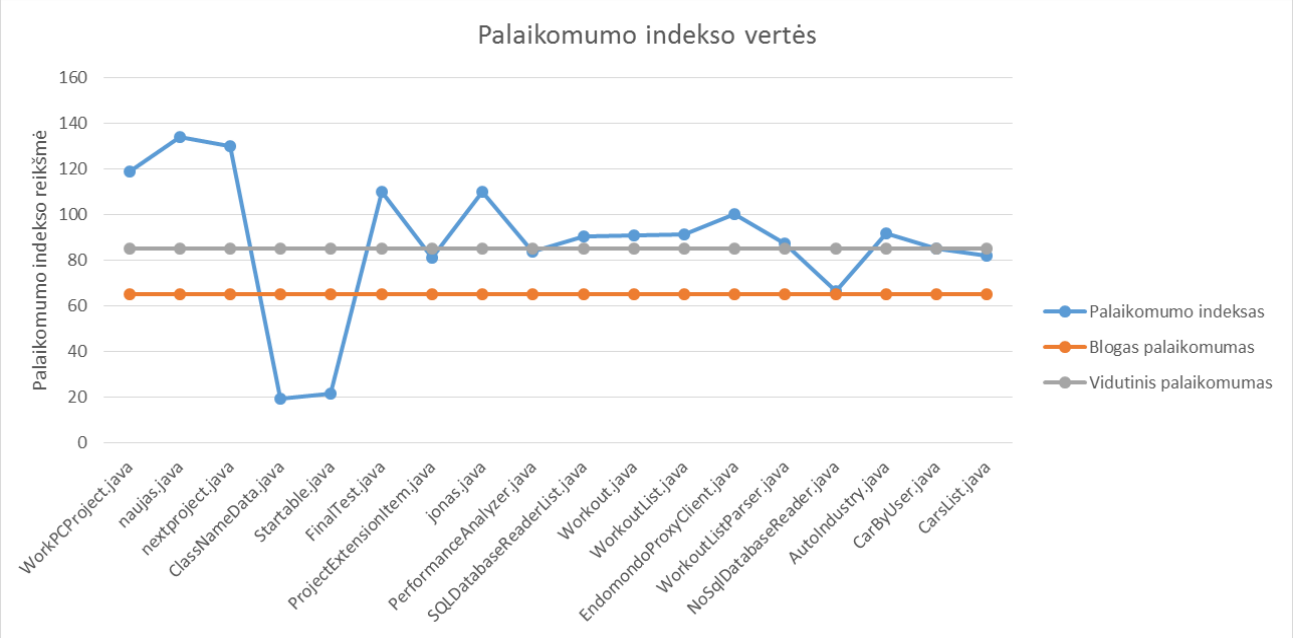


23 pav. Realaus ir teorinio laiko santykis

Iš aukščiau pateiktos diagramos galima daryti išvadas, kad skirtumas tarp realaus ir teorinio programavimo laiko yra didelis. Kaip matome, klasės NoSqlDatabaseReader.java programavimas teoriniu atžvilgiu turėjo trukti apie 18 minučių, tačiau realus programavimo laikas buvo apie 8 min. Tai parodo, jog didelis kodo kiekis per trumpą laiką gali labai iškreipti šį santykį. Jei kodas yra įklijuojamas iš kur nors, tai leidžia parašyti daug programinio kodo per labai trumpą laiką ir programavimo greitis didėja. Taip pat didėja ir skirtumas tarp realių ir teorinių laiko kaštų. Be abejo, yra ir klasių, kurių realaus ir teorinio programavimo laiko skirtumas nėra didelis. Tokių klasių kaip PerformanceAnalyzer.java, CarByUser.java, WorkPCProject.java skirtumas yra nedidelis.

Tyrimo metu surinkti duomenys ne tik leido stebėti sąveiką tarp programavimo laiko ir programinio kodo metrikų, tačiau ir daryti tam tikras išvadas apie programinio kodo palaikomumą.

