Kauno technologijos universitetas

Informatikos fakultetas

Kompiuterio įvykių laike atkūrimo sistemos kūrimas ir tyrimas

Baigiamasis magistro studijų projektas

Tomas Kašelynas

Projekto autorius

Prof. Vacius Jusas

Vadovas

Kaunas, 2023

Kauno technologijos universitetas

Informatikos fakultetas

Kompiuterio įvykių laike atkūrimo sistemos kūrimas ir tyrimas

Baigiamasis magistro studijų projektas

Programų sistemų inžinerija (6211BX011)

Tomas Kašelynas

Projekto autorius

Prof. Vacius Jusas

Vadovas

Dr. Aušra Gadeikytė

Recenzentė

Kaunas, 2023

Kauno technologijos universitetas

Informatikos fakultetas

Tomas Kašelynas

Kompiuterio įvykių laike atkūrimo sistemos kūrimas ir tyrimas

Akademinio sąžiningumo deklaracija

Patvirtinu, kad:

1. baigiamąjį projektą parengiau savarankiškai ir sąžiningai, nepažeisdama(s) kitų asmenų autoriaus ar kitų teisių, laikydamasi(s) Lietuvos Respublikos autorių teisių ir gretutinių teisių įstatymo nuostatų, Kauno technologijos universiteto (toliau – Universitetas) intelektinės nuosavybės valdymo ir perdavimo nuostatų bei Universiteto akademinės etikos kodekse nustatytų etikos reikalavimų;

2. baigiamajame projekte visi pateikti duomenys ir tyrimų rezultatai yra teisingi ir gauti teisėtai, nei viena šio projekto dalis nėra plagijuota nuo jokių spausdintinių ar elektroninių šaltinių, visos baigiamojo projekto tekste pateiktos citatos ir nuorodos yra nurodytos literatūros sąraše;

3. įstatymų nenumatytų piniginių sumų už baigiamąjį projektą ar jo dalis niekam nesu mokėjęs (-usi);

4. suprantu, kad išaiškėjus nesąžiningumo ar kitų asmenų teisių pažeidimo faktui, man bus taikomos akademinės nuobaudos pagal Universitete galiojančią tvarką ir būsiu pašalinta(s) iš Universiteto, o baigiamasis projektas gali būti pateiktas Akademinės etikos ir procedūrų kontrolieriaus tarnybai nagrinėjant galimą akademinės etikos pažeidimą.

Tomas Kašelynas

Patvirtinta elektroniniu būdu

Kauno technologijos universitetas

Informatikos fakultetas

Baigiamojo magistro projekto užduotis

Projekto tema	Kompiuterio įvykių laike atkūrimo sistemos kūrimas ir tyrimas
Reikalavimai ir sąlygos (tikslinti pavadinimą pagal poreikį)	
Vadovas	prof. Vacius Jusas
	(vadovo pareigos, vardas, pavardė, parašas) (data)

Kašelynas Tomas. Kompiuterio įvykių laike atkūrimo sistemos kūrimas ir tyrimas. Magistro baigiamasis projektas vadovas prof. Vacius Jusas; Kauno technologijos universitetas, Informatikos fakultetas.

Studijų kryptis ir sritis (studijų krypčių grupė): Programų sistemos.

Reikšminiai žodžiai: įvykių laike atkūrimas, abstrakcijos lygiai, skaitmeninių tyrimų palengvinimas.

Kaunas, 2023. 125 p.

Santrauka

Baigiamasis magistro projektas yra skirtas išanalizuoti įvairias duomenų struktūras ir algoritmus norint rasti efektyviausius įvykių importavimo iš failo, darbo išsaugojimo ir įkėlimo bei abstrakcijos lygmenų formavimo būdus. Įvairiuose literatūros šaltiniuose yra teigiama, kad skaitmeninių nusikaltimų skaičius nuolatos auga ir jie darosi vis sudėtingesni. Juose taip pat yra pristatomi įvairūs modeliai ir metodai, skirti palengvinti ir pagreitinti skaitmeninių nusikaltimų tyrimų procesą. Kompiuterio įvykių laike atkūrimo sistema yra paremta keturių abstrakcijos lygmenų formavimo metodu. Ji buvo sukurta su .NET karkasu pagal MVVM architektūrą, kuri leidžia kurti naudotojo sąsają nepriklausomai nuo sistemos logikos. Įvykių importavimo iš failo bei įvykių išsaugojimo funkcionalumas yra tiriamas panaudojant įvairias duomenų struktūras ir skirtingus jų iteravimo būdus. Taip pat yra analizuojama *Factory* projektavimo šablono teikiama nauda darbo įkėlimo metu ir aptariamas *Chain of Responsibility* projektavimo šablono pritaikymas kiekvieno abstrakcijos lygmens formavime. Galiausiai, geriausias įvykių importavimas iš failo yra jau sistemoje realizuotas *foreach* ciklas, iteruojantis per *IEnumerable* tipo kolekciją. Efektyviausias darbo išsaugojimo būdas yra *foreach* ciklas, kuris iteruoja per *Span* tipo objektą, gautą iš *List* objekto. Geriausias būdas įkelti darbą iš failo yra naudoti *Factory* projektavimo šabloną. Efektyviausias kiekvieno abstrakcijos lygmens formavimas naudoja *Chain of Responsibility* projektavimą šabloną.

Kašelynas Tomas. Development and Research of the Computer Events' Timeline Reconstruction System. Master's Final Degree Project supervisor prof. Vacius Jusas; Faculty of Informatics, Kaunas University of Technology.

Study field and area (study field group): Software Engineering.

Keywords: event timeline reconstruction, abstraction levels, digital forensics facilitation.

Kaunas, 2023. 125 pages.

Summary

The main goal of Master's thesis is to analyze various data structures and algorithms to find the most effective ways of importing events from file, saving and loading work, and forming abstraction levels. Different sources state that the amount of digital crime occurances is constantly growing and they are getting more sofisticated. The sources also introduce various models and methods to facilitate and accelerate the processes of digital forensics. Computer events' timeline reconstruction system is based on forming four abstraction levels. It has been created on .NET framework using MVVM architecture, which allows to develop user interface independently to business logic. Various data structures and iteration methods on them are used to research the importation of events from file and saving of work. Moreover, the benefits of using Factory design pattern are analyzed for the loading of work as well as the adaptability of Chain of Responsibility design pattern is discussed for the formation of each abstraction level. Finally, the best way to import events from file is the one already implemented in the system which iterates over *IEnumerable* collection in foreach loop. The most effective way to save work is using foreach loop which iterates over Span object which is formed from List type collection. The best way to load work is to use Factory design pattern. The most effective way to form each level of abstraction is to use Chain of Responsibility design pattern.

Turinys

Lentelių sąrašas

Paveikslų sąrašas

Santrumpų ir terminų sąrašas

Santrumpos:

CSV (angl. Comma-Separated Values) – tai tekstinis failas, naudojantis kablelius reikšmių atskyrimui.

DWT (angl. *Dynamic Time Warping*) – tai algoritmas, naudojamas laiko eilučių analizėje apskaičiuoti panašumui tarp dviejų laiko eilučių. Jis apskaičiuoja mažiausią atstumą tarp dviejų skirtingo ilgio laiko eilučių.

HTML (angl. *Hyper Text Markup Language*) – tai standartinis karkasas, naudojamas atvaizduoti turinį naršyklėje.

MAC (angl. *Modified. Accessed. Created*) – tai laikai, kurių metu failas buvo redaguotas, atidarytas ar sukurtas.

MVVM (angl. *Model-View-Viewmodel*) – tai architektūros šablonas, palengvinantis naudotojo sąsajos kūrimo atskyrimą nuo verslo logikos kūrimo.

NTFS (angl. *New Technology File System*) – tai *Windows* operacinėse sistemose naudojama failų sistema.

PLINQ (angl. *Parallel Language-Integrated Query*) – tai lygiagretus užklausų galimybių C# programavimo kalboje įgyvendinimas.

UI (angl. *User Interface*) – tai vieta, kurioje vyksta žmogaus ir sistemos sąsaja, t. y., naudotojo sąsaja.

WPF (angl. *Windows Presentation Foundation*) – tai nemokamas, atviro kodo grafikos posistemis, skirtas kurti naudotojo sąsajas *Windows* operacinei sistemai skirtoms programoms.

XAML (angl. *Extensible Application Markup Language*) – tai *Microsoft* sukurta deklaratyvi kalba *XML* pagrindu.

XML (angl. *Extensible Markup Language*) – tai paprastas, lengvai išplečiamas, tekstinis formatas struktūrizuotai informacijai atvaizduoti.

Terminai:

Disko atvaizdas (angl. *disk image*) – tai tiksli disko turinio kopija.

Laiko žyma (angl. *timestamp*) – tai simbolių seka arba užkoduota informacija, identifikuojanti, kada įvyko tam tikras įvykis.

Programos atspaudas (angl. *fingerprinting*) – programos veiksmą unikaliai identifikuojantys pokyčiai failų sistemos metaduomenyse.

Skaitmeninė kriminalistika (angl. *digital forensics*) – tai elektroninių duomenų, kurie galėtų būti naudingi tyrimui, išgavimo, saugojimo, analizavimo bei išlaikymo procesas.

Tarpinis serveris (angl. *proxy*) – tai serverio programa, kuri yra tarpininkas tarp kliento, prašančio resurso, ir serverio, suteikiančio to resurso.

Įvadas

Skaitmeninių įrenginių skaičius nuolatos auga, o dėl tobulėjančių technologijų žmonės jais naudojasi vis dažniau. Šie įrenginiai gali saugoti asmeninius ar konfidencialius duomenis, kurie pritraukia skaitmeninius nusikaltėlius. Skaitmeniniais tyrimais siekiama surasti ir paaiškinti, kaip buvo įvykdytas nusikaltimas tame įrenginyje. Dauguma tokių tyrimų yra atliekami rankiniu būdu, tačiau dėl eksponentiškai išaugusių duomenų kiekių tyrėjai nebegali efektyviai atlikti tyrimų. Dėl to stipriai išauga skaitmeniniam tyrimui atlikti reikalingas laikas bei nebaigtų tyrimų skaičius. Reikia automatinių priemonių, padedančių tyrėjams efektyviau išanalizuoti gautus duomenis.

Kompiuterio įvykių laike atkūrimas – tai procesas, kurio metu yra bandoma atkurti saugumo incidentą sukėlusių įvykių seką ir identifikuoti tam įvykti leidusias sąlygas [1]. Įvykių atkūrimas leidžia sukurti tokį procesą, kurio rezultatas būtų įvykių, susijusių su tyrimu, laiko juosta, sudaryta iš pateiktos visų įvykių aibės [2]. Įvykių laike atkūrimas yra viena svarbiausių ir sunkiausių skaitmeninės kriminalistikos užduočių, kuri reikalauja išanalizuoti milžinišką kiekį įvykių [3]. Dauguma skaitmeninio tyrimo aspektų vis dar yra daug rankinio darbo reikalaujančios užduotys, nes didelė dalis sukurtų įrankių yra skirti padėti tyrėjui išgauti skaitmeninius įkalčius, o ne juos analizuoti [4].

Darbo tikslas – pagreitinti ir palengvinti skaitmeninius tyrimus automatizuojant dalį tyrimo veiksmų. Tikslui pasiekti iškeliami šie uždaviniai:

1. išanalizuoti įvykių laike atkūrimo sritį;
2. parengti reikalavimų specifikaciją sistemai;
3. suprojektuoti sistemos architektūrą;
4. sukurti kompiuterio įvykių laike atkūrimo sistemą;
5. ištestuoti sukurtą sistemą;
6. atlikti kompiuterio įvykių laike atkūrimo sistemos tyrimą geriausiems įvykių importavimo iš failo į sistemą, darbo išsaugojimo ir įkėlimo bei abstrakcijos lygmenų formavimo būdams rasti.

Pirmame skyriuje yra aprašomi kompiuterio įvykių laike atkūrimo metodai, egzistuojantys įvykių atkūrimo įrankiai, galimos problemos, kolekcijos elementų iteravimo būdai C# programavimo kalboje bei pasirinkti metodai, įrankiai ir technologijos sistemai įgyvendinti. Antrame skyriuje yra pateikiami magistrantūros studijų metu sukurtos kompiuterio įvykių laike atkūrimo sistemos esminiai techninės projektinės dokumentacijos aspektai. Trečiame skyriuje yra aprašomos sukurtos programų sistemos kokybės tobulinimo galimybės. Ketvirtame skyriuje yra pateikiami įgyvendintų patobulinimų eksperimentiniai rezultatai.

1. Analitinė dalis

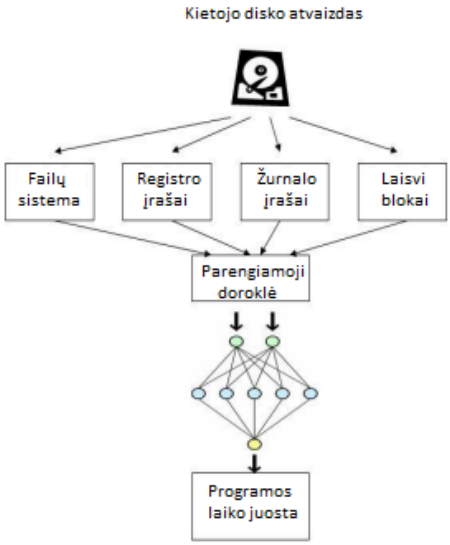
1.1. Kompiuterio įvykių laike atkūrimo metodai

Pirmas įvykių laike atkūrimo metodas yra paremtas aprašymais (angl. *signature*). Jis gali atkurti aukšto lygio naudotojo veiksmus iš žemo lygio pėdsakų aibės, gautos sistemos analizės metu. Dauguma informacijos apsaugos įrankių, pavyzdžiui, antivirusinės programos ar įsibrovimo aptikimo sistemos, naudoja aprašymais paremtą metodą, kad aptiktų tam tikrus veiksmus. Jis yra gana efektyvus, kai yra žinoma, kokių šablonų (angl. *pattern*) reikia ieškoti. Trūkumas yra tas, kad šis metodas negali aptikti nulinės dienos ar specialiai sistemai priderintų atakų, nes nėra sudaryti atitinkami šablonai. Tradiciniai aprašymais paremti metodai stengiasi aptikti tuo metu vykstančias kenkėjiškas veiklas, tačiau buvo pasiūlytas naujas šio metodo panaudojimas. Dėmesys buvo skirtas naudotojo veiksmų aprašymų apibrėžimui. Tie aprašymai yra naudojami po incidento naudotojo veiksmams nustatyti. Be to, toks panaudojimo būdas yra taikomas visos sistemos lygmenyje, o ne susitelkiant į vieną failą [5].

Vėliau aprašymais paremtas metodas buvo panaudotas siekiant užtikrinti greitesnį tyrimo procesą. Jis buvo naudojamas automatiniam naudotojo atliktų veiksmų sistemoje atkūrimui, aproksimuojant aptiktus veiksmus. Šis metodas yra formaliai apibrėžtas kaip objektų, kurių poaibis galėjo būti pakeistas veiksmo atsiradimo metu, aibė. Šiam metodui įgyvendinti buvo išskirtos trys objektų grupės: esminiai (angl. *core*), palaikantys (angl. *supporting*) ir bendri (angl. *shared*). Kiekvienai objektų grupei yra aprašytos laiko žymų (angl. *timestamp*) atnaujinimų pastovumų sąlygos, pagal kurias objektai yra priskiriami į atitinkamas grupes. Šio metodo trūkumas yra toks pat, kaip ir aprašymais paremtų metodų – ne visi įmanomi veiksmai yra žinomi, todėl neaprašyti veiksmai nebus aptikti. Taip pat reikia atkreipti dėmesį, kad naudojant šį metodą gali būti neįmanoma priskirti objekto vienam konkrečiam veiksmui, kai incidentas turi keletą tiesiogiai susijusių veiksmų [6].

Kitas kompiuterio įvykių laike atkūrimo metodas naudoja mašininį mokymąsi. Šis metodas siekia efektyviai apdoroti didelius duomenų kiekius. Neuroniniam tinklui (žr. 1 pav.) yra paduodami pėdsakai, kurie gali būti žurnalo įrašai, registro įrašai, failų sistema ar laisvi atminties blokai, iš kietojo disko. Neuroninio tinklo trūkumas yra tas, kad jį reikia mokyti, o tai užima daug laiko. Be to, jį reikia mokyti kiekvienai programai atskirai. Taip pat reikia nepamiršti, kad sistemoje išlieka tik naujausi pėdsakai, nes programos veikimo metu senesni pėdsakai yra perrašomi [7].

Po metų mašininiu mokymusi paremtas metodas buvo patobulintas. Prie 1 pav. pateikto dizaino papildomai buvo pridėtas rekurentinis neuroninis tinklas. Abu neuroniniai tinklai turi vieną įvesties sluoksnį, du paslėptus sluoksnius bei vieną išvesties sluoksnį. Kiekvienam įvesties parametrui neuroniniai tinklai buvo mokami atskirai, o vėliau sujungiami į vieną atitinkamą tinklą. Šio metodo trūkumas yra toks pat, kaip ir ankstesnio mašininiu mokymusi paremto metodo – neuroninį tinklą reikia mokyti [8].

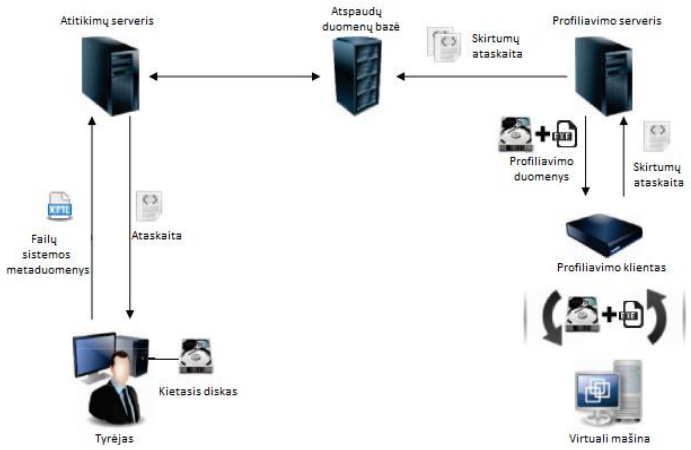


1 pav. Kompiuterio įvykių laike analizės dizainas, naudojant neuroninį tinklą [7]

Kitas metodas remiasi failų ir katalogų sukūrimo, paskutinio rašymo bei paskutinės prieigos laikais. Laiko informacijos forma priklauso nuo failų sistemos bei naudotojo atliekamų veiksmų. Specifiniai pokyčiai joje gali ženkliai pagelbėti analizuojant naudotojo veiksmus sistemoje. Šio metodo autoriai išskyrė dvylika veiksmų su failais bei devynis veiksmus su katalogais NTFS failų sistemoje. Šio metodo trūkumas yra tas, kad reikia turėti daug žinių ir patirties su skirtingomis failų sistemomis. Be to, šio metodo rankinis taikymas gali užimti labai daug laiko [9].

Taip pat buvo pasiūlytas kitas metodas, paremtas NTFS failų sistemos laiko žymų pasikeitimų šablonais. Šio metodo autorius išskyrė dešimt skirtingų laiko žymų pasikeitimo šablonus, naudodamasis septyniomis galimomis operacijomis su failais. Toks metodas gali padėti nustatyti, kokia operacija buvo atlikta su tam tikru failu. Kai kuriuos šablonus galima lengvai identifikuoti, tačiau yra tokių, kuriems reikia ankstesnių laiko žymų. Šio metodo trūkumas yra dviprasmiškumas dėl panašių laiko žymų pasikeitimų šablonų [10].