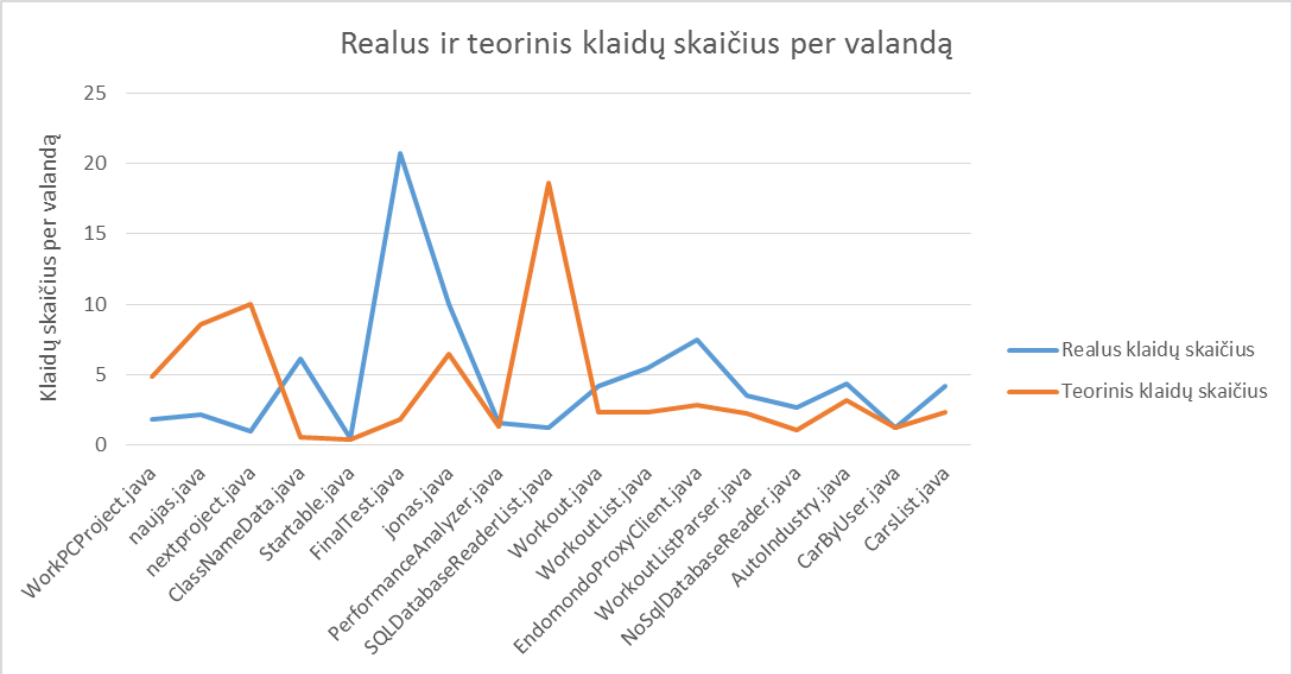
Programavimo metu programinis kodas turi būti rašomas taip, jog kitam programuotojui būtų lengva jį perprasti ir palaikyti. Toliau pateikiami palaikomumo indekso duomenys ir klasės.



24 pav. Palaikomumo indekso įverčiai

Iš pateiktų duomenų matome, jog 11 klasių turi gerą palaikomumo indeksą. Tai parodo, kad klasės lengvai palaikomos ir turi nedidelį ciklomatinį sudėtingumą. Ciklomatinis sudėtingumas, programavimo pastangos, kodo eilučių skaičius yra sudedamosios programavimo palaikomumo indekso skaičiavimui. Dvi klasės (*ClassNameData.java, Startable.java*) turi blogą palaikomumo indeksą. Tai rodo, kad klasės turi didelį ciklomatinį sudėtingumą, daug kodo eilučių ir didelę Halstead'o programavimo pastangų vertę. Pagal diagramoje pateiktus duomenis matome, jog didžioji dalis klasių turi didelį palaikomumo indeksą.

Klaidos programavimo kode yra neišvengiamos. Yra be galo sunku parašyti programinį kodą, kuris būtų be klaidų. Sukurta programinė įranga leidžia paskaičiuoti galimų klaidų kiekį programos kode. Kadangi sukurtos klasės nėra labai didelės, todėl spręsti apie klaidų kiekį yra sunku. Tačiau jei palyginsime programos kodo kūrimo laiką ir galimų klaidų skaičių, galima paskaičiuoti, tikėtina, klaidų skaičių programos kode programuojant valandą laiko. Toliau pateikiama diagrama su surinktais duomenimis ir paskaičiavimais.



25 pav. Realus ir teorinis galimų klaidų skaičius

Pateikta diagrama vaizduoja realių klaidų kiekį per valandą ir teorinių klaidų kiekį per valandą. Pagal duomenis matome, kad daugiau klaidų bus pristatyta pagal realaus laiko matavimus. Teoriniai paskaičiavimai išskaičiuojami iš programavimo pastangų (pagal Halstead‘ą). Panaudojus paskaičiuotas galimas klaidas pagal Halstead‘o metrikas, galimi rezultatai paskaičiuojami su realiu ir teoriniu laiku. Taip gaunama potencialus klaidų skaičius programuojant, nekeičiant programavimo greičio.

9.5.Eksperimentinio tyrimo rezultatai

Atlikus sukauptų duomenų analizę ir atlikus tam tikrus paskaičiavimus gautos išvados apie programavimo laiko koreliaciją su programinės įrangos metrikomis. Atliekant tyrimą gautas ir palygintas realus programavimo laikas ir teorinis programavimo laikas. Gauti rezultatai parodo, jog vienu atveju realus programinio kodo rašymas užtrunka ilgiau nei teoriniai paskaičiavimai, kitu atveju atvirkščiai.

Taip pat atliekant tyrimą pagal gautus rezultatus buvo galima nustatyti klases, kurioms galioja Augančio sudėtingumo dėsnis. Tai leidžia nustatyti klases, kurių sudėtingumas kyla programiniam kodui evoliucionuojant kartu su programos aplinka. Jei nėra atliekami palaikymo darbai, ciklomatinis sudėtingumas laikui bėgant ir kuriant programinį kodą tik auga. Panaudojus šiuos duomenis galima sekti klasės sudėtingumą laike.

Pasinaudojus surinktais programinės įrangos rezultatais galima nustatyti, kurios projekto klasės atitinka gero palaikomumo indekso įverčius, o kurioms turi būti atliekami tvarkymo darbai ir programinis kodas turi būti tvarkomas. Tai labai svarbu, kai programinė įranga vėliau bus palaikoma kito programuotojo.

Panaudojus surinktų duomenų analizę galima spręsti apie programinio kodo galimų klaidų skaičių. Panaudojus matematinius skaičiavimus, remiantis faktiniais duomenimis, kad kiti programavimo parametrai nekis, galima paskaičiuoti suprogramuojamų klaidų per valandą skaičių. Šie duomenys leidžia nuspėti galimų klaidų skaičių ir pagal tai pataisyti parašytą kodą.

10. Išvados

1. Analizės metu buvo išsiaiškinta apie projektui reikalingas technologijas, jų panaudojimą, veikimą, plėtrą, istorija. Išanalizuotas programinės įrangos gyvavimo ciklas ir jo panaudojimas programinės įrangos kūrime.

2. Analizės metu išsiaiškinti panašūs įrankiai skirti sekti programavimo laiką. Tai pat atlikta analizė įrankių, skaičiuojančių programinio kodo metrikas.

3. Suprojektuota ir realizuota programinė įranga (programavimo aplinkos įskiepis), matuojantis programavimo laiką, gebanti skaičiuoti programinės įrangos kodo metrikas ir saugoti duomenis tolesniam apdorojimui. Projektavimo metu surinkti ir išanalizuoti funkciniai ir nefunkciniai programos reikalavimai, sudaryos sekų, veiklos, klasių, duomenų bazių diagramos.

4. Tyrimo metu, pasinaudojus sukurta programine įranga surinkti duomenys apie programavimo laiką ir programinio kodo metrikas.

5. Atlikus tyrimą išsiaiškinta sąsaja tarp realaus suskaičiuoto programavimo laiko ir teorinio, apskaičiuojamo iš programinio kodo. Taip pat tyrimo duomenys leidžia spręsti apie klasės sudėtingumo augimą, kodo eilučių pokyčius. Rezultatai rodo, kad skirtumas tarp realaus ir teorinio laiko yra didelis ir priklauso nuo programuotojo. Tyrimo rezultatai leidžia paskaičiuoti ir viso projekto programavimo laiką.

6. Atlikus tyrimą išsiaiškintas vidutinis programavimo greitis konkrečiam objektui, palaikomumo indekso atitikimas ribinėms reikšmėms. Tyrimas padėjo nuspėti galimų klaidų kiekį per valandą pagal gautus duomenis.

7. Tyrimo metu, iš gautų duomenų nustatyta ciklomatinio sudėtingumo augimo periodiškumas, priimant faktą, jog palaikymo darbai nėra atliekami.