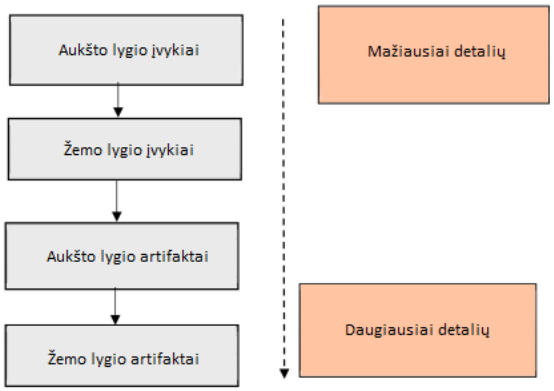
Vienas iš automatinių kompiuterio įvykių atkūrimo metodų yra paremtas programų paliekamais atspaudais. Šis metodas pritaiko programų, kurios padaro kokius nors pokyčius failų sistemos metaduomenyse, atspaudavimo idėją. Tokiam metodui įgyvendinti reikia sudaryti kiekvieno naudotojo veiksmo profilius, kurie apima visus to veiksmo pakeistus failus. 2 pav. pateiktas tokį metodą naudojančios sistemos vaizdas. Tyrėjas pateikia į atitikimų serverį (angl. *Matching Webserver*) failų sistemos metaduomenis ir atgal gauna visas programas, kurios galėjo palikti tokius atspaudus, su jų veiksmais. Atitikimų serveris palygina gautus metaduomenis su atspaudų (angl. *Fingerprinting*) bazėje saugomais atspaudais, kuriuos sugeneruoja profiliavimo klientas (angl. *Profiling Client*). Jis nuolatos vykdo kokį nors veiksmą, sudaro programos profilius ir perduoda į profiliavmo serverį (angl. *Profiling Server*), kuris surenka visus vieno veiksmo rezultatus ir nusiunčia į atspaudų duomenų bazę. Joje yra išsaugomi tik unikalūs profiliai. Šis metodas turi keletą trūkumų. Visų pirma, tą patį veiksmą reikia kartoti daug kartų, jei norima gauti atspaudus be triukšmo, kurį sukelia keli vienu metu sistemoje veikiantys procesai. Kitas trūkumas yra tas, kad reikia nuolatos generuoti programos veiksmų atspaudus, nes kitaip sistema neatpažins naujų programų veiksmų. Be to, sistema negali dirbti su unikaliai sugeneruotomis direktorijomis (pavyzdžiui, sistemos naudotojo vardas), nes skirsis direktorijų pavadinimai [4].



2 pav. Programų atspaudavimu paremto kompiuterio įvykių atkūrimo sistemos vaizdas [4]

Kitas su laiku susijęs metodas yra paremtas failų sistemos modifikacijų laikiniais šablonais. Jis siekia atkurti aukšto lygio įvykius naudodamas failų sistemos metaduomenyse paliktus programų pėdsakus. Tam buvo pasiūlytas įvykių atkūrimo karkasas, kuris nustato, kokia programa buvo paleista tiriamoje sistemoje. Karkasas yra sudarytas iš dviejų fazių. Mokymo fazėje yra sukonstruojami įvairių programų aprašymai (angl. *signature*). Aptikimo fazėje yra sukonstruojami tiriamos sistemos kietojo disko failų sistemos modifikacijų šablonai. Norint nustatyti, ar tam tikra programa buvo paleista sistemoje, yra apskaičiuojamas DWT atstumas tarp tos programos aprašymo ir tiriamos sistemos kietojo disko sukonstruoto šablono. Automatinį sprendimą priima specialus variklis. Šio metodo trūkumas yra tas, kad kiekvienai programai reikia nustatyti kiek galima tikslesnį jos aprašymą [11].

Galiausiai, yra abstrakcijos lygmenų formavimu paremtas metodas. Jis sudaro keturis abstrakcijos lygmenis: aukšto lygio įvykių, žemo lygio įvykių, aukšto lygio artifaktų ir žemo lygio artifaktų. 3 pav. pateikti ryšiai tarp abstrakcijos sluoksnių. Aukšto lygio įvykių (angl. *Events: High Level*) sluoksnis turi mažiausiai informacijos, jame yra pateikiama informacija apie naudotojo sukurtus failus ir naršymą internete. Žemo lygio įvykių (angl. *Events: Low Level*) sluoksnis yra papildomas informacija apie modifikavimo veiksmus. Aukšto lygio artifaktų (angl. *Artefact Location: High Level*) sluoksnyje yra analizuojama naudotojo veikla internete bei vykdomųjų failų paleidimas.Žemo lygio artifaktų (angl. *Artefact Location: Low Level*) sluoksnis turi daugiausiai detalių, nes naudoja visus galimus šaltinius. Šio metodo trūkumas yra tas, kad nėra vieningo sutarimo, kas turėtų būti kiekviename lygmenyje.



3 pav. Abstrakcijos lygmenų ryšiai [3]

1.2. Egzistuojantys įrankiai

1.2.1. *CAT Record*

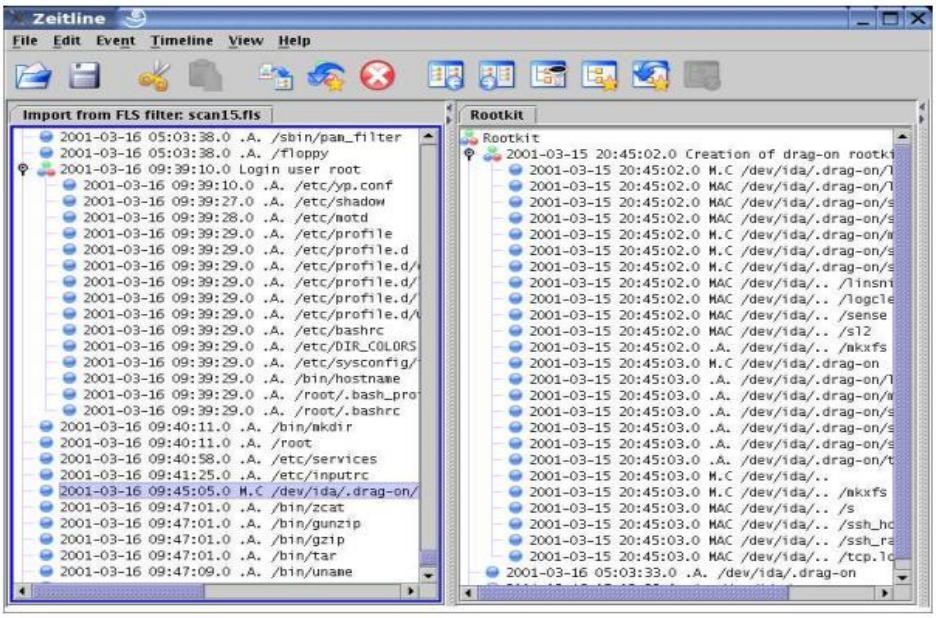
CAT Record yra įrankis, skirtas registruoti naudotojo atliekamus veiksmus kompiuteryje realiu laiku. Prototipas turėjo šešis modulius. Pirmas modulis yra *Windows* įvykių stebėtojas, kuris registruoja operacinės sistemos įvykius. Antras modulis yra aktyvaus lango detektorius, kuris registruoja aktyvius programos langus. Kitas modulis yra šrifto, laiko, energijos bei skiriamosios gebos detektorius, kuris registruoja šrifto, laiko, energijos ir skiriamosios gebos pokyčius sistemoje. Taip pat yra naršyklės modulis, kuris stebi veiklą *Windows Explorer* ar *Internet Explorer* programose. Kitas modulis yra išorinių įrenginių detektorius, kuris registruoja išorinių įrenginių prijungiamą ar atjungimą. Galiausiai, failų sistemos stebėtojas registruoja veiksmus su failų sistema. Šių modulių duomenys yra registruojami duomenų bazėje [12].

Privalumai:
1. įrankis veikia realiu laiku;
2. nereikia analizuoti kietojo disko;
3. sistema atspari laiko žymų klastojimui.

Trūkumai:
1. prototipo negalima atsisiųsti;
2. sistema turi būti įdiegta į kompiuterį prieš incidentą;
3. reikia turėti duomenų bazę;
4. reikia užtikrinti duomenų bazės saugumą;
5. sistema sulėtina kompiuterio darbą;
6. reikia papildomos sistemos, kuri galėtų atvaizduoti užregistruotus duomenis.

1.2.2. *Zeitline*

Zeitline (žr. 4 pav.) yra grafinis laiko juostos redaktorius, leidžiantis tyrėjui atkurti įvykius, kurie buvo gauti iš skirtingų šaltinių, laike. Įrankio autoriai išskyrė tris pagrindines įrankio savybes: įvykių valdymą, teisinės ekspertizės vientisumo išlaikymą ir užklausas. Tyrėjas gali kurti hierarchines įvykių struktūras, sudarydamas sudėtingus įvykius iš atominių ar kitų sudėtingų įvykių. Taip pat galima sukurti tuščią įvykių seką ir ją užpildyti reikiamais įvykiais. Prie įvykių valdymo taip pat galima priskirti kelių laiko linijų peržiūrėjimą vienu metu. Programos pateikti rezultatai gali būti naudojami teisminiuose procesuose, todėl įrankis naudoja paslėptą įvykių seką, kurioje yra laikomi ištrinti ar iškirpti įvykiai. Užklausos padeda greitai rasti norimus įvykius. Užklausos šiame įrankyje atitinka įvykių filtravimą pagal raktažodį, įvykio šaltinį, laiką ar tipą [13].



4 pav. *Zeitline* naudotojo sąsaja [13]

Privalumai:
1. numatytos įrankio išplėtimo galimybės;
2. gali pats surinkti įvykius.

Trūkumai:
1. lėtas esant didelių įvykių skaičiui;
2. reikalauja rankinio darbo;
3. nepalaikomas, paskutinis atnaujinimas buvo prieš 10 metų[1];
4. ribotas įvykių surinkimas, nes įvykius renka tik iš MAC laikų, sistemos žurnalų ir ugniasienės šaltinių;
5. negalima eksportuoti rezultatų.

1.2.3. *PyDFT*

PyDFT yra programa, skirta automatiškai atkurti aukšto lygio įvykius iš surinktų žemo lygio įvykių. Įrankis tiek žemo lygio, tiek aukšto lygio įvykius saugo *SQLite* duomenų bazėje. Aukšto lygio įvykiai yra atkuriami pagal apibrėžtus žemo lygio įvykių šablonus. Aukšto lygio įvykiai taip pat gali būti eksportuojami XML ar HTML formatu [14].

Privalumai:
1. galima eksportuoti rezultatus dviem formatais;
2. galima išplėsti sistemą.

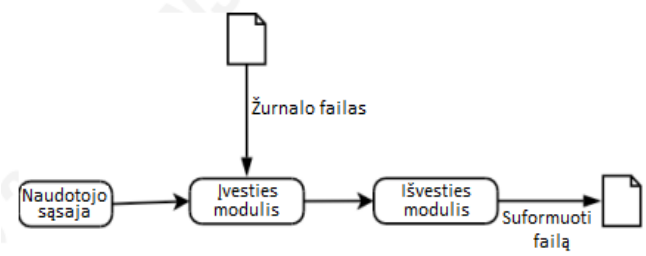
Trūkumai:
1. naudojamas tik vienas branduolys, todėl pridedant daugiau analizatorių sulėtėja programos darbas;
2. reikia sukurti modulius, kurie išgautų žemo lygio įvykius;
3. reikia sukurti analizatorius, kurie atpažintų aukšto lygio įvykius;

[1] https://sourceforge.net/projects/zeitline/

4. reikia duomenų bazės;
5. reikia naudoti kitus įrankius rezultatams atvaizduoti;
6. nėra visiškai išbandytas;
7. prototipo negalima atsisiųsti.

1.2.4. *Log2timeline*

Log2timeline yra karkasas, skirtas lengvai ir automatiškai apdoroti skirtingus žurnalo failus ir artifaktus sukuriant super-laiko juostą, padedančią tyrėjams atlikti pilną sistemos analizę. Pasikliaudamas tik failų sistemos laiko linija tyrėjas gali nesuprasti konteksto, kuris yra būtinas norint gauti visą ir tikslų įvykusio įvykio apibūdinimą. Tam išvengti yra įtraukiama informacija, rasta įvairiuose artifaktuose ir žurnalų failuose. Jie gali būti tiek tiriamoje sistemoje, tiek kitame įrenginyje, pavyzdžiui, ugniasienėje ar tarpiniame serveryje. Įrankį sudaro trys moduliai (žr. 5 pav.): naudotojo sąsaja, įvesties modulis ir išvesties modulis. Naudotojo pusė apdoroja jai pateiktus parametrus ir iškviečia atitinkamą įvesties modulį. Įvesties modulis apdoroja rastus artifaktus ir perduoda rastus įvykius išvesties moduliui. Išvesties modulis išsaugo gautus rezultatus nurodyto tipo faile. Naudotojo pusę galima pasiekti per komandinę eilutę, grafinę naudotojo sąsają ar automatinį skenavimą. Įrankis turi 26-is įvesties modulius, kurie apdoroja atitinkamus artifaktus, bei devynis išvesties modulius, kurie gali išsaugoti rezultatus devyniuose skirtinguose failų formatuose [15].



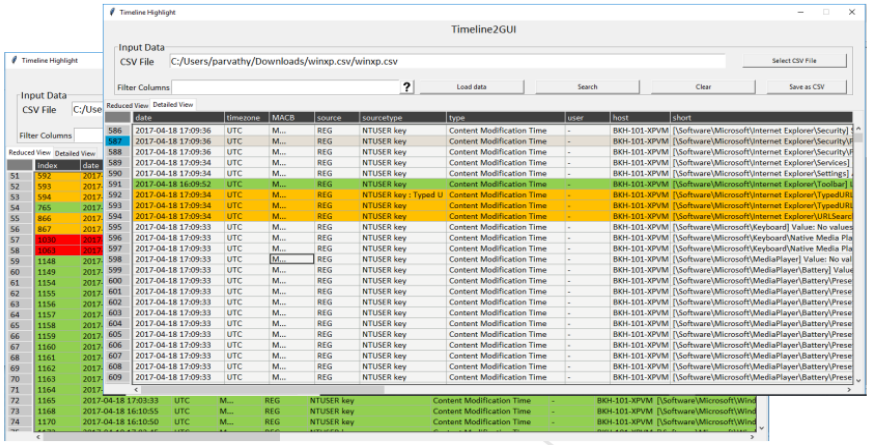
5 pav. *Log2timeline* įrankio veikimo struktūra [15]

Privalumai:
1. galima išplėsti įvesties modulius;
2. galima išplėsti išvesties modulius;
3. aktyviai palaikomas (dabar turi *plaso* pavadinimą);
4. atvirojo kodo;
5. nemokamas;
6. gali išgauti įvykius iš daugybės skirtingų artifaktų;
7. visi įvykiai vienoje vietoje;
8. pakankama dokumentacija (*plaso* versijai).

Trūkumai:
1. labai daug įvykių yra nenaudingi;
2. reikia papildomų įrankių analizei;
3. įvykių surinkimas užtrunka nemažai laiko;
4. reikia turėti sistemos disko, kuriame yra artifaktai, atvaizdą.

1.2.5. *Timeline2GUI*

Timeline2GUI (žr. 6 pav.) yra programinė įranga, skirta lengvai analizuoti *log2timeline* įrankio pateiktus CSV formato failus. Programa yra sukurta su *Python* programavimo kalba. Įrankis turi du pagrindinius langus: sumažinto vaizdo, kur yra pateikiami svarbiausi įvykiai (žr. 6 pav. foną) ir detalaus vaizdo, kur yra rodomi visi įvykiai (žr. 6 pav. priekį). Įrankis gali paryškinti įvykius pagal jų svarbumą tokiomis spalvomis: žalia, geltona, mėlyna, raudona, rausvai raudona ir pilka. Žalia spalva reiškia, kad failas galėjo būti atidarytas ar sukurtas. Geltona spalva parodo, kad tas įvykis yra susijęs su veikla internete. Mėlyni įvykiai parodo išorinių įrenginių sąveiką su sistema. Raudona spalva paryškinami programų paleidimo įvykiai. Rausvai raudona spalva parodo failų ištrynimą. Pilka spalva paryškinami žurnalo įrašai, pavyzdžiui, ugniasienės įrašai [16].



6 pav. *Timeline2GUI* naudotojo sąsaja [16]

Privalumai:
1. naudotojo sąsaja paremta plačiai naudojama *Excel* programa;
2. atvirojo kodo;
3. nemokama;
4. automatiškai paryškina svarbias vietas.

Trūkumai:
1. suderinamas tik su *log2timeline* įrankiu;
2. veikia tik su CSV formato failais;
3. aktyviai nepalaikoma;
4. ribotas funkcionalumas;
5. tyrėjas turi būti susipažinęs su tiriamos sistemos operacine sistema.

1.2.6. Kiti įrankiai

Taip pat verta paminėti keletą profesionalių įrankių, kurie yra ypač funkcionalūs, tačiau dauguma jų yra komerciniai produktai, kurių licencijos kainuoja didelius pinigus. Dėl didelio funkcionalumo, jų naudotojo sąsajos taip pat yra itin sudėtingos ir tam reikia specializuotų darbuotojų mokymo ir naudotojo vadovų.

EnCase[2] yra laikomas globaliu skaitmeninių tyrimų technologijos standartu tyrėjams, norintiems atlikti efektyvų ir teisiškai teisingą duomenų surinkimą ir tyrimą. Jis gali išgauti įkalčius iš daugiau nei 25-ių skirtingų tipų įrenginių. Metinė prenumeratos kaina siekia daugiau nei 3500 dolerių. Yra nemokama dalis *EnCase Imager*, kuri gali surinkti įkalčius. Duomenų apdorojimas užtrunka daug laiko.

The Sleuth Kit[3] yra komandinės eilutės įrankių kolekcija, leidžiančių ištirti disko atvaizdus. Įrankis gali būti įtrauktas į didesnius skaitmeninės analizės įrankius. Kartais gali būti konfigūravimo problemų. Šis įrankis taip pat turi išplėtimą *Autopsy*[4], kuris suteikia grafinę naudotojo sąsają.

Forensic ToolKit[5] yra profesionalus didelio funkcionalumo įrankis. Jis gali sukurti disko atvaizdus, atlikti jų analizę, iššifruoti failus, nulaužti slaptažodžius, apdoroti registro failus ir t.t. Licencija kainuoja daugiau nei 2000 dolerių per metus, turi paprastą naudotojo sąsają. Taip pat yra nemokama *FTK Imager* dalis, skirta sukurti disko atvaizdo kopijas.

X-Ways[6] įrankis išplečia *WinHex*, kuris yra skirtas duomenų atgavimui ir informacinių technologijų apsaugai. *X-Ways* papildomai prideda daugybę funkcijų, skirtų palengvinti analizę. Įrankis yra skirtas *Windows* operacinei sistemai, tačiau licencija kainuoja daugiau nei 1000-į eurų metams. Taip pat jis turi gana sudėtingą naudotojo sąsają.

RedLine[7] yra nemokamas įrankis atminties ir failų analizei. Jis gali surinkti informaciją apie veikiančius procesus, registrų duomenis, užduotis, servisus, tinklo informaciją ir naršymo istoriją.

Bulk Extractor[8] yra atvirojo kodo įrankis, kuris gali skenuoti disko vaizdus, failus ar direktorijas, ignoruodamas failų sistemos struktūrą, todėl jis yra greitesnis nei panašūs įrankiai, tačiau įrankis nebėra aktyviai palaikomas.

1.3. Galimos problemos

1.3.1. Didelio duomenų kiekio problema

Kompiuterio įvykių paėmimas iš skirtingų šaltinių gali pateikti keletą milijonų žemo lygio įvykių, pavyzdžiui, failo modifikavimas ar registro įrašo atnaujinimas [7] [11] [14]. Milijonų įvykių nagrinėjimas ir aukšto lygio informacijos išgavimas iš jų yra itin varginantis ir daug laiko užimantis darbas [11]. Didelį kiekį duomenų gali padėti išanalizuoti automatinės priemonės.