8. Sukurta programinė įranga yra viešai prieinama naudojimui, kaip atviro kodo programavimo aplinkos papildinys.

9. Mokslinis straipsnis apie programavimo laiko ir programinio kodo metrikų ryšį pristatytas *IVUS 2016* konferencijoje.

11. Literatūra

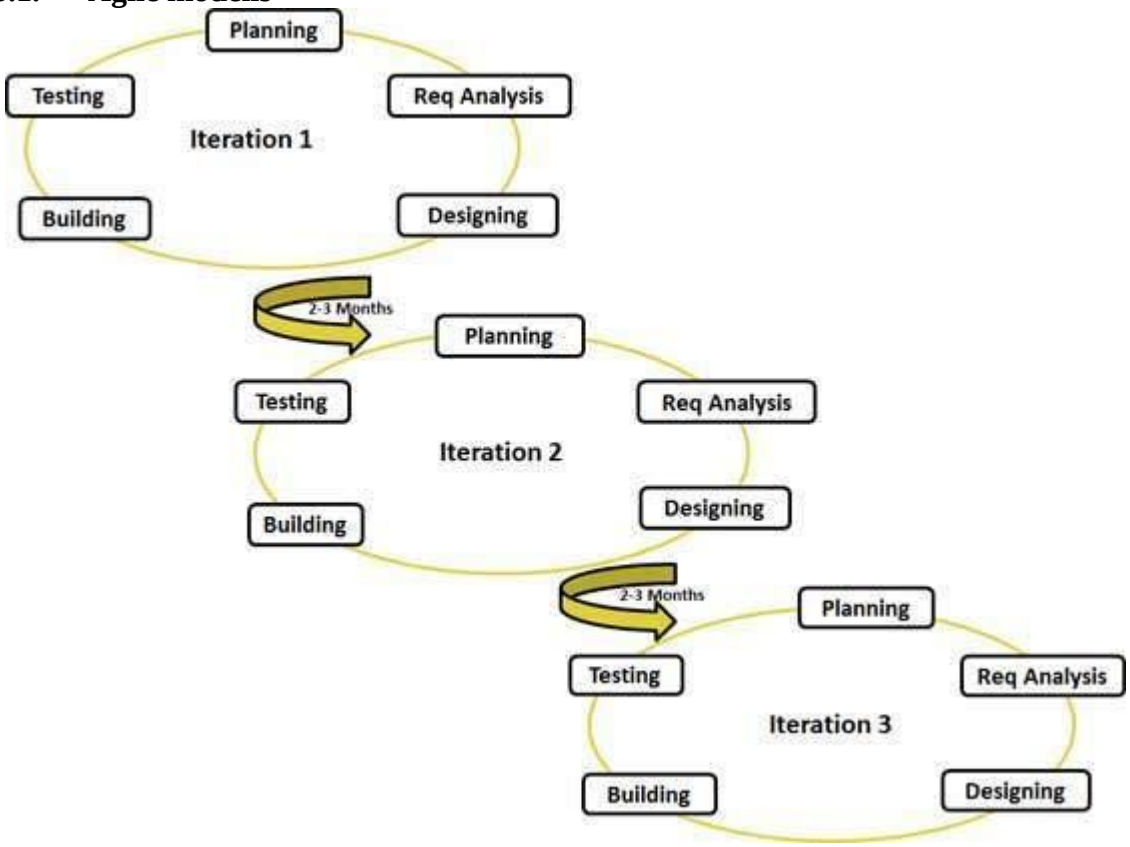
1. Software Development Life Cycle [Tinkle]. Prieiga per internetą: http://www.tutorialspoint.com/software_engineering/software_development_life_cycle.htm [kreiptasi 2016-04-06]
2. SDLC - Waterfall Model [Tinkle]. Prieiga per internetą: http://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm [kreiptasi 2016-04-06]
3. Eimutis Karčiauskas. *Programinės įrangos kokybės standartai, kodo metrikos ir kokybė, įrankiai* [Tinkle]. Priega per internetą: http://proin.ktu.lt/~ekartus/CaseTools2013/CT9_kokybe_12.ppt [kreiptasi 2016-04-07]
4. David Chappell. *The three aspects of software quality: functional, structural and process.* [Tinkle]. Prieiga per internetą: http://www.davidchappell.com/writing/white_papers/The_Three_Aspects_of_Software_Quality_v1.0-Chappell.pdf [kreiptasi 2016-04-07]
5. Stephen H. Kan. *Metrics and Models in Software Quality Engineering, Second Edition.*
6. Alain Abran. *Software Metrics & Software Metrology.* [Tinkle]. Prieiga per internetą: http://profs.etsmtl.ca/aabran/Accueil/ChapersBook/Abran%20-%20Chapter%20007.pdf [kreiptasi 2016-04-08]
7. Nur Islam. *Halsted's Software Science-An analytical technique.* [Tinkle]. Prieiga per internetą: http://www.slideshare.net/NurIslam5/halstedsc [kreiptasi 2016-04-08]
8. Margaret Rouse, *Integrated development environment (IDE).*[Tinkle]. Prieiga per internetą. http://searchsoftwarequality.techtarget.com/definition/integrated-development-environment [Kreiptasi 2014-11-15]
9. *NetBeans*. [Tinkle]. Prieiga per internetą: http://www.techopedia.com/definition/24735/netbeans [Kreiptasi 2014-11-15]
10. *What are plug-ins?*[Tinkle]. Prieiga per internetą: http://www.bbc.co.uk/webwise/guides/about-plugins [Kreiptasi 2014-11-15]
11. *Plug-in*.[Tinkle]. Prieiga per internetą: http://www.techterms.com/definition/plugin [Kreiptasi 2014-11-15]
12. http://www.techopedia.com/definition/3912/compiler [Kreiptasi 2014-11-15]
13. http://www.techopedia.com/definition/597/debugger [Kreiptasi 2014-11-15]
14. David Bolton, *What is Programming?*. [Tinkle]. Priega per internetą: http://cplus.about.com/od/introductiontoprogramming/p/programmers.htm [Kreiptasi 2014-11-17]
15. Eclipse documentation. [Tinkle]. Prieiga per internetą: http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Fint_eclipse.htm [Kreiptasi 2014-11-18]
16. Bill Powell, *What Is a CMS „Plugin"?*. [Tinkle]. Prieiga per internetą: http://cms.about.com/od/cms-basics/g/What-Is-A-Cms-Plugin.htm [Kreiptasi 2014-11-18]
17. Jake Rocheleau. *Beginner's Guide To WordPress Plugin Development.* [Tinkle]. Prieiga per internetą: http://www.hongkiat.com/blog/beginners-guide-to-wordpress-plugin-development/ [Kreiptasi 2014-11-18]
18. *Debugging*. [Tinkle]. Prieiga per internetą: http://searchsoftwarequality.techtarget.com/definition/debugging [Kreiptasi 2014-11-18]
19. *Compilers*. [Tinkle]. Prieiga per internetą http://www.cprogramming.com/compilers.html [Kreiptasi 2014-11-18]

12. Terminų ir santrumpų žodynas

LOC – angl. lines of code. Programinio kodo eilučių skaičius
IDE – angl. Integreated development environment. Programinė įranga skirta programinio kodo rašymui kaip pavyzdžiui: Visual Studio, Eclipse, NetBeans IDE.
Ciklomatinis sudėtingumas – funkcijos (klasės) nepriklausomų kelių grafe skaičius. Kitu atveju sąlyginių ir ciklo sakinių suma.
Palaikomumo indeksas – indeksas, nurodantis klasės kodo kokybę palaikomumo ir tobulinimo atžvilgiu.
Realus laikas – realus laikas, suskaičiuotas panaudojus sukurtą programinę įrangą.
Teorinis laikas – laikas paskaičiuojamas panaudojus Halstead‘o metrikas.
Metrika – kiekybinė arba kokybinė programinio kodo įverčio reikšmė.
MongoDB – nereliacinė duomenų bazė. Greitai praplečiama ir greitai veikianti.
NetBeansIDE – programavimo aplinka, realizuota JAVA programavimo kalba.
Eclipse IDE – programavimo aplinka paremta įskiepių apjungimu į bendrą platformą.
JAVA – objektinė programavimo kalba.
HTML – kompiuterinė žymėjimo kalba, naudojama pateikti turinį internete.
PHP – dinaminė interpretuojama programavimo kalba.
CSS – stilių aprašymo kalba naudojama turiniui internete.
Tortoise SVN – versijų valdymo programinė įranga.
NoSQL – nereliacinė duomenų bazės schema.
Klaida (angl. bug) – programinio kodo klaida, dėl kurios neveikia tam tikras funkcionalumas.

13. Priedai

13.1. Agile modelis



26 pav. Agile modelis

13.2. Straipsnis

Approach to evaluation of correlation coding time with software metrics

Laimonas Mikelionis
Kaunas University of Technology
Faculty of software engineering
Kaunas, Lithuania
Laimonas.mikelionis@ktu.edu

Eimutis Karčiauskas
Kaunas University of Technology
Faculty of software engineering
Kaunas, Lithuania
Eimutis.karciauskas@ktu.edu

***Abstract* - This paper shows how time spent on software developing correlates with metrics of software engineering. This leads to analysis of metrics of software engineering, what these metrics can describe and how it can describe quality of software. This is because each stage of software developing requires time. This will lead to correlation between concrete code metric and time spent on it. This paper describes how time calculation could be improved and collected data could be used for estimating projects in the future. This leads to estimating development time based on predicted value of metrics.**

Keywords - Time, Software Quality, Software Metrics, Estimates

Introduction

The idea of this article was to check how time is related with software metrics that are used by many companies and developers. This research helps to check how to evaluate your work, allows checking your developing speed and etc. Main goal of this experiment was to find connection between time and other basic software metrics and to have a method to collect data for future development and estimation. For this purpose the analysis of software metrics was made. Explaining of what it is, how to work with it and what advantages and disadvantages it gives for developer/customer/company. The experimental part is a description of improvement for existing time estimate methods. The result allows checking your developing speed, checking how the complexity of the software changes in the period of time. With collected data and estimated project size estimation of project time can be made. This allows estimating project times more accurate.

Description of Software metrics

Code metrics is one of the part of software engineering process that describes software quality and complex of software. As Linda Westfall explains "Software metrics are an integral part of the state-of the-practice in software engineering. More and more customers are specifying software and/or quality metrics reporting as part of their contractual requirements. Industry standards like ISO 9000 and industry models like the Software Engineering Institute's (SEI) Capability Maturity Model Integrated (CMMI®) include measurement. Companies are using metrics to better understand, track, control and predict software projects, processes and products."[2]. This statement describes how we should understand software metrics, what it is and how it is implemented in such industry standards like ISO 9000.

Nowadays customers can include software metrics in their requirements because it may help to improve quality and maintainability for software. But there is one of the challenges for software metrics because there are only few standardized mapping systems. This means that such software metrics like lines of codes (LOC) does not have standard measurement methods [2]. For instance, when measuring the lines of code we should decide how we will count them: "Do we count physical or logical lines of code? Do we count comments or data definition statements? Do we expand macros before counting and do
we count the lines in those macros more than once?"[2]. This is an abstract statement as we cannot have only one internationally accepted method for counting lines of code for software.

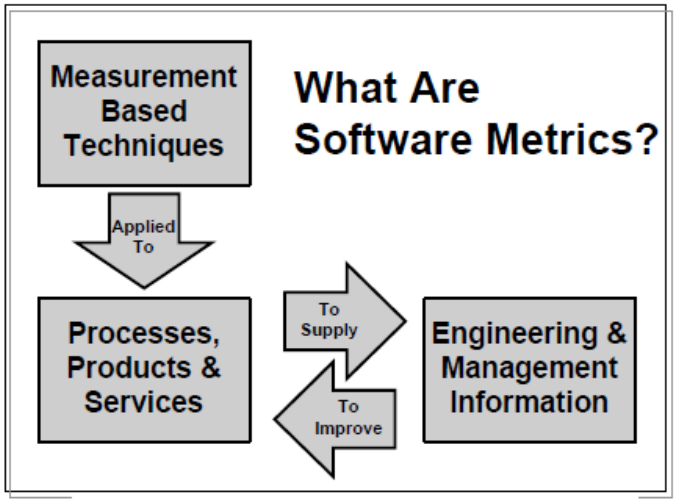


Figure 1. What are software metrics? [2]

As this article talks about time and other software metrics we must know how we can measure time for projects, for example, engineering hours. This is also very hard to calculate metric because there can be a question: "[…] do we include the effort of testers, managers, secretaries, and other support personnel" [2]. Although there are two metrics that has concrete and internationally accepted method to measure – it is McCabe's Cyclomatic Complexity and Function Point

Counting Standard. It is described by International Function Point User Group (IFPUG).
When focusing on time measurement we should go through some steps and make a model how we can calculate the time and how we can connect it with other software metrics. First, we must define customer. Customer can be actual customer of software or one of the software developing team. For example: testers, managers or developers. We want to calculate time spent on developing and it's correlation to other software metrics. This would be good for developers and managers. Managers could give report for customer about time spent on the software developing and progress of it. When customer is identified we have to define our goals - where these metrics results will be used and how. "Basili and Rombach define a Goal/Question/Metric paradigm that provides an excellent mechanism for defining a goal-based measurement program. Figure 2 illustrates the Goal/Question/Metric paradigm."[2]