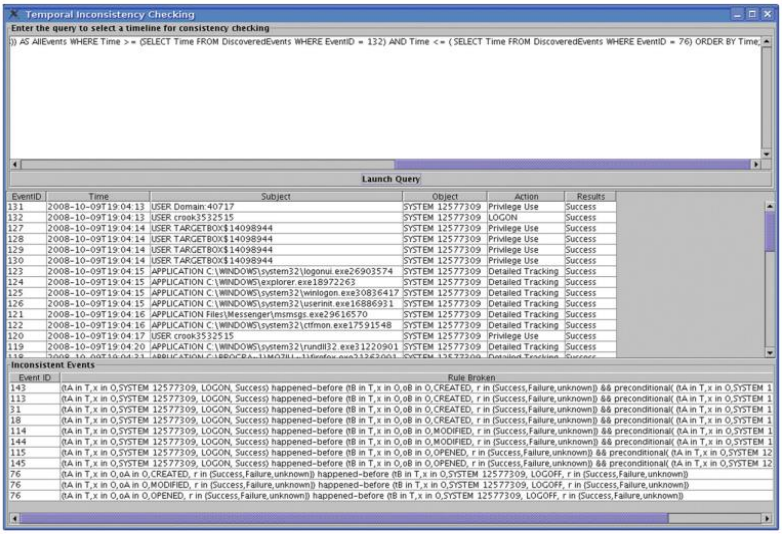
Kuo didesnis duomenų kiekis, tuo ilgiau užtrunka automatinių priemonių darbas. Kuriama sistema bus tiriama norint rasti efektyviausius būdus atlikti veiksmus su dideliu duomenų kiekiu. Įvykių įkėlimo iš sugeneruoto super-laiko linijos failo į sistemą bei darbo išsaugojimo ir įkėlimo algoritmai bus tiriami išbandant įvairius ciklo iteravimo būdus. Abstrakcijos lygmenų formavimo algoritmui bus bandoma pritaikyti *Chain of Responsibility* projektavimo šabloną.

[2] https://e-forensic.ca/products/encase-forensic-suite/
[3] https://www.sleuthkit.org/sleuthkit/
[4] https://www.sleuthkit.org/autopsy/
[5] https://www.exterro.com/forensic-toolkit
[6] https://www.x-ways.net/forensics/index-m.html
[7] https://fireeye.market/apps/211364
[8] https://github.com/simsong/bulk_extractor

1.3.2. Įvykių klastojimo problema

Skaitmeninį nusikaltimą vykdantis asmuo gali bandyti paslėpti savo pėdsakus. Tą padaryti galima pakeičiant laiko žymas NTFS failų sistemoje [17].

Vienas iš šios problemos sprendimo būdų yra specialių įrankių naudojimas. *CAT Record* (žr. 7 pav.) yra prototipinė sistema, skirta aptikti nepastovumus kompiuterio veiklų laiko juostoje. Įrankis gali aptikti įvykius, kurie įvyko ne iš eilės ar yra dingę iš laiko juostos. Tokiems neatitikimams rasti įrankis naudoja specialiai apibrėžtas taisykles. Vienas iš didesnių įrankio trūkumų yra tas, kad jis negali automatiškai aptikti naudotojo sesijų, todėl reikia nurodyti laiko ribas, tarp kurių įrankis turi ieškoti nepastovumų [18]. Be to, naudotojo sąsaja nėra patogi. Pats įrankis taip pat nebėra palaikomas, nes paskutinė rasta įrankio versija buvo išleista prieš 8-erius metus[9].



7 pav. *CAT Detect* naudotojo sąsaja [18]

Kitas būdas yra analizuoti žurnalo įrašų failą. Iš jo galima atkurti atliktas operacijas su failais. Remiantis pastebėtais skirtumais tarp laiko žymų po normalių operacijų su failais ir programų, leidžiančių pakeisti laiko žymas, naudojimo, buvo sudarytos septynios taisyklės laiko žymų padirbimo aptikimui. Tos taisyklės yra skirtos *.txt, .docx* bei *.pdf* plėtinio failams [17]. Šio metodo taikymui reikia turėti žinių apie NTFS failų sistemą, žurnalo failą, jo struktūrą bei atitinkamų operacijų kodus.

Kuriamas projektas nesirūpins šios problemos sprendimu, nes visas dėmesys yra skiriamas kompiuterio įvykių laike atkūrimui.

1.3.3. Gautų rezultatų teisinės galios problema

Norint teisme pateikti kompiuterio analizės metu gautus įkalčius, reikia aiškiai parodyti, kad įkalčių gavimo proceso metu buvo laikomasi priimtų procedūrų ir praktikų. Dirbtinio intelekto ir mašininio mokymosi metodai reikalauja ypač svaraus pagrindimo jų tikslumui ir patikimumui [8]. Taip pat reikia pateikti išsamius ir suprantamus argumentus, kodėl ir kaip buvo naudojami tam tikri moksliniai metodai įkalčių radimui.

[9] https://sourceforge.net/projects/catdetect/

Kuriamas projektas leidžia patikrinti nurodyto failo bei jame esančių įvykių vientisumą, kas tyrėjui turėtų leisti įsitikinti, kad niekas kažkaip nepakeitė juose esančių duomenų. Taip pat testavimo metu buvo atlikta statinė analizė, kurios metu buvo patikrintas visos sistemos kodas norint išvengti netikėtų duomenų pakeitimų.

1.4. Elementų iteravimo būdai

For sakinys (žr. 8 pav.) vykdo nurodytą kodo bloką tol, kol yra tenkinama pateikta loginė sąlyga. Jį sudaro trys dalys: iniciatorius, sąlyga bei iteratorius. Iniciatoriaus sekcija yra vykdoma tik vieną kartą prieš pradedant ciklą. Čia dažniausiai yra sukuriami lokalūs kintamieji, kurių negalima pasiekti už *for* sakinio ribų. Sąlygos sekcija nustato, ar turėtų būti vykdoma kita ciklo iteracija. Ji gali nieko neturėti (begalinis ciklas), arba turėti kokią nors loginę sąlygą. Iteratoriaus sekcija nurodo, kas bus daroma po kiekvieno kodo bloko įvykdymo [19]. Šis iteravimo būdas yra naudojamas, kai yra žinomas iteracijų skaičius.

```
//    Iniciatorius; Sąlyga; Iteratorius
for (int i = 0; i < allElements.Count; i++)
{
    // Kodo blokas
}
```

8 pav. *For* sakinio struktūra

Foreach sakinys (žr. 9 pav.) vykdo nurodytą kodo bloką kiekvienam objekto, kurio klasė įgyvendina *IEnumerable* ar *IEnumerable<T>* sąsają, elementui [19].

```
foreach (var element in allElements)
{
    // Kodo blokas
}
```

9 pav. *Foreach* sakinio struktūra

Await foreach sakinys (žr. 10 pav.) yra naudojamas, kai yra bandoma iteruoti per asinchroniškai pasiekiamą elementų kolekciją, kurios klasė įgyvendina *IAsyncEnumerable<T>* sąsają [19].

```
await foreach (var element in GetSequenceAsync())
{
    // Kodo blokas
}
```

10 pav. *Await foreach* sakinio struktūa

Taip pat verta paminėti, kad iki .NET 7 versijos *foreach* sakinys, buvo lėtesnis nei *for* sakinys. .NET 7 versijoje taip pat sumažėjo *for* ciklo našumas, tačiau tai buvo klaida, kuri turėtų būti ištaisyta .NET 8 versijoje [20]. Prieš .NET 7 versiją *foreach* sakinys buvo konvertuojamas į *while* tipo ciklą, kuris naudojo enumeratorių, o *for* ciklas buvo konvertuojamas į *while* ciklą be enumeratoriaus. Po .NET 8 versijos *for* bei *foreach* sakinių našumas turėtų būti gana panašus, bet *for* turėtų likti šiek tiek greitesnis.

Do...while sakinys (žr. 11 pav.) vykdo nurodytą kodo bloką tol, kol yra tenkinama pateikta loginė sąlyga. Kodo blokas yra įvykdomas būtinai bent vieną kartą, nes loginė sąlyga yra tikrinama po jo įvykdymo [19].

```
int n = 0;
do
{
    // Kodo blokas
} while (n < 5);
```

11 pav. *Do...while* sakinio struktūra

While sakinys (žr. 12 pav.) taip pat vykdo nurodytą kodo bloką tol, kol yra tenkinama pateikta loginė sąlyga. Kodo blokas gali būti nė karto neįvykdytas, nes loginė sąlyga yra tikrinama prieš jo vykdymą [19]. Šis iteravimo būdas yra naudojamas, kai nėra žinomas iteracijų skaičius.

```
int n = 0;
while (n < 5)
{
    // Kodo blokas
}
```

12 pav. *While* sakinio struktūra

Parallel.For sakinys (žr. 13 pav.) leidžia kiekvienam kolekcijos elementui lygiagrečiai įvykdyti nurodytą kodo bloką. Šio tipo sakiniui reikia nurodyti pradžią ir pabaigą bei kodo bloką. Papildomai galima nurodyti lygiagretumo parametrus, pavyzdžiui, lygiagretumo laipsnį.

```
Parallel.For(fromInclusive, toExclusive, parallelOptions, index =>
{
    // Kodo blokas
});
```

13 pav. *Parallel.For* sakinio struktūra

Parallel.ForEach sakinys (žr. 14 pav.) yra panašus į *Parallel.For* sakinį. Jis leidžia lygiagrečiai atlikti veiksmus su kiekvienu kolekcijos elementu. Šiame metode yra prarandama prieiga prie elemento indekso.

```
Parallel.ForEach(allElements, parallelOptions, element =>
{
    // Kodo blokas
});
```

14 pav. *Parallel.ForEach* sakinio struktūra

Bandant išnaudoti lygiagretumo galimybes reikėtų nepersistengti, nes bandant maksimaliai išnaudoti procesoriaus branduolius galima pasiekti tokią ribą, kur gijų valdymas pradės mažinti lygiagretumo suteikiamą naudą. Todėl reikia gerai įvertinti, kokia kodo dalis galėtų geriau išnaudoti lygiagretumą.

Taip pat nuo .NET 6 versijos yra asinchroninis *Parallel.ForEach* variantas – *Parallel.ForEachAsync* (žr. 15 pav.). Jis savo kodo bloko viduje leidžia kviesti asinchronines funkcijas.

```
await Parallel.ForEachAsync(allElements, async(element, cancellationToken) =>
{
    // Kodo blokas
});
```

15 pav. *Parallel.ForEachAsync* sakinio struktūra

Galiausiai, galima naudoti PLINQ užklausas (žr. 16 pav.). *AsParallel* plėtinys leidžia užklausą įvykdyti lygiagrečiai. Ši metodą galima naudoti, kai rezultatų elementų eilės tvarka nėra svarbi. Užklausa padalina elementus į atskiras užduotis, kurios yra asinchroniškai įvykdomos ant keleto gijų. Užduočių baigimo tvarka priklauso ne tik nuo atliekamo darbo kiekio, bet ir operacinės sistemos atliekamo gijų planavimo.

```
var parallelQuery = allElements
    .AsParallel()
    .Where(element => element % 2 == 0);
```

16 pav. PLINQ struktūra

1.5. Pasirinkti metodai, įrankiai ir technologijos

.NET karkasas buvo pasirinktas, nes sistema yra skirta *Windows* operacinei sistemai. Taip pat šis karkasas leidžia nesunkiai panaudoti lygiagretumą, dėl ko turėtų būti galima išnaudoti visus kompiuterio procesoriaus branduolius ir greičiau atlikti analizę. Be to, C# programavimo kalba turi neblogą bibliotekų palaikymą, kurios gali sutaupyti nemažai laiko, ir aktyvią bendruomenę, kuri padėtų iškilus problemoms su .NET ar C#. Galiausiai, C# programavimo kalba yra kompiliuojama, dėl ko nenukenčia greitaveika kaip interpretuojamose programavimo kalbose.

WPF projekto tipas buvo pasirinktas, nes jis yra .NET karkaso dalis, todėl jis neturėtų turėti kritinių klaidų. Jis taip pat leidžia gana lengvai kurti naudotojo sąsają programoms, skirtoms *Windows* operacinei sistemai. Galiausiai, naudotojo sąsają galima nesunkiai kurti su XAML.

MVVM architektūra buvo pasirinkta, nes ji buvo sukurta *Microsoft* ir yra puikiai suderinama su WPF projekto tipu. Jis leidžia pašalinti visą kodą iš naudotojo sąsajos, todėl ją galima plėsti nepriklausomai nuo kitų sistemos dalių.

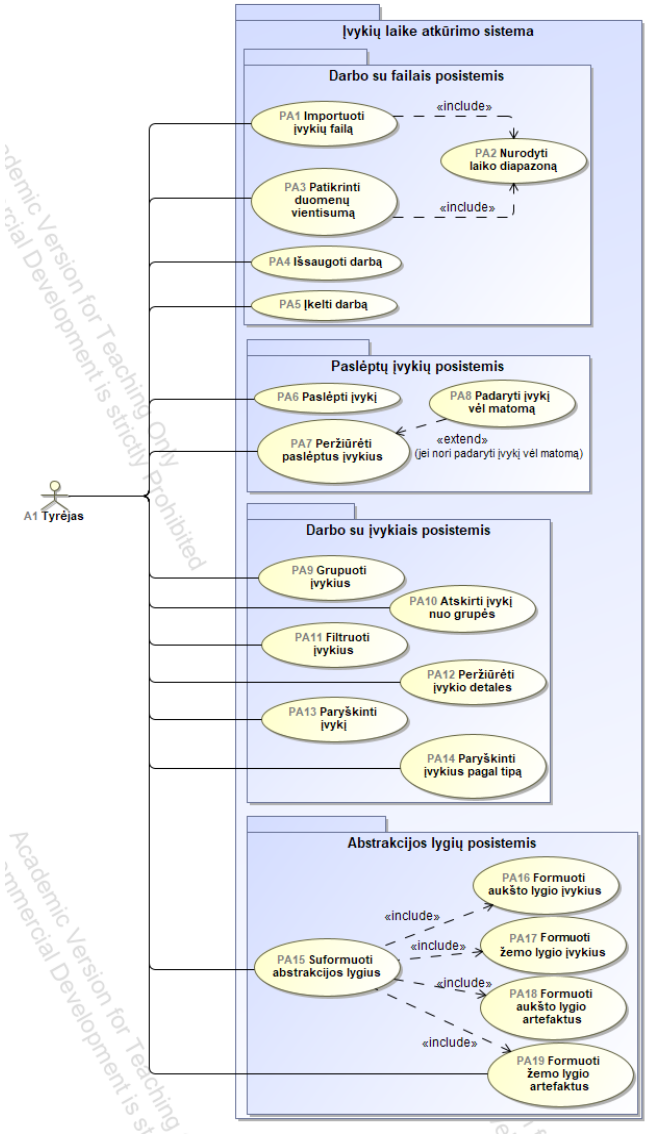
Log2timeline įrankis buvo pasirinktas dėl kelių priežasčių. Visų pirma, tai yra užsakovo reikalavimas naudoti šį įrankį kompiuterio artifaktų surinkimui. Be to, šis įrankis yra atvirojo kodo, todėl jis yra nuolatos tobulinamas, plečiamas ir tvarkomas. Jis taip pat turi neblogą dokumentaciją, kuri leidžia greitai susipažinti su šiuo įrankiu ir jo galimybėmis.

Abstrakcijų lygiais paremtas metodas buvo pasirinktas, nes jis leidžia automatizuoti dalį įvykių analizės. Be to, projekto užsakovas yra vienas straipsnio, kuriame buvo pristatytas šis metodas, autorių, todėl galima gauti šio metodo įgyvendinimo metodologiją. Taip pat straipsnyje yra užsimenama, kad toks metodas yra įgyvendintas Java programavimo kalba, tačiau nėra pateikiamas įrankio pavadinimas.

2. Projektinė dalis

2.1. Sistemos sudėtis

Kompiuterio įvykių laike atkūrimo sistemos panaudojimo atvejai pateikiami 17 pav. Sistema yra padalinta į keturis posistemius. Darbo su failais posistemis apima *log2timeline* įrankio sugeneruoto failo importavimą į sistemą, failo ir įvykių vientisumo patikrinimą bei esamos sistemos būsenos išsaugojimą ir įkėlimą. Paslėptų įvykių posistemis įtraukia įvykio paslėpimą, kad jis nebūtų matomas įvykių hierarchijoje, paslėptų įvykių peržiūrėjimą ir įvykio paslėpimo anuliavimą. Darbo su įvykiais posistemis apima įvykių grupavimą ir atskyrimą nuo grupės, filtravimą, įvykio detalios informacijos peržiūrėjimą bei įvykio paryškinimo funkcijas. Galiausiai, abstrakcijos lygių posistemyje esantis funkcionalumas yra skirtas keturių abstrakcijos lygmenų formavimui.



17 pav. Kompiuterio įvykių laike atkūrimo sistemos panaudojimo atvejų diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

Svarbiausias sistemos funkcionalumas yra įvykių importavimas į sistemą bei keturių abstrakcijos lygmenų formavimas. Šių panaudojimo atvejų veiklos diagramos pateiktos 20-21 pav., o sekų diagramos 22-23 pav.

2.2. Svarbiausi nefunkciniai reikalavimai

2.2.1. Reikalavimai panaudojamumui

Pirmas svarbus nefunkcinis reikalavimas yra tas, kad sistema turi būti paprasta naudotis (žr. 1 lentelę). Kitu atveju, sistema nepalengvintų tyrėjų darbo, o tai neleistų pasiekti užsibrėžto sistemos tikslo.

1 lentelė. Nefunkcinis reikalavimas NR-4 (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

Reikalavimas #:	**NR-4**	**Reikalavimo tipas:**	**11**	**PA #:**	**visi**
Aprašymas:	Sistema turi būti paprasta naudotis.				
Pagrindimas:	Sistema turi palengvinti tyrėjų darbą be daugybės apmokymų.				
Šaltinis:	Užsakovas, sistemos kūrėjas.				
Tenkinimo kriterijus:	Naujas sistemos naudotojas neužtruks daugiau nei 10 sekundžių norimam sistemos funkcionalumui surasti.				
Užsakovo pasitenkinimas:	**1**		**Užsakovo nepasitenkinimas:**		**4**
Priklausomybės:	FR-3, FR-4, FR-7, FR-9, FR-10, FR-11, FR-19, FR-20, FR-21, FR-22		**Konfliktai:**	Nėra.	
Papildoma medžiaga:	Nėra.				
Istorija:	Užregistruotas 2022.03.08.				

2.2.2. Reikalavimai vykdymo charakteristikoms

Antras svarbus nefunkcinis reikalavimas yra skirtas sistemos našumui. Sistema turi importuoti įvykius iš *log2timeline* įrankio suformuoto failo, išsaugoti ir įkelti darbą bei formuoti abstrakcijos lygius greičiau nei per 15 sekundžių dirbant su 4 milijonais elementų (žr. 2 lentelę). Šis reikalavimas taip pat yra tiesiogiai susijęs su darbo tikslu.