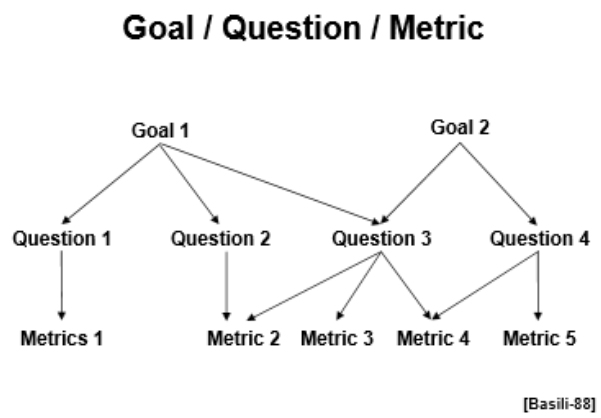


Figure 2. Goal/Question/Metric [2]

In this case our goal is to measure each software development stage time and how the code changes in measured time. This article explains not only time as a software metrics but also how it interacts with other metrics. We should choose what metrics we would like to correlate with. Better and easier way is to calculate: Lines of Code, McCabe's Cyclomatic Complexity and Function Point. These three metrics will be correlated with time spent while developing software.
Software metrics, that needs to be measured and correlated, is selected, but all metrics must have standard definitions [2]. Units how these metrics will be calculated must be selected. For this step it is good to use standard units. Lines of codes are measured by lines per file (class). McCabe's Cyclomatic complexity has his own algorithm of how it must be measured. Functional Points, in other words, is just functions calculated per file. Time must be measured in seconds.
There is an example of how time is expressed in other units, also as a metric of software. Alain Abran described about Halstead's metrics. His article explains that "Halstead's
Metrics are commonly known collectively as 'software science' [1]. "Researchers have used them:

- To evaluate student programs and query languages
- To measure software written for a real
- time switching system,
- To measure functional programs,
- To incorporate software measurements into a compiler, and more
- recently
- To measure open source software" [1].

These metrics measurement principles and metrics are included in number of current commercial tools that are used to count lines of code. According to Alain Abran, Halstead found a function how to express time as software metric. Halstead defined that "The required programming time (T) for a program P of effort E is defined as:

$$T = \frac{E}{S} = \frac{n_1 N_2 N \log_2 n}{2 n_2 S}$$

where S is Stroud number…[1]" "and the E is s defined as a measurement of the mental activity required to reduce a preconceived algorithm to a program P"[1]. N is sum of N_1 and N_2. And n is equal to $n_1 + n_2$. Halstead defined that all base metrics can be measured using these:"

- n1: Number of distinct operators.
- n2: Number of distinct operands.
- N1: Total number of occurrences of operators.
- N2: Total number of occurrences of operands"[1]

This allows expressing required time as a function.
The Stroud number used in the function can be described as "The Stroud number S, indicating the number of elementary decisions per second, satisfies this definition. Stroud's number has values between 5 and 25 elementary discriminations / second, 25 being the upper limit of images people can discriminate within a second."[3]. Halstead make a description of how the complexity of the software or program can be measured by having "measurable properties of the program. One of those properties is the language's vocabulary (number of distinct characters and signs, or operators and operands)." [3]. This gives definition about Stroud number. According to Stroud, in software science Stroud number is set to 18.

Agile and statistical methodology

Agile methodology has its own rules for measuring and estimating time for software development. Differently from waterfall method, Agile suggests doing all tasks in iterations. Iterations are a time frame in which some stories (in Agile stories means tasks) must be done. Length of iteration can vary on team and company but usually it is two weeks. In that time team/developer should finish the list of tasks that is assigned for this iteration. Stories must have a point value which represents difficulty of the story and time, which will be spent while developing software. Usually, there is three point scales: 1, 2 and 3 points. Each team/developer can choose its own system to set these points. For instance, some of the team estimates time for story using Fibonacci values: "so a very small story will be one point, a slightly larger one, two points, a slight larger one than that three points, then five points, eight points, 13 points, etc."[5]. This helps to know how many story's points did the team finished in

the iteration. At the beginning of Agile all teams will fail. This is because they have to do some of iterations and get a number of point values of the story that can be done in iteration. For instance, if a team at the beginning thinks that they can do 60 points in one iteration, they will fail and do for example 40. Then next time the team can predict point for iteration carefully and more accurate. After some time, team can become good and accurate Agile team. But this means that we know only point for the story and how many points we can do in one iteration. There is no meaning for estimates. There comes another way to include time into Agile methodology and to estimate time for the software development. The system can be called "ideal hours"[5]. This can be described as "A team that estimates using ideal hours thinks, "If we had no meetings, no appointments, and no distractions at all, how many hours would it take to finish this story?""[5]. Then team will track how many hours they have spent to finish the story and will calculate their velocity for the next iterations. However, "ideal hours" method is very optimistic but it never works. Typically, real hours will be half of the ideal hours. Even best of Agile teams cannot do better than ratio between real hours and "ideal hours" more than 3:4 [5]. In this case, the time estimate in Agile methodology can be done in many ways and it depends on the team/developer which way they are going to use. But when the team has velocity they can estimate times for next stories/project very accurate. But this is quite new and fresh Agile methodology. There is another way to estimate times not using Agile.