2 lentelė. Nefunkcinis reikalavimas NR-6 (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

Reikalavimas #:	**NR-6**	**Reikalavimo tipas:**	**12**	**PA #:**	**1, 3, 4, 8, 9, 11**
Aprašymas:	Sistemos operacijos su visais įvykiais negali trukti ilgiau nei 15 sekundžių esant 4 milijonams įvykių.				
Pagrindimas:	Tyrėjas gali nenorėti ilgai laukti, kol sistema atliks veiksmus su visais įvykiais.				
Šaltinis:	Tyrėjas.				
Tenkinimo kriterijus:	Sistema nuo naudotojo inicijuoto veiksmo iki jo rezultato užtrunka ne ilgiau kaip 15 sekundžių su 4 milijonais įvykių. Testavimo metu bus atliekama po 10 operacijų kiekvienam funkcionalumui ir skaičiuojamas vidutinis laikas. Testavimas bus atliekamas intel i5-9400F 2.9 GHz procesoriaus ir 16GB RAM aplinkoje.				
Užsakovo pasitenkinimas:	**4**		**Užsakovo nepasitenkinimas:**		**5**
Priklausomybės:	FR-1, FR-3, FR-6, FR-9, FR-15, FR-17, FR-18, FR-23		**Konfliktai:**	Nėra.	

Papildoma medžiaga:	Nėra.
Istorija:	Užregistruotas 2022.03.08.

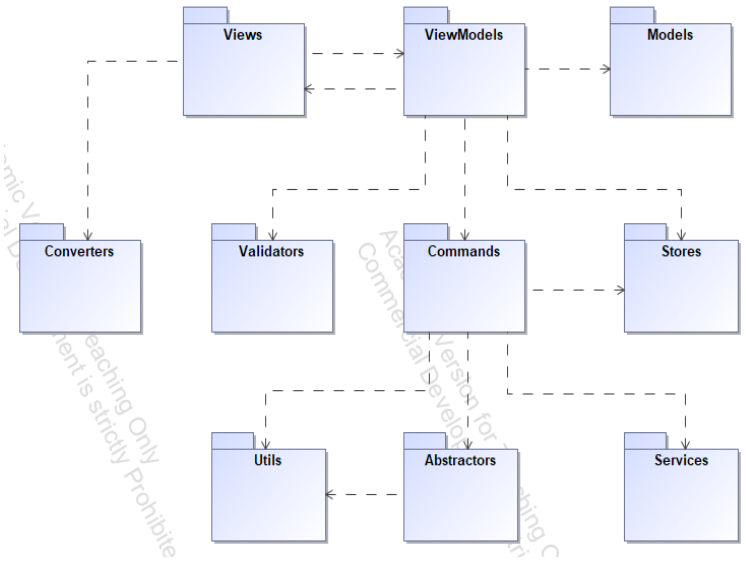
Trečias svarbus reikalavimas yra numatyti būdą išplėsti sistemą leidžiant palaikyti kitus *log2timeline* įrankio grąžinamus failus (žr. 3 lentelę). Šio reikalavimo įgyvendinimas taip pat leistų nepriklausyti nuo vieno įrankio.

3 lentelė. Nefunkcinis reikalavimas NR-9 (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

Reikalavimas #:	**NR-9**	**Reikalavimo tipas:**	**12**	**PA #:**	**1, 8**
Aprašymas:	Turi būti numatytas būdas sistemą išplėsti leidžiant palaikyti kitus *log2timeline* įrankio generuojamų failų formatus.				
Pagrindimas:	*Log2timeline* įrankis gali pakeisti duomenų struktūrą arba gali reikėti pereiti prie kito įrankio, kuris turi kitokį duomenų formatą.				
Šaltinis:	Užsakovas.				
Tenkinimo kriterijus:	Sistemoje galima lengvai pridėti naują modulį, skirtą skaityti kitokios struktūros failus.				
Užsakovo pasitenkinimas:	**4**		**Užsakovo nepasitenkinimas:**		**1**
Priklausomybės:	FR-1, FR-2, FR-15		**Konfliktai:**	Nėra.	
Papildoma medžiaga:	Nėra.				
Istorija:	Užregistruotas 2022.03.08.				

2.3. Sistemos statinis vaizdas

Sistema sukurta pagal MVVM architektūrą (žr. 18 pav.), nes ji puikiai tinka WPF projekto tipui. Joje yra trys pagrindinės dalys: *Models, Views* ir *ViewModels*. *Views* paketas yra skirtas naudotojo sąsajos langams. *ViewModels* paketas yra skirtas atvaizdavimo logikai aprašyti. Šie paketai yra susieti abipusiu ryšiu, kad *View* galėtų atnaujinti naudotojo sąsają pasikeitus *ViewModel* reikšmėms, o *ViewModel* gautų reikšmes iš *View*. *Models* paketas yra skirtas dalykinės srities duomenų modeliams, kuriuos naudoja *ViewModel*. Kiti paketai yra nebūtini, tačiau jie padeda išlaikyti kodą švaresnį.



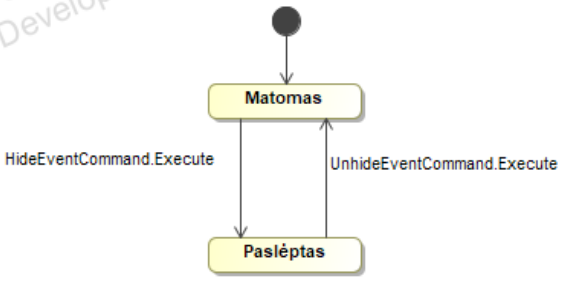
18 pav. Kompiuterio įvykių laike atkūrimo sistemos paketų diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

Converters paketas yra skirtas specifinėms komandoms, kurios gali perduoti daugiau nei vieno elemento objektus komandoms iš to *View*. Paketas *Validators* yra skirtas patikrinti naudotojo įvestų laukų reikšmes. *Commands* paketas yra skirtas mygtukų funkcionalumo įgyvendinimui. Paketas *Stores* yra skirtas saugoti nustatymus ir visus įvykius bei išlaikyti jų vientisumą. *Utils* paketas turi pagalbines funkcijas, kad jos neperkrautų pagrindinių klasių ir jas būtų galima panaudoti keliose skirtingose klasėse. Paketas *Abstractors* saugo keturių abstrakcijos lygių formavimo logiką. Galiausiai, *Services* paketas yra skirtas bendravimui su išoriniais duomenų šaltiniais, pavyzdžiui, failais.

2.4. Sistemos dinaminis vaizdas

2.4.1. Būsenos diagramos

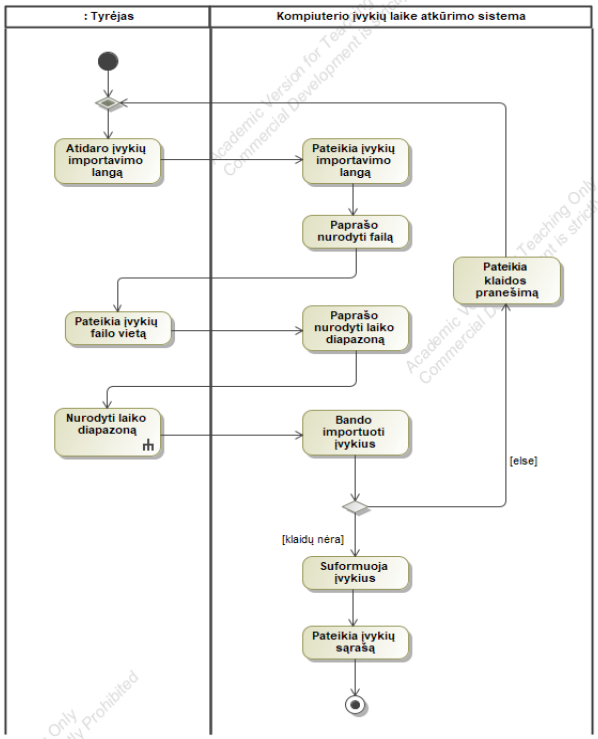
Įvykio modelį atitinkantis *ViewModel* gali turėti dvi būsenas (žr. 19 pav.): matomas ir paslėptas. Visi iš *log2timeline* įrankio suformuoto failo importuoti įvykiai yra matomi. Po to tyrėjas gali paslėpti pasirinktus įvykius, kurių būsena pasikeičia į paslėptą. Toks veiksmas gali būti anuliuotas ir įvykio būsena grįžta į matomą.



19 pav. Įvykio būsenų diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

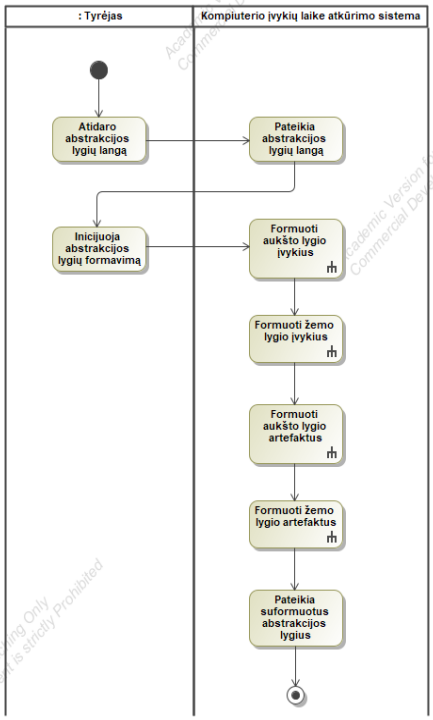
2.4.2. Veiklos diagramos

Įvykių importavimo iš failo į sistemą veiklos diagrama pateikta 20 pav. Norėdamas importuoti įvykius iš failo, tyrėjas turi atidaryti įvykių importavimo langą, pasirinkti *log2timeline* įrankio grąžintą failą ir nurodyti norimą laiko intervalą. Jeigu nėra klaidų, sistema pateikia importuotų įvykių sąrašą. Kitu atveju, sistema parodo klaidos pranešimą.



20 pav. PA1 „Importuoti įvykių failą“ veiklos diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

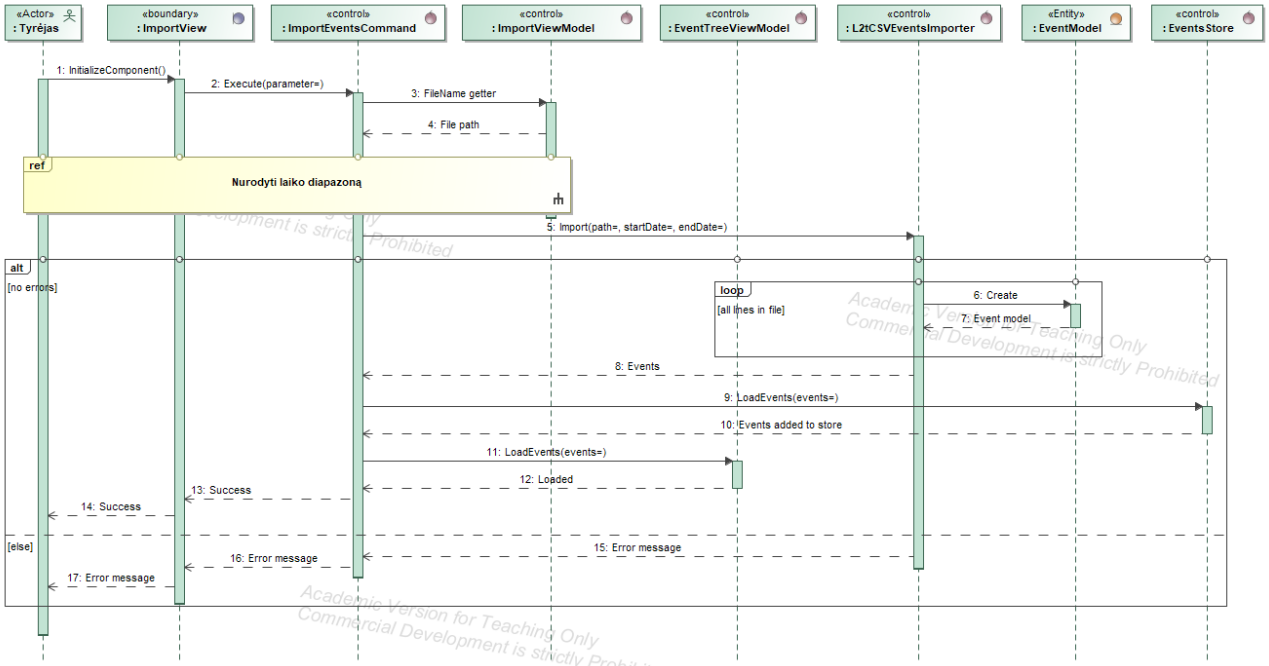
Abstrakcijos lygmenų formavimas (žr. 21 pav.) taip pat prasideda lango atidarymu. Jame tyrėjas gali inicijuoti lygių formavimą, po kurio sistema automatiškai suformuoja aukšto lygio įvykius ir artifaktus bei žemo lygio įvykius ir artifaktus.



21 pav. PA15 „Suformuoti abstrakcijos lygius“ veiklos diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

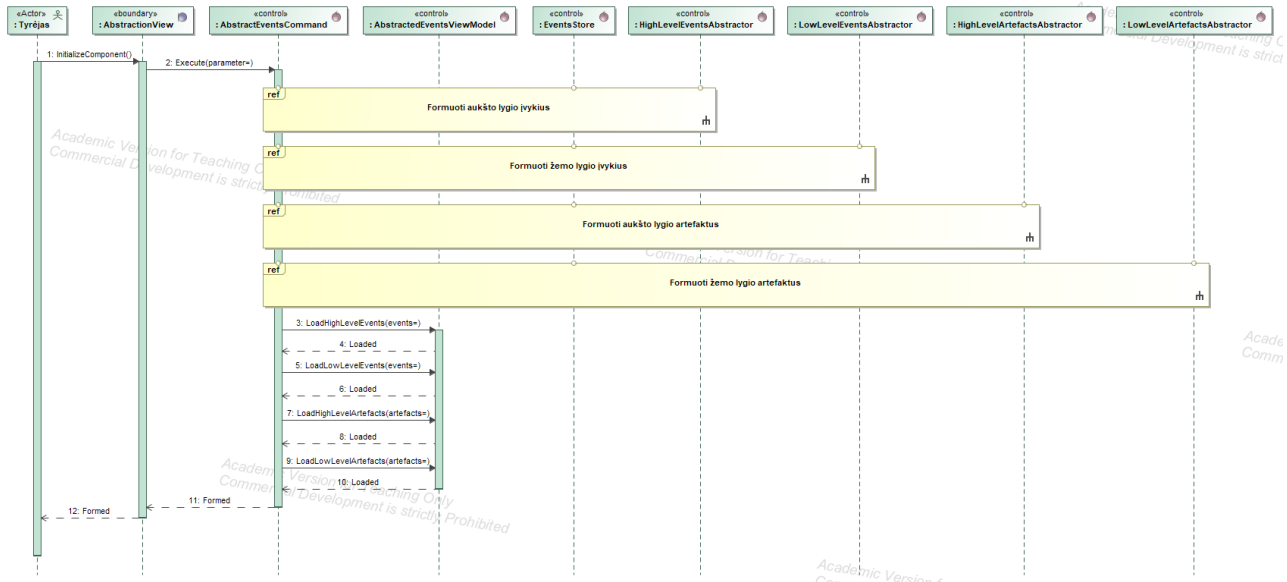
2.4.3. Sekų diagramos

Sistemos komponentų bendravimas importuojant įvykius iš failo į sistemą pateiktas 22 pav. Tyrėjas atidaro *ImportView* langą, kuriame gali pradėti įvykių importavimą. Šiam veiksmui inicijuoti, jis turi pateikti failo kelią bei nurodyti laiko diapazoną. Po to sistema pradeda skaityti failą, sukurdama atitinkamus *EventModel* objektus, kuriuos vėliau išsisaugo ir į *EventsStore*. Tada nuskaitytus įvykius sistema atiduoda *EventTreeViewModel* klasei, kuri atvaizduoja visus įvykius pagrindiniame lange. Jei failo skaitymo metu įvyksta klaida, sistema naudotojui pateikia klaidos pranešimą.



22 pav. PA1 „Importuoti įvykių failą“ sekų diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

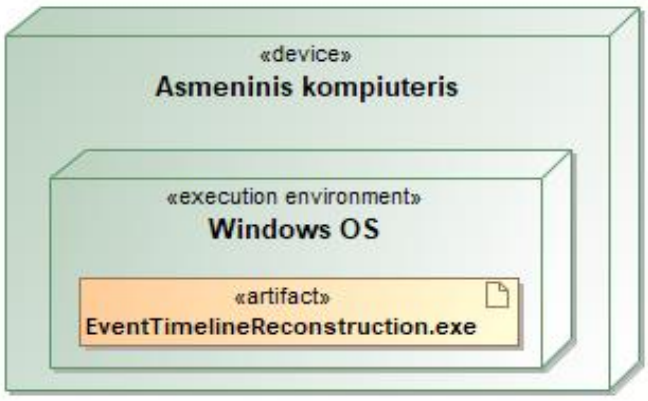
Abstrakcijos lygmenų formavimo veiksmų seka pateikta 23 pav. Visų pirma, tyrėjas atidaro *AbstractionView* langą, kuriame gali inicijuoti keturių abstrakcijos lygių formavimą. Sistema naudoja atitinkamus objektus (*HighLevelEventsAbstractor*, *LowLevelEventsAbstractor*, *HighLevelArtefactsAbstractor ir LowLevelArtefactsAbstractor*) kiekvieno lygmens formavimui. Įvykiai yra imami iš *EventsStore* objekto. Kiekvieno abstrakcijos lygmens įvykiai yra perduodami *AbstractedEventsViewModel* objektui, kuris atvaizduoja kiekvieną abstrakcijos lygmenį *AbstractionView* lange.



23 pav. PA15 „Suformuoti abstrakcijos lygius“ sekų diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

2.5. Išdėstymo vaizdas

Sistema yra išleidžiama kaip nepriklausoma programa, todėl jos diegimui nereikia jokių papildomų veiksmų (žr. 24 pav.). Naudotojui reikia turėti kompiuterį su *Windows* operacine sistema. Kompiuterio įvykių laike atkūrimo sistemos veikimui reikėtų *.NET Framework*, tačiau jis yra įtraukiamas sistemos išleidimo metu, kad naudotojui nereikėtų pačiam diegti papildomos programinės įrangos.



24 pav. Kompiuterio įvykių laike atkūrimo sistemos išdėstymo diagrama (T.Kašelynas, Įvykių laike atkūrimo sistemos kūrimas ir tyrimas Techninė dokumentacija, 2023)

3. Tyrimo dalis

Kompiuterio įvykių laike atkūrimo sistema turi dirbti su labai dideliais įvykių kiekiais. Didelis duomenų kiekis gali paveikti ne tik sistemos greitaveiką, bet ir naudojamos operatyviosios atminties kiekį. Dėl šios priežasties bus tiriami keturi svarbiausi sistemos funkcionalumai: įvykių importavimas iš failo, darbo išsaugojimas ir įkėlimas bei abstrakcijos lygmenų formavimas.

3.1. Įvykių importavimo iš failo tobulinimo galimybės

Įvykių importavimą iš failo galima padalinti į dvi dalis: faile esančių duomenų nuskaitymą ir įvykių modelių sukonstravimą iš nuskaitytų duomenų. Pirmos dalies (duomenų nuskaitymo iš failo) galima netirti, nes yra du galimi būdai tą padaryti su C# programavimo kalba. Pirmas būdas yra naudoti *File.ReadAllLines* metodą. Jis atidaro failą, nuskaito visas jame esančias eilutes į *string* masyvą ir uždaro failą [21]. Kitas būdas yra naudoti *File.ReadLines* metodą. Jis vietoj *string* masyvo grąžina *IEnumerable<string>* objektą, kas leidžia atlikti LINQ užklausas bei skaityti failą po vieną eilutę. Taip galima sutaupyti daug operatyviosios atminties ir laiko, nes nereikia užkrauti viso failo [22]. Remiantis *Microsoft* dokumentacija, failas bus skaitomas naudojant *File.ReadLines* metodą, kuris leidžia sumažinti naudojamos atminties kiekį, ir nebus tiriami kiti failo skaitymo būdai.