The statistical method explains more about why it is not possible to estimate time accurately:"

- The productivity and experience level of the engineer, particularly if multiple people are involved
- PTO, Late arrivals, early departures, sickness, etc.
- Unforeseen defect and customer requests, troubleshooting, challenges, system/environment issues, software/library issues, learning and ramp-up, design/architecture, required research, etc.
- The fact that software engineers notoriously underestimate
- Unforeseen issues with maintainability, architectural flaws/imperfections, scalability, performance, testability, etc.
- Time associated with spikes, R&D, design, architecture, mockup, prototype, POC, etc.
- Administrative work and non-engineering related requests"[4]

This reveals the problem that causes inaccurate time estimation for software development process. Some people think that "One may think that the only way to get a better estimate of time and delivery date is to try to subtask and think of everything **up front**, which involves more requirements gathering and documentation, mockups, prototypes, UML diagrams (sequence, use case, etc.), and so on."[4]. This method is opposite to Agile. This method is also wrong because of requirements. They can change before development start, for example, on testing level, on beta test, etc. This will lead to an inaccurate estimated time. Time estimate is very important in software engineering and it may cost a lot. There is theory that developers spent 5% of their time working with task related with estimating the time. It can cost for developer/company/team a huge amount of money.

How can we get rid of problems with inaccurate estimates but not use Agile? There is another approach. As Alex Castrounis described "that I use for characterizing the relative size of development tasks is a variation of the tee-shirt sizing method. Each task is given a relative size corresponding to five tee-shirt sizes, along with a very rough lead time estimate (for a single developer) as shown.

- XS: Half day or less
- S: Half day to one day
- M: Two to three days
- L: One week
- XL: One to two weeks

"[4]. Tasks longer that two weeks must be split into subtasks. This will lead to more accurate estimation of time and also more data collected to statistical methods, which will help to improve accuracy of estimate by using history of tasks with the same size.

Similar research to this article has been made by Wolfgang Holz, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. They made an assumption that they can predict software metrics by includes in the code [6]. They used 89 Eclipse plugins and made a *predictor* – a prediction software that predicts metrics. For example, this program can predict source lines of code by includes in the code. This prediction is made by using machine learning algorithm. They give a lot of data to analyze and predictor is *learning*. After that they ask *him* to predict software metrics for next project by number of includes. As mentioned in this article, this can be improvement for existing methods. In *predictor* method they can predict metrics and this described method can show correlation between metrics. Then *predictor* methods can also predict time for development. For example, if they predict that software size will be 10000 lines of code and described methods show that developer can write 10000 lines of code in 2 weeks then estimate for the project can be made.

There is also another model that can be used for predictions and estimations – COCOMO [7]. This method is also theoretical and can estimate efforts, duration, size and etc. of the software. This leads to the same conclusion as mentioned above. We can make prediction of how many lines will be used in the code, according to: inputs for the software, predicted number of lines of code, requirements type, methods and etc. This can predict duration and efforts of software, but it is in theoretical approach. COCOMO method can be connected with the method described in this article, using data collected through time, and using statistical methods - more accurate estimations can be done. This is because real practical data is more accurate than assumption.

Data for the method

Method that is going to be described in this section is a suggestion for existing methods, as an improvement for correct time estimates, and correlation between other software metrics. Method is based on both previously described methodologies – Agile and statistical. The idea

of this analysis is to collect data from previous task and to make a calculation to try to estimate next project/task/story time.
It is a part of statistical method because it is using time values for stories/projects or files and tells how much time you have spent on other projects, task or file (project class file or etc.). This will allow collecting data and using statistical methods and some calculations predict time for next project.
Agile methods come here too. As described above, Agile is based on giving stories (tasks) point and knowing how many points developer/team can complete in one of iteration. So from some perspectives this is also based on statistics. If in Agile, developers give stories' points; in this method other software metrics can be assigned to the story. For example, task is to write function to get data from database. Method can calculate time, cyclomatic complexity and functional points for the story and save it into database. Next time when similar story comes, collected data can be analyzed and estimation for a new story can be made. In that time, accuracy of estimation can be improved because every time when the story is completed new data comes to the database.
For this method there is some data that must be collected to make method work. It is described in the table below(Table 1).

Data type\Column name	ID	ProjectName	ClassName	TimeSpent	CyclomaticComplexity	FunctionalPoints	DateTime	TaskNumber	LineOfCode
	integer	string	string	int	int	int	DateTime	int	int

Table 1. Data that must be collected for methods.

As we can see by the data description in the table there is a lot of metrics that must be selected.
ID – it is unique identifier for record in the database.
ProjectName – every story/task depends to project.
ClassName – concrete file in which we are going to write code.
TimeSpent – calculated time for how long developer was working in this particular class.
CyclomaticComplexity – this metrics must be calculated using some methods and function or some third party software. But this should be done in IDE while working.
FunctionalPoints – this metrics, also as cyclomatic complexity must be calculated using methods or software.
DateTime – date and time when the record was inserted.
TaskNumber – if task/story that we working on has number it can be saved also here to map data for story. This allows you to map multiple records in database to one story/task.
LineOfCode – this also must be calculated using some methods or software.
Best practice to collect data is to develop extension of software development IDE and make it automatically collect data for specific tasks. This will lead to correct data and better accuracy in analysis stage.
When data is collected there can be time calculated that was spend on that task that we have in the database. This allows to sum all time. Then, exact time which was spent on the story can be entered in project management system. This will give some statistical data for future development. If the task for example is rejected or must be done in another way there can be an estimation made based on previously collected data.