3.1.1. Paprastas *for* ciklas

Paprasto *for* ciklo naudojimas įvykių iš failo importavimui pateiktas 25 pav. Čia norint panaudoti *for* ciklą, reikia turėti ne iteratorių (*IEnumerable<string>*), o masyvą arba sąrašą su visais elementais. Dėl šios priežasties iteratorius turi būti konvertuojamas į sąrašą su *ToList* metodu, o tai jau užima *O(n)* laiko ir atminties. Ciklo iteravimo metu kiekviena nuskaityta failo eilutė yra konvertuojama į atitinkamą įvykio modelį. Ciklo laiko sudėtingumas taip pat yra *O(n)*. Sąlygos sakinys nesumažina ciklo laiko sudėtingumo, nes blogiausiu atveju reikės nuskaityti visas eilutes. Taip pat galima nutraukti ciklo darbą, kai sukonstruoto įvykio data viršija *toDate* parametrą, tačiau tai nepakeistų ciklo sudėtingumo, bet padidintų ciklomatinį sudėtingumą. Taip pat *events* sąrašas blogiausiu atveju turės visus sukonstruotus įvykius, todėl jo atminties sudėtingumas yra *O(n)*. Galiausiai, bendras šio patobulinimo laiko sudėtingumas yra *O(2n),* o užimamos atminties sudėtingumas taip pat yra *O(2n)*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    List<string> rows = File.ReadLines(path).Skip(1).ToList();
    List<EventModel> events = new(rows.Count);

    for (int i = 0; i < rows.Count; i++)
    {
        string[] columns = rows[i].Split(',');

        if (columns.Length != _colCount)
        {
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, i + 2);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
    }

    return events;
}
```

25 pav. Įvykių importavimas iš failo naudojant *for* ciklą

3.1.2. Nesaugus *for* ciklas naudojant *Span* iš *List*

Nesaugaus *for* ciklo naudojimas su *Span* iš *List* yra pateiktas 26 pav. Čia yra naudojama nesaugi *CollectionsMarshal* klasė, kuri leidžia pasiekti pagrindinius duomenis, atvaizduojančius C# programavimo kalboje naudojamas kolekcijas. Ši klasė yra nesaugi, nes ji leidžia tiesiogiai dirbti su kolekcijos duomenimis, todėl negalima pridėti ar išmesti elementų naudojant *Span* [23]. Kaip ir paprasto *for* ciklo atveju, laiko ir atminties sudėtingumai lieka tokie patys – *O(2n)*, tačiau *Span* naudojimo efektyvumas gali būti akivaizdus dirbant su labai dideliais duomenų kiekiais.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    List<string> rows = File.ReadLines(path).Skip(1).ToList();
    List<EventModel> events = new(rows.Count);

    Span<string> rowsAsSpan = CollectionsMarshal.AsSpan(rows);
    for (int i = 0; i < rowsAsSpan.Length; i++)
    {
        string[] columns = rowsAsSpan[i].Split(',');

        if (columns.Length != _colCount)
        {
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, i + 2);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
    }

    return events;
}
```

26 pav. Įvykių importavimas iš failo naudojant *for* ciklą ir konvertuojant *List* į *Span*

3.1.3. Nesaugus *for* ciklas naudojant *Span* iš masyvo

Nesaugaus *for* ciklo naudojimas su *Span* iš masyvo yra pateiktas 27 pav. Šis patobulinimas yra praktiškai identiškas 3.1.2 skyrelyje aprašytam, tačiau vietoj *List* klasės yra naudojamas masyvas. Masyvas galėjo būti gautas naudojant *File.ReadAllLines* metodą, tačiau jis grąžintų ir antraštinę eilutę, kuri yra nereikalinga. Jos tikrinimas taip pat pridėtų papildomą sąlygos sakinį, o tai padidintų ciklomatinį sudėtingumą. Dėl šios priežasties yra naudojamas *File.ReadLines* metodas, kuris leidžia praleisti nurodytą eilučių skaičių su *Skip* užklausa. Šio patobulinimo laiko ir atminties sudėtingumai yra *O(2n)*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    string[] rows = File.ReadLines(path).Skip(1).ToArray();
    List<EventModel> events = new(rows.Length);
    Span<string> rowsSpan = rows.AsSpan();

    for (int i = 0; i < rowsSpan.Length; i++)
    {
        string[] columns = rowsSpan[i].Split(',');

        if (columns.Length != _colCount)
        {
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, i + 2);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
    }

    return events;
}
```

27 pav. Įvykių importavimas iš failo naudojant *for* ciklą ir konvertuojant masyvą į *Span*

3.1.4. Nesaugus *for* ciklas naudojant *MemoryMarshal*

Nesaugaus *for* ciklo naudojimas su *MemoryMarshal* yra parodytas 28 pav. Čia taip pat yra naudojama nesaugi klasė (*MemoryMarshal*). Ši klasė yra nesaugi, nes ji leidžia tiesiogiai dirbti su atmintimi [24]. Reikiamas masyvo elementas yra paimamas naudojant *Unsafe.Add* metodą, kurie prie masyvo *rows* pradžios atmintyje prideda nurodytą reikšmę (šiuo atveju, tai yra norimo elemento indeksas). Kaip ir kitais *for* ciklo atvejais, laiko ir atminties sudėtingumai yra *O(2n)*, tačiau *MemoryMarshal* naudojimas gali būti efektyvesnis ne tik už paprastus masyvus ar sąrašus, bet ir *Span*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    string[] rows = File.ReadLines(path).Skip(1).ToArray();
    List<EventModel> events = new(rows.Length);
    ref var searchSpace = ref MemoryMarshal.GetArrayDataReference(rows);

    for (int i = 0; i < rows.Length; i++)
    {
        string row = Unsafe.Add(ref searchSpace, i);
        string[] columns = row.Split(',');

        if (columns.Length != _colCount)
        {
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, i + 2);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
    }

    return events;
}
```

28 pav. Įvykių importavimas iš failo naudojant *for* ciklą su *MemoryMarshal*

3.1.5. Lygiagretus *for* ciklas

Lygiagretaus *for* ciklo naudojimas įvykių iš failo importavimui pateiktas 29 pav. Jam realizuoti yra panaudojamas *Parallel.For* metodas, kuris lygiagrečiai apdoroja visas iš failo nuskaitytas eilutes. Kaip ir visuose kituose *for* ciklą naudojančiuose metoduose, reikia turėti visą eilučių masyvą. Užimamos atminties sudėtingumas taip pat yra *O(2n)*, o laiko sudėtingumas yra $O(\frac{n}{k})$, kur k yra lygiagretumo laipsnis. Kuo daugiau gijų gali būti panaudojama, tuo jos greičiau apdoros visas eilutes. Lygiagretumo laipsnis yra konstanta, todėl ciklo laiko sudėtingumą galima suprastinti iki *O(n)*. Galutinis šio būdo laiko sudėtingumas yra *O(2n)*, nes reikia įvertinti ir masyvo sukūrimą su *ToArray* metodu.

Viena iš lygiagretumo problemų yra gijų valdymas. Gali būti atvejis, kad dvi ar daugiau gijos bando pridėti elementą į sąrašą tuo pačiu metu. Taip susidaro lenktynių sąlygos, dėl ko į sąrašą yra pridedamas tik vienas elementas. Norint suvaldyti lenktynių sąlygas, reikia naudoti užraktą (angl *lock*), kuris neleidžia vykdyti to paties kodo bloko dviem ar daugiau gijoms vienu metu. Taip yra užtikrinama, kad visi elementai yra pridedami į sąrašą. Užraktų naudojimas sulėtina greitaveiką.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    List<string> rows = File.ReadLines(path).Skip(1).ToList();
    List<EventModel> events = new(rows.Count);
    object lockObj = new();

    Parallel.For(0, rows.Count, index =>
    {
        string[] columns = rows[index].Split(',');

        if (columns.Length != _colCount)
        {
            return;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, index + 2);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                lock (lockObj)
                {
                    events.Add(eventModel);
                }
            }
        }
        catch (FormatException)
        {
            return;
        }
        catch (IndexOutOfRangeException)
        {
            return;
        }
    });

    return events;
}
```

29 pav. Įvykių importavimas iš failo naudojant lygiagretų *for* ciklą

3.1.6. Paprastas *foreach* ciklas

Paprasto *foreach* ciklo naudojimas yra parodytas 30 pav. Čia nebereikia turėti masyvo, nes *foreach* gali naudoti iteratorių, gautą iš *IEnumerable<string>* objekto. Dėl šios priežasties yra sutaupoma *O(n)* laiko ir atminties. Čia taip pat reikia naudoti papildomą kintamąjį *lineNumber*, kuris užima *O(1)* atminties, kad būtų galima sekti ir išsaugoti eilutės numerį. Pačio ciklo laiko sudėtingumas yra *O(n)*, nes yra iteruojama per visas eilutes. Atminties sudėtingumas čia taip pat yra mažesnis, nes nereikia atmintyje laikyti visų nuskaitytų eilučių. Bendras šio patobulinimo laiko sudėtingumas yra *O(n)*, o užimamos atminties sudėtingumas – *O(n)*.

Šis būdas šiuo metu yra įgyvendintas kompiuterio įvykių laike atkūrimo sistemoje.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    IEnumerable<string> rows = File.ReadLines(path).Skip(1);
    List<EventModel> events = new();

    int lineNumber = 2;
    foreach (string line in rows)
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            lineNumber++;
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, lineNumber);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
        finally
        {
            lineNumber++;
        }
    }

    return events;
}
```

30 pav. Įvykių importavimas iš failo naudojant *foreach* ciklą

3.1.7. Paprastas *foreach* ciklas naudojant LINQ

Foreach užklausų panaudojimas pateiktas 31 pav. Užklausa gali būti panaudojama ant konkretaus sąrašo, todėl reikia nuskaityti visas failo eilutes į atmintį. Tai užima *O(n)* laiko ir atminties. Pats *foreach* blokas yra toks pat kaip ir 3.1.6 skyrelyje. Galutiniai šio būdo laiko ir atminties sudėtingumai yra *O(2n)*, kas yra du kartus blogiau nei paprasto *foreach* metodo.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    List<string> rows = File.ReadLines(path).Skip(1).ToList();
    List<EventModel> events = new(rows.Count);

    int lineNumber = 2;
    rows.ForEach(line =>
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            lineNumber++;
            return;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, lineNumber);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            return;
        }
        catch (IndexOutOfRangeException)
        {
            return;
        }
        finally
        {
            lineNumber++;
        }
    });

    return events;
}
```

31 pav. Įvykių importavimas iš failo naudojant *foreach* užklausą

3.1.8. Nesaugus *foreach* ciklas naudojant *Span* iš *List*

Nesaugaus *foreach* ciklo naudojimas su *Span* iš *List* yra pateiktas 32 pav. Čia, kaip ir 3.1.2 skyrelyje, yra naudojama nesaugi *CollectionsMarshal* klasė, kuri leidžia tiesiogiai dirbti su kolekcijos duomenimis. Tam reikia turėti visas eilutes, o tai užima *O(n)* laiko ir atminties. Pats *foreach* taip pat užima *O(n)* laiko, o *events* objektas blogiausiu atveju turės visus įvykius, todėl jis gali užimti *O(n)* atminties. Galiausiai, bendri laiko ir atminties sudėtingumai yra *O(2n)*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    List<string> rows = File.ReadLines(path).Skip(1).ToList();
    List<EventModel> events = new(rows.Count);

    int lineNumber = 2;
    foreach (string line in CollectionsMarshal.AsSpan(rows))
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            lineNumber++;
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, lineNumber);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
        finally
        {
            lineNumber++;
        }
    }

    return events;
}
```

32 pav. Įvykių importavimas iš failo naudojant *foreach* ciklą ir konvertuojant *List* į *Span*

3.1.9. Nesaugus *foreach* ciklas naudojant *Span* iš masyvo

Nesaugaus *foreach* ciklo naudojimas su *Span* iš masyvo yra parodytas 33 pav. Šis patobulinimas yra panašus į aprašytą 3.1.8 skyrelyje, tačiau yra naudojamas masyvas. Kaip ir 3.1.3 skyrelyje, čia nenaudojamas *File.ReadAllLines* metodas, nes nenoriu didinti ciklomatinio sudėtingumo. Šio patobulinimo laiko ir atminties sudėtingumai taip pat yra *O(2n)*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    string[] rows = File.ReadLines(path).Skip(1).ToArray();
    List<EventModel> events = new(rows.Length);

    int lineNumber = 2;
    foreach (string line in rows.AsSpan())
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            lineNumber++;
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, lineNumber);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
        finally
        {
            lineNumber++;
        }
    }

    return events;
}
```

33 pav. Įvykių importavimas iš failo naudojant *foreach* ciklą ir konvertuojant masyvą į *Span*

3.1.10. Lygiagretus *foreach* ciklas

Lygiagretaus *foreach* ciklo panaudojimas įvykių iš failo importavimui yra pateiktas 34 pav. Jam realizuoti yra panaudojamas *Parallel.ForEach* metodas, kuriam galima paduoti *IEnumerable<string>*. Dėl to galima sutaupyti *O(n)* laiko ir atminties neužkraunant viso failo į atmintį. Laiko sudėtingumą, kaip ir lygiagretaus *for* ciklo naudojimo atveju, galima suprastinti iki *O(n)*. Galutinis šio patobulinimo laiko ir atminties sudėtingumas yra *O(n)*. Papildomai laiko gali užimti dinaminio sąrašo *List* naudojimas. Jam užsipildžius, jis turi būti praplečiamas. Norint to išvengti, galima nurodyti jo dydį, tačiau *IEnumerable<string>* objektas negali grąžinti duomenų kiekio nepereidamas per visus elementus, o tai užtruktų *O(n)* laiko. Papildomai galima ištirti, kaip šio ir kitų būdų, naudojančių *IEnumerable* objektą, greitaveika priklauso nuo elementų kiekio paėmimo ir perdavimo dinaminiam sąrašui. Laiko sudėtingumas inicializuojant sąrašo dydį išaugtų iki *O(2n)*.

Šiame patobulinime taip pat reikia nepamiršti gijų valdymų, kad būtų galima išvengti lenktynių sąlygų.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    IEnumerable<string> rows = File.ReadLines(path).Skip(1);
    List<EventModel> events = new();
    object lockObj = new();

    Parallel.ForEach(rows, (line, _, lineNumber) =>
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            return;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, (int)lineNumber + 2);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                lock (lockObj)
                {
                    events.Add(eventModel);
                }
            }
        }
        catch (FormatException)
        {
            return;
        }
        catch (IndexOutOfRangeException)
        {
            return;
        }
    });

    return events;
}
```

34 pav. Įvykių importavimas iš failo naudojant lygiagretų *foreach* ciklą

3.1.11. Lygiagretus asinchroninis *foreach* ciklas

Lygiagretaus asinchroninio *foreach* ciklo panaudojimas pateiktas 35 pav. Šis būdas tinka, kai reikia lygiagrečiai apdoroti asinchronines užduotis. *Parellel.ForEachAsync* metodui galima paduoti *IEnumerable<stirng>* objektą, o tai sutaupo *O(n)* laiko bei atminties. Ciklo viduje reikia dirbtinai sukurti asinchroninę užduotį naudojant *Task.Run* metodą, kad būtų galima įgyvendinti šį patobulinimą. Bendri šio būdo laiko ir atminties sudėtingumai yra *O(n)*.

Vienas iš šio patobulinimo trūkumų yra tas, kad negalima žinoti, kuri eilutė yra apdorojama. Tai pažeistų užsakovo abstrakcijos lygmenų formavimo metodologiją, kuri prašo išsaugoti įvykio eilutės faile numerį. Kitas trūkumas yra tas, kad jo negalima lengvai pritaikyti dabartinei sistemai, nes *Parallel.ForeachAsync* bloko viduje turi būti naudojamas asinchroninis metodo kvietimas. Dirbtinis asinchroninių užduočių kūrimas apsunkina sistemos palaikomumą ir suprantamumą.

```
public async Task<List<EventModel>> Import(string path, DateTime fromDate, DateTime toDate)
{
    IEnumerable<string> rows = File.ReadLines(path).Skip(1);
    List<EventModel> events = new();
    object lockObj = new();

    await Parallel.ForEachAsync(rows, async (line, token) =>
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            return;
        }

        try
        {
            await Task.Run(() =>
            {
                EventModel eventModel = ConvertRowToModel(columns, 0);
                DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                         eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

                if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
                {
                    lock (lockObj)
                    {
                        events.Add(eventModel);
                    }
                }
            }, token);
        }
        catch (FormatException)
        {
            return;
        }
        catch (IndexOutOfRangeException)
        {
            return;
        }
    });

    return events;
}
```

35 pav. Įvykių importavimas iš failo naudojant lygiagretų asinchroninį *foreach* ciklą

3.1.12. Paprastas *while* ciklas

Paprasto *while* ciklo naudojimas įvykių importavimui iš failo pateiktas 36 pav. Norint sutaupyti *O(n)* laiko ir atminties, yra naudojamas *IEnumerator<string>* objektas, gautas iš *IEnumerable<string>*. Jis leidžia su *while* ciklu pereiti per visus elementus. Metodas *MoveNext* grąžina loginį operatorių (angl. *bool*), kuris parodo, ar kolekcijoje dar yra elementų. Taip pat jis pakeičia *Current* rodyklę taip, kad ji rodytų į sekantį elementą kolekcijoje. Galutiniai laiko ir atminties sudėtingumai šiam patobulinimui yra *O(n)*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    IEnumerable<string> rows = File.ReadLines(path).Skip(1);
    IEnumerator<string> enumerator = rows.GetEnumerator();
    List<EventModel> events = new();
    int lineNumber = 2;

    while (enumerator.MoveNext())
    {
        string[] columns = enumerator.Current.Split(',');

        if (columns.Length != _colCount)
        {
            lineNumber++;
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, lineNumber);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
        finally
        {
            lineNumber++;
        }
    }

    return events;
}
```

36 pav. Įvykių importavimas iš failo naudojant *while* ciklą

3.1.13. Nesaugus *while* ciklas naudojant *MemoryMarshal*

Nesaugaus *while* ciklo panaudojimas su *MemoryMarshal* yra parodytas 37 pav. Kaip ir 3.1.4 skyrelyje, čia yra naudojama nesaugi klasė *MemoryMarshal*, kuri leidžia dirbti su atmintimi. Visų pirma, šio patobulinimo įgyvendinimui reikia turėti masyvą, o tai papildomai užima *O(n)* laiko ir atminties. Masyvo reikia, kad būtų galima sužinoti jo pradžią (kintamasis *start*) bei pabaigą (kintamasis *end*) atmintyje. Pabaigos kintamasis rodo vienu elementu toliau, nei baigiasi masyvas, ir jis niekada nebus surinktas šiukšlių surinkėjo (angl. *Garbage Collector*). Tačiau tokiu būdu negalima eiti per masyvą nuo galo, nes elementas prieš masyvą gali būti surinktas šiukšlių surinkėjo, o tai gali nulemti skirtingus rezultatus [25]. Kiekvienos iteracijos metu *start* rodyklė yra didinama vienetu naudojant *Unsafe.Add* metodą, kol *Unsafe.IsAddressLessThan* grąžina *false*, o tai reiškia, kad *start* rodyklė rodo į tą pačią atminties vietą kaip ir *end* rodyklė. Galiausiai, laiko ir atminties sudėtingumai yra *O(2n)*.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    string[] rows = File.ReadLines(path).Skip(1).ToArray();
    List<EventModel> events = new(rows.Length);
    ref string start = ref MemoryMarshal.GetArrayDataReference(rows);
    ref string end = ref Unsafe.Add(ref start, rows.Length);

    int lineNumber = 2;
    while (Unsafe.IsAddressLessThan(ref start, ref end))
    {
        string[] columns = start.Split(',');

        if (columns.Length != _colCount)
        {
            start = ref Unsafe.Add(ref start, 1);
            lineNumber++;
            continue;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, lineNumber);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                events.Add(eventModel);
            }
        }
        catch (FormatException)
        {
            continue;
        }
        catch (IndexOutOfRangeException)
        {
            continue;
        }
        finally
        {
            start = ref Unsafe.Add(ref start, 1);
            lineNumber++;
        }
    }

    return events;
}
```

37 pav. Įvykių importavimas iš failo naudojant *while* ciklą su *MemoryMarshal*

3.1.14. Lygiagrečios užklausos

Lygiagrečių užklausų naudojimas pateiktas 38 pav. Tai galima įgyvendinti naudojant PLINQ, kuris yra įjungiamas su *AsParallel* metodu. Kaip ir paprastame LINQ, reikia turėti konkretų masyvą ar sąrašą, todėl yra prarandama *O(n)* laiko ir atminties. PLINQ kodo blokas yra praktiškai identiškas 3.1.7 skyrelyje esančiam blokui. Vienintelis skirtumas yra tas, kad čia neįmanoma išsaugoti apdorojamos eilutės numerio, o tai neleistų įgyvendinti reikalavimų. Galutiniai tokio patobulinimo laiko ir atminties sudėtingumai yra *O(2n)*, nes PLINQ kodo bloką galima suprastinti iki *O(n)* dirbant tik su viena gija.

```
public List<EventModel> Import(string path, DateTime fromDate, DateTime toDate)
{
    List<string> rows = File.ReadLines(path).Skip(1).ToList();
    List<EventModel> events = new(rows.Count);
    object lockObj = new();

    rows.AsParallel().ForAll(line =>
    {
        string[] columns = line.Split(',');

        if (columns.Length != _colCount)
        {
            return;
        }

        try
        {
            EventModel eventModel = ConvertRowToModel(columns, 0);
            DateTime eventDate = new(eventModel.Date.Year, eventModel.Date.Month, eventModel.Date.Day,
                                     eventModel.Time.Hour, eventModel.Time.Minute, eventModel.Time.Second);

            if (DateTime.Compare(eventDate, fromDate) >= 0 && DateTime.Compare(eventDate, toDate) <= 0)
            {
                lock (lockObj)
                {
                    events.Add(eventModel);
                }
            }
        }
        catch (FormatException)
        {
            return;
        }
        catch (IndexOutOfRangeException)
        {
            return;
        }
    });

    return events;
}
```

38 pav. Įvykių importavimas iš failo naudojant PLINQ

3.2. Darbo išsaugojimo tobulinimo galimybės

Darbo išsaugojimas apima pagrindiniame sistemos lange matomų įvykių (jų hierarchijas, paryškinimo spalvas, paslėptus įvykius) bei suformuotų keturių abstrakcijos lygmenų įrašymą į failą. Pagrindiniame lange esančių įvykių negalima lygiagrečiai rašyti į failą, nes būtų prarandami hierarchiniai ryšiai arba jiems atkurti reikėtų realizuoti daug sudėtingesnį algoritmą. Taip pat kelios gijos negalėtų rašyti į tą patį failą, o tai gali ženkliai sumažinti lygiagretumo suteikiamą greičio naudą. Dėl tos pačios priežasties negalima išlygiagretinti įvykių bei keturių abstrakcijos lygmenų rašymo į vieną failą procesų. Tiek įvykiams, tiek abstrakcijoms lygmenims išsaugoti bus naudojamas toks pat būdas. *Foreach* ciklas naudojant LINQ taip pat netinka, nes įvykių hierarchija yra rekursiškai rašoma į failą. Rekursijos naudojimas gali sukelti dėklo perpildymą, todėl ateityje gali reikėti perdaryti įvykių išsaugojimo algoritmą. Visi patobulinimai taip pat sumažina kodo kartojimą, o tai palengvina sistemos palaikymą.

3.2.1. Paprastas *for* ciklas

Paprasto *for* ciklo panaudojimas darbo išsaugojimui pateiktas 39 pav. Įvykiai ir abstrakcijos lygmenys gali būti rašomi į failą asinchroniškai. Įvykiai yra išsaugomi rekursiškai kviečiant metodą kiekvieno įvykio vaikiniams elementams. Įvykių išsaugojimo faile laiko sudėtingumas yra *O(n*m)*, nes per visus sąrašo elementus bus pereinama tik po vieną kartą, tačiau taip pat reikia įvertinti ir jo vaikinių elementų konvertavimą į sąrašą. Čia *n* yra elementų skaičius sąraše, o *m* – didžiausias vaikinių elementų skaičius. Užimamos atminties sudėtingumas taip pat yra *O(n*m)*, nes kiekvieno įvykio vaikus sauganti kolekcija turi būti konvertuota į sąrašą, o tai yra daroma rekursiškai ir atmintyje lieka anksčiau konvertuoti įvykių sąrašai. Jei visi elementai suformuoja įvykių grandinėlę, dėklas gali būti *O(n)* dydžio, o tai gali sukelti dėklo perpildymą. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(n)*, o užimamos atminties sudėtingumas – *O(1)*.

```
private async Task WriteTreeToFile(List<EventViewModel> events, StreamWriter outputStream, int currentLevel)
{
    for (int i = 0; i < events.Count; i++)
    {
        string serializedEventViewModel = events[i].Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        await outputStream.WriteLineAsync(dataToWrite);

        await this.WriteTreeToFile(events[i].Children.ToList(), outputStream, currentLevel + 1);
    }
}

4 references
private async Task WriteAbstractionLevelToFile(List<ISerializableLevel> abstractionLevel, StreamWriter outputStream)
{
    for (int i = 0; i < abstractionLevel.Count; i++)
    {
        string serializedLine = abstractionLevel[i].Serialize();
        await outputStream.WriteLineAsync(serializedLine);
    }
}
```

39 pav. Darbo išsaugojimas naudojant *for* ciklą

3.2.2. Nesaugus *for* ciklas naudojant *Span* iš *List*

Nesaugaus *for* ciklo panaudojimas su *Span* iš *List* yra pateiktas 40 pav. Įvykiai ir abstrakcijos lygmenys negali būti rašomi į failą asinchroniškai, nes yra naudojama *Span* klasė. Dėl šios priežasties gali reikėti iškviesti darbo išsaugojimą su *Task.Run* metodu, kad naudotojo sąsaja liktų interaktyvi. Įvykių išsaugojimo laiko sudėtingumas yra *O(n*m)*, o atminties sudėtingumas – *O(n*m)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(n)*, o užimamos atminties sudėtingumas – *O(1)*.

```
private void WriteTreeToFile(List<EventViewModel> events, StreamWriter outputStream, int currentLevel)
{
    Span<EventViewModel> span = CollectionsMarshal.AsSpan(events);

    for (int i = 0; i < span.Length; i++)
    {
        string serializedEventViewModel = span[i].Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        outputStream.WriteLine(dataToWrite);

        this.WriteTreeToFile(new List<EventViewModel>(span[i].Children), outputStream, currentLevel + 1);
    }
}

4 references
private void WriteAbstractionLevelToFile(List<ISerializableLevel> abstractionLevel, StreamWriter outputStream)
{
    Span<ISerializableLevel> span = CollectionsMarshal.AsSpan(abstractionLevel);

    for (int i = 0; i < span.Length; i++)
    {
        string serializedLine = span[i].Serialize();
        outputStream.WriteLine(serializedLine);
    }
}
```

40 pav. Darbo išsaugojimas naudojant *for* ciklą ir konvertuojant *List* į *Span*

3.2.3. Nesaugus *for* ciklas naudojant *Span* iš masyvo

Nesaugaus *for* ciklo panaudojimas su *Span* iš masyvo parodytas 41 pav. Įvykiai ir abstrakcijos lygmenys čia taip pat negali būti rašomi į failą asinchroniškai. Prieš kviečiant šiuos metodus, duomenis saugančias kolekcijas reikia konvertuoti į masyvus, o tai užima *O(n)* laiko ir atminties. Dėl šios priežasties, įvykių išsaugojimo laiko sudėtingumas yra *O(n*m+n)*, o atminties sudėtingumas taip pat yra *O(n*m+n)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(2n)*, o užimamos atminties sudėtingumas – *O(n)*.

```
private void WriteTreeToFile(EventViewModel[] events, StreamWriter outputStream, int currentLevel)
{
    Span<EventViewModel> span = events.AsSpan();

    for (int i = 0; i < span.Length; i++)
    {
        string serializedEventViewModel = span[i].Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        outputStream.WriteLine(dataToWrite);

        this.WriteTreeToFile(span[i].Children.ToArray(), outputStream, currentLevel + 1);
    }
}

4 references
private void WriteAbstractionLevelToFile(ISerializableLevel[] abstractionLevel, StreamWriter outputStream)
{
    Span<ISerializableLevel> span = abstractionLevel.AsSpan();

    for (int i = 0; i < span.Length; i++)
    {
        string serializedLine = span[i].Serialize();
        outputStream.WriteLine(serializedLine);
    }
}
```

41 pav. Darbo išsaugojimas naudojant *for* ciklą ir konvertuojant masyvą į *Span*

3.2.4. Nesaugus *for* ciklas naudojant *MemoryMarshal*

Nesaugaus *for* ciklo naudojimas su *MemoryMarshal* yra parodytas 42 pav. Čia naudojama nesaugi klasė (*MemoryMarshal*) taip pat negali būti asinchroniniame metode. Kaip ir anksčiau, pirmo masyvo

elemento vieta atmintyje yra gaunama *MemoryMarshal.GetArrayDataReference* metodu. Prie tos vietos galima pridėti *offset* su *Unsafe.Add* metodu ir gauti reikiamą masyvo elementą iš atminties. Įvykių išsaugojimo laiko ir atminties sudėtingumas yra *O(n*m+n)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(2n)*, o užimamos atminties sudėtingumas – *O(n)*.

```
private void WriteTreeToFile(EventViewModel[] events, StreamWriter outputStream, int currentLevel)
{
    ref EventViewModel searchSpace = ref MemoryMarshal.GetArrayDataReference(events);

    for (int i = 0; i < events.Length; i++)
    {
        EventViewModel eventViewModel = Unsafe.Add(ref searchSpace, i);
        string serializedEventViewModel = eventViewModel.Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        outputStream.WriteLine(dataToWrite);

        this.WriteTreeToFile(eventViewModel.Children.ToArray(), outputStream, currentLevel + 1);
    }
}

4 references
private void WriteAbstractionLevelToFile(ISerializableLevel[] abstractionLevel, StreamWriter outputStream)
{
    ref ISerializableLevel searchSpace = ref MemoryMarshal.GetArrayDataReference(abstractionLevel);

    for (int i = 0; i < abstractionLevel.Length; i++)
    {
        ISerializableLevel highLevelEvent = Unsafe.Add(ref searchSpace, i);
        string serializedLine = highLevelEvent.Serialize();
        outputStream.WriteLine(serializedLine);
    }
}
```

42 pav. Darbo išsaugojimas naudojant *for* ciklą su *MemoryMarshal*

3.2.5. Paprastas *foreach* ciklas

Paprasto *foreach* ciklo naudojimas darbo išsaugojimui pateiktas 43 pav. Įvykiai ir abstrakcijos lygmenys gali būti asinchroniškai rašomi į failą. Išsaugant įvykius, nebereikia vaikinių elementų konvertuoti į sąrašą ar masyvą, o tai sutaupo laiko ir atminties. Įvykių išsaugojimo faile laiko sudėtingumas yra *O(n)*, nes nebereikia konvertuoti kolekcijos, o užimamos atminties sudėtingumas taip pat yra *O(n)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(n)*, o užimamos atminties sudėtingumas nekinta ir lieka konstanta.

Šis būdas šiuo metu yra įgyvendintas kompiuterio įvykių laike atkūrimo sistemoje. Čia taip pat nebuvo naudojama *ISerializableLevel* sąsaja, todėl galima pastebėti dubliuojamo kodo.

```
private static async Task WriteTreeToFile(IEnumerable<EventViewModel> events, StreamWriter outputStream, int currentLevel)
{
    foreach (EventViewModel eventViewModel in events) {
        string serializedEventViewModel = eventViewModel.Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        await outputStream.WriteLineAsync(dataToWrite);

        await WriteTreeToFile(eventViewModel.Children, outputStream, currentLevel + 1);
    }
}

1 reference
private static async Task WriteHighLevelEventsToFile(IEnumerable<HighLevelEventViewModel> highLevelEvents, StreamWriter outputStream)
{
    foreach (HighLevelEventViewModel highLevelEvent in highLevelEvents)
    {
        string serializedLine = highLevelEvent.Serialize();
        await outputStream.WriteLineAsync(serializedLine);
    }
}

1 reference
private static async Task WriteLowLevelEventsToFile(IEnumerable<LowLevelEventViewModel> lowLevelEvents, StreamWriter outputStream)
{
    foreach (LowLevelEventViewModel lowLevelEvent in lowLevelEvents)
    {
        string serializedLine = lowLevelEvent.Serialize();
        await outputStream.WriteLineAsync(serializedLine);
    }
}

1 reference
private static async Task WriteHighLevelArtefactsToFile(IEnumerable<HighLevelArtefactViewModel> highLevelArtefacts, StreamWriter outputStream)
{
    foreach (HighLevelArtefactViewModel highLevelArtefact in highLevelArtefacts)
    {
        string serializedLine = highLevelArtefact.Serialize();
        await outputStream.WriteLineAsync(serializedLine);
    }
}

1 reference
private static async Task WriteLowLevelArtefactsToFile(IEnumerable<LowLevelArtefactViewModel> lowLevelArtefacts, StreamWriter outputStream)
{
    foreach (LowLevelArtefactViewModel lowLevelArtefact in lowLevelArtefacts)
    {
        string serializedLine = lowLevelArtefact.Serialize();
        await outputStream.WriteLineAsync(serializedLine);
    }
}
```

43 pav. Darbo išsaugojimas naudojant *foreach* ciklą

3.2.6. Nesaugus *foreach* ciklas naudojant *Span* iš *List*

Nesaugaus *foreach* ciklo naudojimas su *Span* iš *List* yra pateiktas 44 pav. Čia, kaip ir 3.2.2 skyrelyje, įvykiai ir abstrakcijos lygmenys negali būti rašomi į failą asinchroniškai. Laiko ir atminties sudėtingumas įvykių išsaugojimui lieka toks pat – *O(n*m)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas taip pat lieka *O(n)*, o užimamos atminties sudėtingumas – *O(1)*.

```
private void WriteTreeToFile(List<EventViewModel> events, StreamWriter outputStream, int currentLevel)
{
    foreach (EventViewModel eventViewModel in CollectionsMarshal.AsSpan(events))
    {
        string serializedEventViewModel = eventViewModel.Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        outputStream.WriteLine(dataToWrite);

        this.WriteTreeToFile(eventViewModel.Children.ToList(), outputStream, currentLevel + 1);
    }
}

4 references
private void WriteAbstractionLevelToFile(List<ISerializableLevel> abstractionLevel, StreamWriter outputStream)
{
    foreach (ISerializableLevel eventInLevel in CollectionsMarshal.AsSpan(abstractionLevel))
    {
        string serializedLine = eventInLevel.Serialize();
        outputStream.WriteLine(serializedLine);
    }
}
```

44 pav. Darbo išsaugojimas naudojant *foreach* ciklą ir konvertuojant *List* į *Span*

3.2.7. Nesaugus *foreach* ciklas naudojant *Span* iš masyvo

Nesaugaus *foreach* ciklo naudojimas su *Span* iš masyvo parodytas 45 pav. Įvykiai ir abstrakcijos lygmenys čia taip pat negali būti rašomi į failą asinchroniškai. Be to, kolekcijas reikia konvertuoti į masyvus. Įvykių išsaugojimo laiko sudėtingumas yra *O(2n)*, o atminties sudėtingumas taip pat yra *O(2n)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(2n)*, o užimamos atminties sudėtingumas – *O(n)*.

```
private void WriteTreeToFile(EventViewModel[] events, StreamWriter outputStream, int currentLevel)
{
    foreach (EventViewModel eventViewModel in events.AsSpan())
    {
        string serializedEventViewModel = eventViewModel.Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        outputStream.WriteLine(dataToWrite);

        this.WriteTreeToFile(eventViewModel.Children.ToArray(), outputStream, currentLevel + 1);
    }
}

4 references
private void WriteAbstractionLevelToFile(ISerializableLevel[] abstractionLevel, StreamWriter outputStream)
{
    foreach (ISerializableLevel eventInLevel in abstractionLevel.AsSpan())
    {
        string serializedLine = eventInLevel.Serialize();
        outputStream.WriteLine(serializedLine);
    }
}
```

45 pav. Darbo išsaugojimas naudojant *foreach* ciklą ir konvertuojant masyvą į *Span*

3.2.8. Paprastas *while* ciklas

Paprasto *while* ciklo naudojimas darbo išsaugojimui yra parodytas 46 pav. Šiame būde yra naudojamas *IEnumerator* objektas, gautas iš atitinkamo *IEnumerable* objekto. Vėliau galima pereiti per visus *IEnumerable* elementus su *MoveNext* metodu. Įvykių išsaugojimo faile laiko sudėtingumas yra *O(n)*, nes nereikia konvertuoti kolekcijos, o užimamos atminties sudėtingumas taip pat yra *O(n)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(n)*, o užimamos atminties sudėtingumas yra *O(n)*.

```
private async Task WriteTreeToFile(IEnumerable<EventViewModel> events, StreamWriter outputStream, int currentLevel)
{
    IEnumerator<EventViewModel> enumerator = events.GetEnumerator();

    while (enumerator.MoveNext())
    {
        string serializedEventViewModel = enumerator.Current.Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        await outputStream.WriteLineAsync(dataToWrite);

        await this.WriteTreeToFile(enumerator.Current.Children, outputStream, currentLevel + 1);
    }
}

4 references
private async Task WriteAbstractionLevelToFile(IEnumerable<ISerializableLevel> abstractionLevel, StreamWriter outputStream)
{
    IEnumerator<ISerializableLevel> enumerator = abstractionLevel.GetEnumerator();

    while (enumerator.MoveNext())
    {
        string serializedLine = enumerator.Current.Serialize();
        await outputStream.WriteLineAsync(serializedLine);
    }
}
```

46 pav. Darbo išsaugojimas naudojant *while* ciklą

3.2.9. Nesaugus *while* ciklas naudojant *MemoryMarshal*

Nesaugaus *while* ciklo naudojimas su *MemoryMarshal* pateiktas 47 pav. Kaip ir 3.2.4 skyrelyje, šis būdas negali būti naudojamas asinchroniniuose metoduose. Be to, reikia gauti masyvą, o tai užima *O(n)* atminties ir laiko. Bendras įvykių išsaugojimo laiko ir atminties sudėtingumas yra *O(n*m+n)*. Kiekvieno abstrakcijos lygmens išsaugojimo laiko sudėtingumas yra *O(2n)*, o užimamos atminties sudėtingumas – *O(n)*.

```
private void WriteTreeToFile(EventViewModel[] events, StreamWriter outputStream, int currentLevel)
{
    ref EventViewModel start = ref MemoryMarshal.GetArrayDataReference(events);
    ref EventViewModel end = ref Unsafe.Add(ref start, events.Length);

    while (Unsafe.IsAddressLessThan(ref start, ref end))
    {
        string serializedEventViewModel = start.Serialize();
        string dataToWrite = string.Format("{0}{1}", new string('\t', currentLevel), serializedEventViewModel);
        outputStream.WriteLine(dataToWrite);

        this.WriteTreeToFile(start.Children.ToArray(), outputStream, currentLevel + 1);

        start = ref Unsafe.Add(ref start, 1);
    }
}

4 references
private void WriteAbstractionLevelToFile(ISerializableLevel[] abstractionLevel, StreamWriter outputStream)
{
    ref ISerializableLevel start = ref MemoryMarshal.GetArrayDataReference(abstractionLevel);
    ref ISerializableLevel end = ref Unsafe.Add(ref start, abstractionLevel.Length);

    while (Unsafe.IsAddressLessThan(ref start, ref end))
    {
        string serializedLine = start.Serialize();
        outputStream.WriteLine(serializedLine);

        start = ref Unsafe.Add(ref start, 1);
    }
}
```

47 pav. Darbo išsaugojimas naudojant *while* ciklą su *MemoryMarshal*

3.3. Darbo įkėlimo tobulinimo galimybės

Darbo įkėlimas apima įvykių (jų hierarchijas, paryškinimo spalvas, paslėptus įvykius) bei suformuotų keturių abstrakcijos lygmenų nuskaitymą iš failo. Kaip ir išsaugant įvykius, jų negalima lygiagrečiai nuskaityti iš failo, nes bus prarandami ryšiai tarp įvykių. Darbo įkėlimui būtų galima naudoti panašius patobulinimus kaip 3.1 ir 3.2 poskyriuose. Dėl šios priežasties darbo įkėlimui bus sukurtas tik vienas patobulinimas.