Theoretical experiment

Let's assume that we have several lines in database. For example (Table 2):

ID	PrN	Class	Time	CC	FP	DT	TN	LOC
0	Test	Class 1	15	2	2	02/02/16	1	54
1	Test	Class 2	25	1	1	01/02/16	2	36
2	Test	Class 1	5	3	1	03/02/16	1	40
3	Test 1	Test1	26	2	1	04/02/16	11	69
4	Test 1	Class 5	105	5	4	04/04/16	7	105

Table 2. Example data to explain method

According to this data, we can check that there are two projects that were worked on and few different classes. There can be several conclusions done by this data.
First, data of several tasks are in database. This means that we can measure software metrics and time for these tasks. We can see that for task number 1 there are two records. For the first time developer added some code which is 54 lines length, cyclomatic complexity is 2 and number of functional points is 2. Then developer did some changes in the code. After those lines of codes decreased by 14 lines, cyclomatic complexity has increased by one and functional points have decreased by one. There is really important to check datetime value in database, because latest data is the real one. Other one is change history but it helps to see time and other software metrics changed through development. Below there is example how data can be used for statistics (Figure 3).

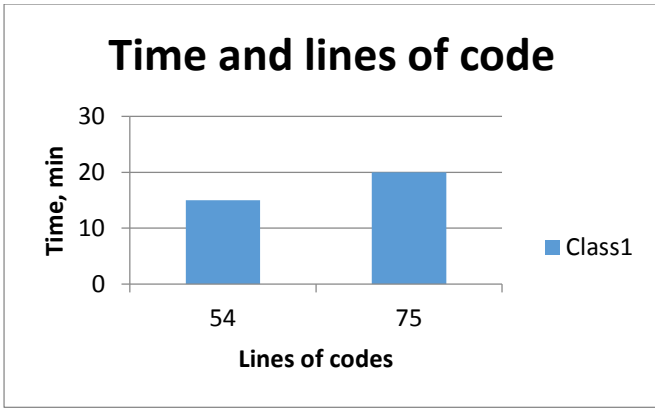


Figure 3. Time and lines of code correlation

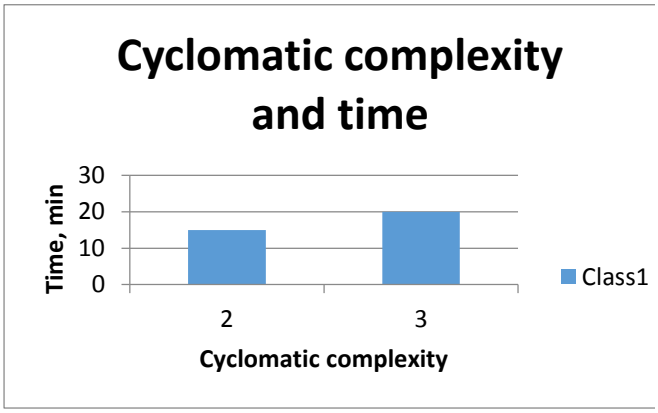


Figure 4. Cyclomatic complexity and time correlation

So for class Class1 whole time spent is 20 minutes and the total line of code in that file is 75. We can do prediction that this will also work with other software metrics and data.

results Theoretical of Experiment

As we can see in the diagrams there is quite easy to see how software metrics is changing by the time spent for the project/task/class. This means that when we have big amount of data collected we can calculate time. Next time when task is given, by the requirement we can predict such things as cyclomatic complexity and by that estimate time for task and etc.

Conclusions

- Analysis part shows that there are a lot of methods to estimate time for tasks/project but the accuracy of these methods are different.
- Each project management methodology has its own methods to estimate time for the project and check correlation between other software metrics.
- Improvement for existing methods can help to estimate time for project/task more accuracy and this will help developers/teams/companies to save a lot of time and money.
- Using already existing methods to estimate software size and connection with described method statistical data can lead to very accurate estimation for the project time.
- Software must be developed to prove this theoretical example.

References

[1] Alain Abran. “Software Metrics & Software Metrology”. 2010. [Reviewed 2016 02 22] Available: <http://profs.etsmtl.ca/aabran/Accueil/ChapersBook/Abran%20-%20Chapter%20007.pdf >

[2] Linda Westfall “12 Steps to Useful Software Metrics”. [Reviewed 2016 02 19] Available: <http://www.westfallteam.com/Papers/12_steps_paper.pdf >

[3] Cees Jan Koomen, “The Stroud number in engineering” [Reviewed 2016-02-22] Available: <http://www.eetimes.com/author.asp?section_id=36&doc_id=1265859>

[4] Alex Castrounis, “Why Software Development Time Estimation Doesn't Work and Alternative Approaches” 2015-08-15. Available: [Reviewed 2016-02-22] Available <http://www.innoarchitech.com/why-software-development-time-estimation-does-not-work-alternative-approaches/>

[5] Chris McMahon, “Estimation approaches in Agile development”. 2011 May. [Reviewed 2016-02-23]. Available: <http://searchsoftwarequality.techtarget.com/tip/Estimation-approaches-in-Agile-development>

[6] Wolfgang Holz, Rahul Premraj, Thomas Zimmermann, Andreas Zeller. “Predicting Software Metrics at Design Time”. [Reviewed 2016-04-12]. Available: <http://thomas-zimmermann.com/publications/files/holz-profes-2008.pdf>

[7] “Overview of COCOMO”. [Reviewed 2016-04-12]. Available: <http://www.softstarsystems.com/overview.htm>