3.3.1. Failo skaitymas po vieną eilutę naudojant *StreamReader*

Įvykių skaitymas po vieną eilutę yra parodytas 48 pav. Iš failo yra skaitoma po vieną eilutę tol, kol nuskaityta eilutė nėra tuščia. Norint atkurti įvykių hierarchiją, kiekvienai nuskaitytai įvykio eilutei reikia nustatyti jos gylį, kad būtų galima nustatyti, kokiam hierarchijos lygiui priklauso tas įvykis. Reikiamo hierarchijos lygio nustatymas smarkiai padidina šio metodo sudėtingumą. Šio metodo atminties sudėtingumas yra *O(2n)*, nes papildomai yra naudojamas *Stack* objektas, kuris padeda atkurti teisingą įvykių hierarchiją. Laiko sudėtingumas yra $O(n^2)$, nes reikia nuskaityti visas įvykių eilutes bei išimti visus elementus iš *Stack* objekto, kuriame yra vaikiniai elementai.

Abstrakcijos lygmenų skaitymas po vieną eilutę yra pateiktas 49 pav. Čia galima pastebėti nemažai kodo kartojimo. Vienintelė eilutė, kuri nėra kartojama, yra skirta atitinkamo abstrakcijos lygio įvykio atkūrimui. Kiekviena eilutė, pagal kurią yra atkuriamas įvykis, taip pat yra nuosekliai skaitoma po vieną. Kiekvieno abstrakcijos lygio įkėlimo laiko ir atminties sudėtingumas yra *O(n)*.

Šis būdas buvo įgyvendintas tiriamoje kompiuterio įvykių laike atkūrimo sistemoje.

```
private static async Task<List<EventViewModel>> LoadEvents(StreamReader inputStream)
{
    List<EventViewModel> events = new();
    string row = await inputStream.ReadLineAsync();
    int currentDepth = 0;
    Stack<EventViewModel> stack = new();

    while (row != "")
    {
        string[] columns = row.Split(',');
        int depth = GetDepth(columns[0]);
        columns[0] = columns[0].Trim(new char[] { '\t' });

        EventModel eventModel = ConvertRowToModel(columns);
        EventViewModel eventViewModel = ConvertToViewModel(eventModel, columns);

        if (depth == 0)
        {
            events.Add(eventViewModel);
        }
        else if (depth == currentDepth)
        {
            while (depth <= currentDepth) {
                stack.Pop();
                currentDepth--;
            }

            currentDepth++;

            EventViewModel current = stack.Peek();
            current.AddChild(eventViewModel);
        }
        else if (depth > currentDepth)
        {
            currentDepth++;
            EventViewModel current = stack.Peek();
            current.AddChild(eventViewModel);
        }
        else if (depth < currentDepth)
        {
            while (depth <= currentDepth)
            {
                stack.Pop();
                currentDepth--;
            }

            currentDepth++;
            EventViewModel current = stack.Peek();
            current.AddChild(eventViewModel);
        }

        stack.Push(eventViewModel);
        row = await inputStream.ReadLineAsync();
    }

    return events;
}
```

48 pav. Įvykių hierarchijos įkėlimas skaitant failą po vieną eilutę naudojant *StreamReader*

```
private static async Task<List<HighLevelEventViewModel>> LoadHighLevelEvents(StreamReader inputStream)
{
    List<HighLevelEventViewModel> highLevelEvents = new();

    string row = await inputStream.ReadLineAsync();

    while (row != "")
    {
        string[] columns = row.Split(',');
        HighLevelEventViewModel highLevelEvent = ConvertRowToHighLevelEvent(columns);
        highLevelEvents.Add(highLevelEvent);

        row = await inputStream.ReadLineAsync();
    }

    return highLevelEvents;
}

1 reference
private static async Task<List<LowLevelEventViewModel>> LoadLowLevelEvents(StreamReader inputStream)
{
    List<LowLevelEventViewModel> lowLevelEvents = new();

    string row = await inputStream.ReadLineAsync();

    while (row != "")
    {
        string[] columns = row.Split(',');
        LowLevelEventViewModel lowLevelEvent = ConvertRowToLowLevelEvent(columns);
        lowLevelEvents.Add(lowLevelEvent);

        row = await inputStream.ReadLineAsync();
    }

    return lowLevelEvents;
}

1 reference
private static async Task<List<HighLevelArtefactViewModel>> LoadHighLevelArtefacts(StreamReader inputStream)
{
    List<HighLevelArtefactViewModel> highLevelArtefacts = new();

    string row = await inputStream.ReadLineAsync();

    while (row != "")
    {
        string[] columns = row.Split(',');
        HighLevelArtefactViewModel highLevelArtefact = ConvertRowToHighLevelArtefact(columns);
        highLevelArtefacts.Add(highLevelArtefact);

        row = await inputStream.ReadLineAsync();
    }

    return highLevelArtefacts;
}

1 reference
private static async Task<List<LowLevelArtefactViewModel>> LoadLowLevelArtefacts(StreamReader inputStream)
{
    List<LowLevelArtefactViewModel> lowLevelArtefacts = new();

    string row = await inputStream.ReadLineAsync();

    while (row != null)
    {
        string[] columns = row.Split(',');
        LowLevelArtefactViewModel lowLevelArtefact = ConvertRowToLowLevelArtefact(columns);
        lowLevelArtefacts.Add(lowLevelArtefact);

        row = await inputStream.ReadLineAsync();
    }

    return lowLevelArtefacts;
}
```

49 pav. Abstrakcijos lygmenų įkėlimas skaitant failą po vieną eilutę naudojant *StreamReader*

3.3.2. Darbo įkėlimas naudojant *Factory* projektavimo šabloną

Įvykių įkėlimas šiame patobulinime yra atliekamas kaip ir 3.3.1 skyrelyje. Skirtumas yra tas, kad čia naudojamas *IEnumerable* objektas failo eilučių skaitymui. Įvykių įkėlimo algoritmas yra parodytas 50 pav. Metodui yra paduodamas *IEnumerator* objektas, kuris leidžia pereiti per visus *IEnumerable* kolekcijos elementus. Šio patobulinimo sudėtingumas nesikeičia, tik naudojamas kitoks failo skaitymo būdas. Laiko sudėtingumas yra $O(n^2)$, o atminties sudėtingumas – $O(2n)$.

```
private static List<EventViewModel> LoadEvents(IEnumerator<string> enumerator)
{
    List<EventViewModel> events = new();
    Stack<EventViewModel> stack = new();
    int currentDepth = 0;

    while (enumerator.MoveNext() && string.IsNullOrEmpty(enumerator.Current) == false)
    {
        string[] columns = enumerator.Current.Split(',');
        int depth = GetDepth(columns[0]);
        columns[0] = columns[0].Trim(new char[] { '\t' });

        EventModel eventModel = ConvertRowToModel(columns);
        EventViewModel eventViewModel = ConvertToViewModel(eventModel, columns);

        if (depth == 0)
        {
            events.Add(eventViewModel);
        }
        else if (depth == currentDepth)
        {
            while (depth <= currentDepth)
            {
                stack.Pop();
                currentDepth--;
            }

            currentDepth++;

            EventViewModel current = stack.Peek();
            current.AddChild(eventViewModel);
        }
        else if (depth > currentDepth)
        {
            currentDepth++;
            EventViewModel current = stack.Peek();
            current.AddChild(eventViewModel);
        }
        else if (depth < currentDepth)
        {
            while (depth <= currentDepth)
            {
                stack.Pop();
                currentDepth--;
            }

            currentDepth++;
            EventViewModel current = stack.Peek();
            current.AddChild(eventViewModel);
        }

        stack.Push(eventViewModel);
    }

    return events;
}
```

50 pav. Įvykių hierarchijos įkėlimas naudojant *Factory* projektavimo šabloną

Abstrakcijos lygmenų įkėlimas pasinaudojant *Factory* šablonu pateiktas 51-52 pav. Abstrakcijos lygmens nuskaitymo metodui reikia paduoti, kuris lygmuo yra skaitomas. Skaitant failą, yra iškviečiamas 52 pav. parodytas metodas, kuris atkuria reikiamo abstrakcijos lygmens įvykį. Laiko ir atminties sudėtingumai lieka $O(n)$, tačiau yra sumažinama kodo kartojimo ir gali būti palengvinamas

jo palaikymas. Šis patobulinimas sumažina kodo kartojimą bei leidžia gana lengvai pridėti naujus abstrakcijos lygmenis.

```
private List<ISerializableLevel> LoadAbstractionLevel(IEnumerator<string> enumerator, AbstractionLevel abstractionLevel)
{
    List<ISerializableLevel> abstractionEvents = new();

    while (enumerator.MoveNext() && string.IsNullOrEmpty(enumerator.Current) == false)
    {
        ISerializableLevel abstractionEvent = _factory.CreateAbstractionLevel(enumerator.Current, abstractionLevel);
        abstractionEvents.Add(abstractionEvent);
    }

    return abstractionEvents;
}
```

51 pav. Abstrakcijos lygmenų įkėlimas naudojant *Factory* projektavimo šabloną

```
public ISerializableLevel CreateAbstractionLevel(string row, AbstractionLevel level)
{
    return level switch
    {
        AbstractionLevel.HighLevelEvent => HighLevelEventViewModel.Deserialize(row),
        AbstractionLevel.LowLevelEvent => LowLevelEventViewModel.Deserialize(row),
        AbstractionLevel.HighLevelArtefact => HighLevelArtefactViewModel.Deserialize(row),
        AbstractionLevel.LowLevelArtefact => LowLevelArtefactViewModel.Deserialize(row),
        _ => throw new NotSupportedException(),
    };
}
```

52 pav. Abstrakcijos lygmenų įkėlimui naudojamas *Factory* metodas

3.4. Abstrakcijos lygmenų formavimo tobulinimo galimybės

Abstrakcijos lygmenų formavimas apima keturių lygių sukūrimą iš sistemoje esančių įvykių. Tie lygmenys yra formuojami nepriklausomai vienas nuo kito, todėl galima pamėginti juos formuoti lygiagrečiai. Čia taip pat reikia iteruoti per visus įvykius, tačiau tai nebus šio funkcionalumo tyrimo dalis. Šio funkcionalumo tyrime visas dėmesys yra skiriamas kiekvieno abstrakcijos lygmens formavimo efektyvumui bei palaikomumui.

3.4.1. Abstrakcijos lygmenų formavimas naudojant *switch* sakinį

Šiuo metu kompiuterio įvykių laike atkūrimo sistemoje įgyvendintas abstrakcijos lygmenų formavimo algoritmas naudoja *switch* sakinį, kuris yra alternatyva *if* sąlygos sakiniui. Kiekvienas lygmuo turi būti formuojamas nepriklausomai kaip foninė užduotis, kad nebūtų užblokuota pagrindinė naudotojo sąsajos gija (žr. 53 pav.). Čia yra sukuriamas keturių asinchroninių užduočių sąrašas. Kiekviena užduotis yra atsakinga už atitinkamo abstrakcijos lygio suformavimą. Metodo gale yra palaukiama, kol kiekviena užduotis baigs savo darbą.

```
List<EventViewModel> events = _store.GetStoredEventViewModelsAsOneLevel();
List<DispatcherOperation> tasks = new()
{
    Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, () =>
    {
        List<HighLevelEventViewModel> highLevelEvents = _highLevelEventsAbstractor.FormHighLevelEvents(events);
        _viewModel.LoadHighLevelEvents(highLevelEvents);
    }),
    Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, () =>
    {
        List<LowLevelEventViewModel> lowLevelEvents = _lowLevelEventsAbstractor.FormLowLevelEvents(events);
        _viewModel.LoadLowLevelEvents(lowLevelEvents);
    }),
    Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, () =>
    {
        List<HighLevelArtefactViewModel> highLevelArtefacts = _highLevelArtefactsAbstractor.FormHighLevelArtefacts(events);
        _viewModel.LoadHighLevelArtefacts(highLevelArtefacts);
    }),
    Application.Current.Dispatcher.BeginInvoke(DispatcherPriority.Background, () =>
    {
        List<LowLevelArtefactViewModel> lowLevelArtefacts = _lowLevelArtefactsAbstractor.FormLowLevelArtefacts(events);
        _viewModel.LoadLowLevelArtefacts(lowLevelArtefacts);
    })
};

foreach (DispatcherOperation task in tasks)
{
    await task;
}
```

53 pav. Abstrakcijos lygmenų formavimo iškvietimas

Aukšto lygio įvykių formavimas naudojant *switch* sakinį yra pateiktas 54 pav. Penki skirtingi įvykių tipai yra tikrinami *switch* sakinio viduje, o tai stipriai išplečia metodo eilučių skaičių. Taip pat ši klasė turi žinoti, kaip reikia elgtis su kiekvienu įvykio tipu, kad galėtų suformuoti aukšto lygio įvykį. Visa tai labai stipriai išplečia klasės dydį ir apsunkina jos suprantamumą ir palaikomumą. Šio metodo atminties sudėtingumas yra $O(n)$, o laiko sudėtingumas – $O(n^2)$, nes gali reikėti pereiti per tokio pačio ilgio sąrašą dar kartą tame pačiame cikle.

```
public List<HighLevelEventViewModel> FormHighLevelEvents(List<EventViewModel> events)
{
    List<HighLevelEventViewModel> highLevelEvents = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        if (events[i].MACB.Contains('B'))
        {
            switch (events[i].Source)
            {
                case "LOG":
                    HighLevelEventViewModel logEvent = this.FormEventFromLogSource(events[i]);
                    int lastLogEventIndex = FindLastEventIndexOf(highLevelEvents, events[i].FullDate, "LOG");

                    if (lastLogEventIndex != -1)
                    {
                        highLevelEvents.RemoveAt(lastLogEventIndex);
                        highLevelEvents.Insert(lastLogEventIndex, logEvent);
                    }
                    else
                    {
                        highLevelEvents.Add(logEvent);
                    }

                    break;
                case "LNK":
                    HighLevelEventViewModel lnkEvent = this.FormEventFromLnkSource(events[i]);
                    int lastLnkEventIndex = FindLastEventIndexOf(highLevelEvents, events[i].FullDate, "LNK");

                    if (lastLnkEventIndex != -1)
                    {
                        highLevelEvents.RemoveAt(lastLnkEventIndex);
                        highLevelEvents.Insert(lastLnkEventIndex, lnkEvent);
                    }
                    else
                    {
                        highLevelEvents.Add(lnkEvent);
                    }

                    break;
                case "META":
                    int lastMetaEventIndex = FindLastEventIndexOf(highLevelEvents, events[i].FullDate, "META");

                    if (lastMetaEventIndex == -1)
                    {
                        HighLevelEventViewModel metaEvent = FormEventFromMetaSource(events[i]);
                        highLevelEvents.Add(metaEvent);
                    }

                    break;
                case "OLECF":
                    HighLevelEventViewModel olecfEvent = FormEventFromOlecfSource(events[i]);

                    if (IsOlecfEventValid(highLevelEvents[^1], olecfEvent))
                    {
                        highLevelEvents.Add(olecfEvent);
                    }

                    break;
                case "PE":
                    int lastPeEventIndex = FindLastEventIndexOf(highLevelEvents, events[i].FullDate, "PE");

                    if (lastPeEventIndex == -1 && _abstractorUtils.IsValidPeEvent(events[i]))
                    {
                        HighLevelEventViewModel peEvent = FormEventFromPeSource(events[i]);
                        highLevelEvents.Add(peEvent);
                    }

                    break;
                default:
                    break;
            }
        }

        if (events[i].Source == "WEBHIST" && _abstractorUtils.IsValidWebhistLine(events[i]))
        {
            HighLevelEventViewModel webhistEvent = this.FormEventFromWebhistSource(events[i]);

            if (webhistEvent is not null && IsWebhistEventValid(highLevelEvents, webhistEvent))
            {
                highLevelEvents.Add(webhistEvent);
            }
        }
    }

    return highLevelEvents;
}
```

54 pav. Aukšto lygio įvykių formavimas naudojant *switch* sakinį

Panaši situacija yra ir su žemo lygio įvykių lygmens formavimu (žr. 55 pav.). Vietoj penkių tipų, reikia tikrinti devynis įvykių tipus. Tai dar labiau išplečia klasės dydį ir padidina jos sudėtingumą. Šio metodo atminties sudėtingumas taip pat yra $O(n)$, o laiko sudėtingumas – $O(n^2)$.

```
public List<LowLevelEventViewModel> FormLowLevelEvents(List<EventViewModel> events)
{
    List<LowLevelEventViewModel> lowLevelEvents = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        LowLevelEventViewModel lowLevelEvent = null;

        switch (events[i].Source)
        {
            case "WEBHIST":
                if (_highLevelAbstractorUtils.IsValidWebhistLine(events[i]))
                {
                    lowLevelEvent = this.FormEventFromWebhistSource(events[i]);

                    if (!IsWebhistEventValid(lowLevelEvents, lowLevelEvent))
                    {
                        lowLevelEvent = null;
                    }
                }

                break;
            case "FILE":
                int validEventIndex = this.GetValidFileEventIndex(events, i);

                if (validEventIndex != -1 && !this.DoesNeedComposing(events, i - 1, events[validEventIndex]))
                {
                    lowLevelEvent = this.FormEventFromFileSource(events[validEventIndex]);

                    int offset = validEventIndex - i;
                    i += offset;

                    bool isSearchedFormat = events[i].Format == "lnk/shell_items" || events[i].Format == "filestat";
                    if (isSearchedFormat)
                    {
                        DateTime date = events[i].FullDate;

                        while (i < events.Count - 1 && events[i].Source == "FILE" && events[i + 1].FullDate.CompareTo(date) == 0)
                        {
                            i++;
                        }
                    }
                }

                break;
            case "LNK":
                if (!this.DoesNeedComposing(events, i - 1, events[i]))
                {
                    lowLevelEvent = this.FormEventFromLnkSource(events[i]);
                }

                break;
            case "LOG":
                if (!this.DoesNeedComposing(events, i - 1, events[i]))
                {
                    lowLevelEvent = this.FormEventFromLogSource(events[i]);
                }

                break;
            case "META":
                lowLevelEvent = this.FormEventFromMetaSource(events[i]);
                break;
            case "REG":
                if (_lowLevelAbstractorUtils.IsValidRegEvent(events[i]))
                {
                    lowLevelEvent = this.FormEventFromRegSource(events[i]);
                }

                break;
            case "OLECF":
                while (i < events.Count - 1 && AreOlecfEventsOfSameTime(events[i], events[i + 1]))
                {
                    i++;
                }

                lowLevelEvent = FormEventFromOlecfSource(events[i]);
                break;
            case "PE":
                if (_highLevelAbstractorUtils.IsValidPeEvent(events[i]))
                {
                    lowLevelEvent = FormEventFromPeSource(events[i]);
                }

                break;
            case "RECBIN":
                lowLevelEvent = this.FormEventFromRecbinSource(events[i]);
                break;
            default:
                break;
        }

        if (lowLevelEvent is not null)
        {
            lowLevelEvents.Add(lowLevelEvent);
            NormalizeEvents(lowLevelEvents, events, events[i]);
        }
    }

    return lowLevelEvents;
}
```

55 pav. Žemo lygio įvykių formavimas naudojant *switch* sakinį

Dar blogiau yra su aukšto lygio artifaktų formavimu (žr. 56 pav.). Nors yra tikrinami aštuoni įvykių tipai, metodo dydis daug labiau išauga dėl sudėtingesnių metodologinių reikalavimų šio lygmens formavimui. Laiko sudėtingumas yra $O(n^2)$, o atminties sudėtingumas – $O(n)$.

```
public List<HighLevelArtefactViewModel> FormHighLevelArtefacts(List<EventViewModel> events)
{
    List<HighLevelArtefactViewModel> highLevelArtefacts = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        HighLevelArtefactViewModel highLevelArtefact = null;

        switch (events[i].Source)
        {
            case "WEBHIST":
                if (_highLevelEventsAbstractorUtils.IsValidWebhistLine(events[i]))
                {
                    highLevelArtefact = this.FormEventFromWebhistSource(events[i]);

                    if (!IsWebhistEventValid(highLevelArtefacts, highLevelArtefact))
                    {
                        highLevelArtefact = null;
                    }
                }

                break;
            case "LNK":
                highLevelArtefact = this.FormEventFromLnkSource(events[i]);

                if (!IsLnkEventValid(highLevelArtefacts, highLevelArtefact))
                {
                    highLevelArtefact = null;
                }

                break;
            case "FILE":
                if (i == 0)
                {
                    highLevelArtefact = this.FormEventFromFileSource(events[i]);
                    break;
                }

                if (!_highLevelArtefactsAbstractorUtils.IsFileDuplicateOfLnk(events, i - 1, events[i]))
                {
                    int fileCountInRowAtSameMinute = _highLevelArtefactsAbstractorUtils.GetFileCountInRowAtSameMinute(events, i);
                    int endIndex = i + fileCountInRowAtSameMinute;

                    if (fileCountInRowAtSameMinute <= _fileArtefactsInRowCount)
                    {
                        for (int index = i; index < endIndex; index++)
                        {
                            HighLevelArtefactViewModel artefact = this.FormEventFromFileSource(events[i]);
                            // single artefact per second
                            if (IsFileEventValid(highLevelArtefacts, artefact))
                            {
                                highLevelArtefacts.Add(artefact);
                            }
                        }
                    }
                    else
                    {
                        List<int> validIndices = _highLevelArtefactsAbstractorUtils.GetValidFileEventIndices(events, i, endIndex);
                        for (int index = 0; index < validIndices.Count; index++)
                        {
                            HighLevelArtefactViewModel artefact = this.FormEventFromFileSource(events[validIndices[index]]);
                            highLevelArtefacts.Add(artefact);
                        }
                    }

                    i += fileCountInRowAtSameMinute - 1;
                }

                break;
            case "LOG":
                highLevelArtefact = this.FormEventFromLogSource(events[i]);

                if (!this.IsLogEventValid(events, i))
                {
                    highLevelArtefact = null;
                }

                break;
            case "REG":
                highLevelArtefact = this.FormEventFromRegSource(events[i]);

                if (!IsRegEventValid(highLevelArtefacts, highLevelArtefact))
                {
                    highLevelArtefact = null;
                }

                break;
            case "META":
                highLevelArtefact = this.FormEventFromMetaSource(events[i]);
                break;
            case "OLECF":
                while (i < events.Count - 1 && AreOlecfEventsOfSameTime(events[i], events[i + 1]))
                {
                    i++;
                }

                highLevelArtefact = FormEventFromOlecfSource(events[i]);
                break;
            case "PE":
                if (this.IsPeEventValid(highLevelArtefacts, events[i]))
                {
                    highLevelArtefact = FormEventFromPeSource(events[i]);
                }

                break;
            default:
                break;
        }

        if (IsEventValid(highLevelArtefacts, highLevelArtefact))
        {
            highLevelArtefacts.Add(highLevelArtefact);
        }
    }

    return highLevelArtefacts;
}
```

56 pav. Aukšto lygio artifaktų formavimas naudojant *switch* sakinį

Daug geresnė situacija yra su žemo lygio artifaktų formavimu (žr. 57 pav.), nes čia reikia tikrinti tik tris įvykių tipus ir reikalavimai šio lygmens formavimui nėra tokie griežti. Deja, bet laiko sudėtingumas ir čia lieka $O(n^2)$, o atminties sudėtingumas – $O(n)$.

```
public List<LowLevelArtefactViewModel> FormLowLevelArtefacts(List<EventViewModel> events, double periodInMinutes = 1.0)
{
    List<LowLevelArtefactViewModel> lowLevelArtefacts = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        LowLevelArtefactViewModel lowLevelArtefact = null;

        switch (events[i].Source)
        {
            case "WEBHIST":
                if (_lowLevelArtefactsAbstractorUtils.IsValidWebhistLine(events[i].SourceType, events[i].Type))
                {
                    lowLevelArtefact = this.FormEvent(events[i]);

                    if (lowLevelArtefact.SourceType.ToLower().Contains("cookies"))
                    {
                        lowLevelArtefact = this.NormalizeCookie(lowLevelArtefacts, lowLevelArtefact);
                    }
                }

                break;
            case "LNK":
                lowLevelArtefact = this.FormEvent(events[i]);
                break;
            case "FILE":
                lowLevelArtefact = this.FormEvent(events[i]);
                int needsSkipping = this.SkipFileEvents(events, i, periodInMinutes);

                if (!IsFileEventValid(lowLevelArtefacts, lowLevelArtefact))
                {
                    lowLevelArtefact = null;
                }

                i += needsSkipping;
                break;
            default:
                break;
        }

        if (lowLevelArtefact is not null)
        {
            lowLevelArtefacts.Add(lowLevelArtefact);
        }
    }

    return lowLevelArtefacts;
}
```

57 pav. Žemo lygio artifaktų formavimas naudojant *switch* sakinį

3.4.2. Abstrakcijos lygmenų formavimas naudojant *Chain of Responsibility* projektavimo šabloną

Aukšto lygio įvykių abstrakcijos lygmens formavimas naudojant *Chain of Responsibility* projektavimo šabloną pateiktas 58 pav. Laiko sudėtingumas liko toks pat – $O(n^2)$, nes funkcionalumas buvo iškeltas į atskiras klases. Atminties sudėtingumas taip pat liko $O(n)$. Akivaizdus šio šablono panaudojimo privalumas yra tas, kad metodo dydis ženkliais sumažėjo. Tai leidžia ne tik lengviau suprasti ir palaikyti kodą, bet ir nesunkiai pakeisti vieną metodologijos žingsnį nekeičiant kitų klasių.

```
public List<ISerializableLevel> FormHighLevelEvents(List<EventViewModel> events)
{
    List<ISerializableLevel> highLevelEvents = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        ISerializableLevel highLevelEvent = _handler.FormAbstractEvent(events, highLevelEvents, events[i]);

        if (highLevelEvent is not null)
        {
            highLevelEvents.Add(highLevelEvent);
        }
    }

    return highLevelEvents;
}
```

58 pav. Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* šabloną

Žemo lygio įvykių abstrakcijos lygmens formavimas su *Chain of Responsibility* šablonu yra pateiktas 59 pav. Laiko ir atminties sudėtingumas čia taip pat nepakito. Projektavimo šablono panaudojimas čia nesumažino metodo dydžio tiek, kiek formuojant aukšto lygio įvykius. Kita vertus, čia taip pat galima nesunkiai pakeisti kokį nors metodologijos žingsnį nepaveikiant kitų žingsnių. Viso funkcionalumo nepavyko iškelti į atskiras klases, nes šio lygmens formavimui neužtenka dirbti tik su vienu įvykiu.

```
public List<ISerializableLevel> FormLowLevelEvents(List<EventViewModel> events)
{
    List<ISerializableLevel> lowLevelEvents = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        if (events[i].Source == "OLECF")
        {
            while (i < events.Count - 1 && AreOlecfEventsOfSameTime(events[i], events[i + 1]))
            {
                i++;
            }
        }

        ISerializableLevel lowLevelEvent = _handler.FormAbstractEvent(events, lowLevelEvents, events[i]);

        if (events[i].Source == "FILE")
        {
            int validEventIndex = this.GetValidFileEventIndex(events, i);

            if (validEventIndex != -1 && !_lowLevelAbstractorUtils.DoesNeedComposing(events, i - 1, events[validEventIndex]))
            {
                int offset = validEventIndex - i;
                i += offset;

                bool isSearchedFormat = events[i].Format == "lnk/shell_items" || events[i].Format == "filestat";
                if (isSearchedFormat)
                {
                    DateTime date = events[i].FullDate;

                    while (i < events.Count - 1 && events[i].Source == "FILE" && events[i + 1].FullDate.CompareTo(date) == 0)
                    {
                        i++;
                    }
                }
            }
        }

        if (lowLevelEvent is not null)
        {
            lowLevelEvents.Add(lowLevelEvent);
            NormalizeEvents(lowLevelEvents, events, events[i]);
        }
    }

    return lowLevelEvents;
}
```

59 pav. Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* šabloną

Aukšto lygio artifaktų abstrakcijos lygmens formavimas panaudojant *Chain of Responsibility* projektavimo šabloną parodytas 60 pav. Laiko ir atminties sudėtingumas nepasikeitė. Projektavimo šablono panaudojimas padėjo gana ženkliai sumažinti metodo dydį, tačiau čia taip pat nepavyko iškelti viso funkcionalumo į atitinkamas klases.

```
public List<ISerializableLevel> FormHighLevelArtefacts(List<EventViewModel> events)
{
    List<ISerializableLevel> highLevelArtefacts = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        if (events[i].Source == "OLECF")
        {
            while (i < events.Count - 1 && AreOlecfEventsOfSameTime(events[i], events[i + 1]))
            {
                i++;
            }
        }

        ISerializableLevel highLevelArtefact = _handler.FormAbstractEvent(events, highLevelArtefacts, events[i]);

        if (events[i].Source == "FILE")
        {
            if (!_highLevelArtefactsAbstractorUtils.IsFileDuplicateOfLnk(events, i - 1, events[i]))
            {
                int fileCountInRowAtSameMinute = _highLevelArtefactsAbstractorUtils.GetFileCountInRowAtSameMinute(events, i);
                i += fileCountInRowAtSameMinute - 1;
            }
        }

        if (IsEventValid(highLevelArtefacts, (HighLevelArtefactViewModel)highLevelArtefact))
        {
            highLevelArtefacts.Add(highLevelArtefact);
        }
    }

    return highLevelArtefacts;
}
```

60 pav. Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* šabloną

Žemo lygio artifaktų abstrakcijos lygmens formavimas su *Chain of Responsibility* projektavimo šablonu yra parodytas 61 pav. Laiko ir atminties sudėtingumas nepakito. Projektavimo šablonas čia taip pat padėjo sumažinti metodo kodo eilučių skaičių. Šio lygmens formavime nepavyko iškelti tik su failo tipu susijusio funkcionalumo, skirto praleisti nurodytame laiko intervale pasikartojančius įvykius.

```
public List<ISerializableLevel> FormLowLevelArtefacts(List<EventViewModel> events, double periodInMinutes = 1.0)
{
    List<ISerializableLevel> lowLevelArtefacts = new(events.Count);

    for (int i = 0; i < events.Count; i++)
    {
        ISerializableLevel lowLevelArtefact = _handler.FormAbstractEvent(events, lowLevelArtefacts, events[i]);

        if (events[i].Source == "FILE")
        {
            int needsSkipping = this.SkipFileEvents(events, i, periodInMinutes);
            i += needsSkipping;
        }

        if (lowLevelArtefact is not null)
        {
            lowLevelArtefacts.Add(lowLevelArtefact);
        }
    }

    return lowLevelArtefacts;
}
```

61 pav. Žemo lygio artifaktų formavimas naudojant *Chain of Responsibility* šabloną

Chain of Responsibility projektavimo šablono įgyvendinimui buvo sukurta *IHandler* sąsaja (žr. 62 pav.). Ji turi *Next* savybę, kuri leidžia nurodyti kitą grandinėlės narį. Metodas *FormAbstractEvent* yra skirtas suformuoti reikiamą abstrakcijos lygį.

```
public interface IHandler
{
    99+ references
    IHandler Next { get; set; }

    78 references
    public ISerializableLevel FormAbstractEvent(List<EventViewModel> events, List<ISerializableLevel> abstractionLevel, EventViewModel currentEvent);
}
```

62 pav. *IHandler* sąsaja, naudojama *Chain of Responsibility* įgyvendinimui

Kiekvieno įvykio tipo abstrakcijos lygmens formavimo klasei papildomai yra sukuriama atskira tuščia sąsaja (žr. 63 pav.), kad būtų galima lengviau panaudoti priklausomybės inversiją. Kitu atveju reikėtų sukurti servisą, kuris grąžintų reikiamą tą pačią sąsają realizuojantį objektą pagal tipą.

```
public interface IHighLnkEventHandler : IHandler
{
}
```

63 pav. Tuščia sąsaja, paveldinti *IHandler* sąsają

4. Eksperimentinė dalis

Visual Studio aplinka gali apskaičiuoti keletą kodo metrikų: palaikomumo indeksą, ciklomatinį sudėtingumą, paveldėjimo medžio gylį, klasių susietumą bei kodo eilučių skaičių. Šių metrikų įvertinimai prieš patobulinimų įgyvendinimą yra pateikti 64 pav.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
EventTimelineReconstruction (Release)	81	1,461	9	258	9,382

64 pav. Kompiuterio įvykių laike sistemos metrikos prieš patobulinimus

Palaikomumo indeksas gali būti tarp 0 ir 100. *Visual Studio* įrankis rodo, kad sistemos palaikomumo indeksas yra 81, o tai reiškia, kad kodas turi gerą palaikomumo lygį. Ciklomatinis sudėtingumas yra 1461, kas reiškia, kad tiek yra nepriklausomų kodo vykdymo kelių. Šią metriką reikia stengtis minimizuoti, o tai padaryti galima mažinant loginių sąlygos sakinių skaičių. Didžiausias paveldėjimo medžio gylis yra 9. Kuo ši metrika yra didesnė, tuo sistemos palaikymas yra sudėtingesnis. Klasės susietumas parodo, kiek toje klasėje yra naudojama kitų konkrečių klasių. Galiausiai, sistemą sudaro 9382 kodo eilutės.

Greitaveikos tyrimai buvo atlikti ant .NET 6 karkaso naudojant *BenchmarkDotNet* paketą.

4.1. Įvykių importavimo iš failo eksperimentų rezultatai

Įvykių importavimo iš failo hipotezės:
1. Jeigu patobulinimas išnaudoja lygiagretumą, tai jis bus greitesnis nei kiti, nenaudojantys lygiagretumo.
2. Jeigu patobulinimo teorinis laiko sudėtingumas yra tiesinis, tai per gautus greitaveikos atskaitos taškus bus galima nubrėžti tiesę.

4.1.1. Paprastas *for* ciklas

Įgyvendinus 3.1.1 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 65 pav. parodytas metrikas.

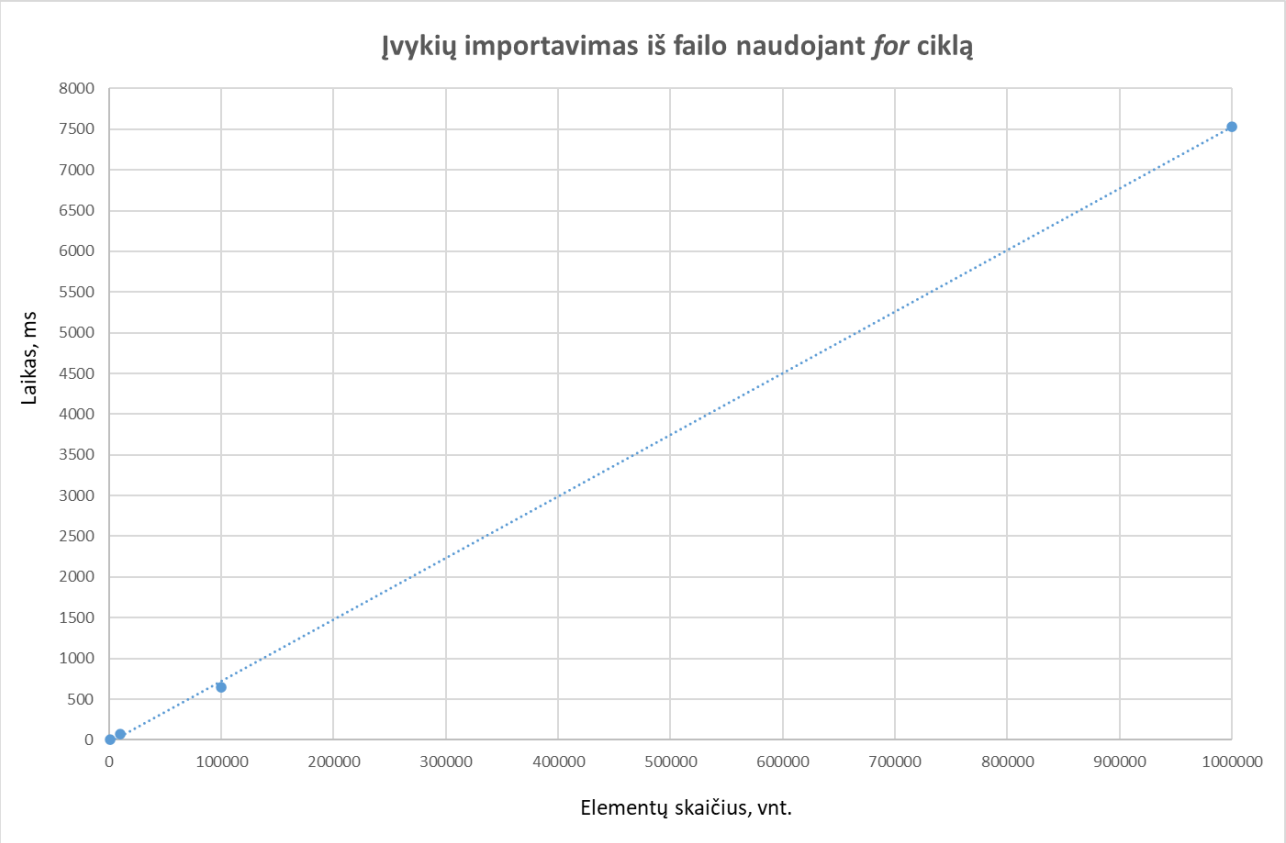
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	13	104

65 pav. Įvykių importavimo iš failo naudojant *for* ciklą kodo metrikos

Šio patobulinimo palaikomumo indeksas yra 61. Jis yra tarp vidutinio palaikomumo (tarp 10 ir 19) ir gero palaikomumo (tarp 20 ir 100). Ciklomatinis sudėtingumas yra 10. Jį būtų galima dar šiek tiek sumažinti gaudant vieną bendresnę išimtį, tačiau tai pažeistų švaraus kodo principą. Paveldėjimo medžio gylis yra 1, nes klasė įgyvendina tik vieną sąsają. Klasės susietumas yra 13. Jį taip pat būtų galima sumažinti gaudant vieną išimtį. Šį patobulinimą įgyvendinančios klasės dydis yra 104 kodo eilutės.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 66 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis

metodas yra trečias pagal greitumą dirbant su 1000 ir 10000 elementų, ketvirtas dirbant su 100000 elementų bei aštuntas dirbant su 1 milijonu įvykių.



66 pav. Įvykių importavimo iš failo naudojant *for* ciklą greitaveika

4.1.2. Nesaugus *for* ciklas naudojant *Span* iš *List*

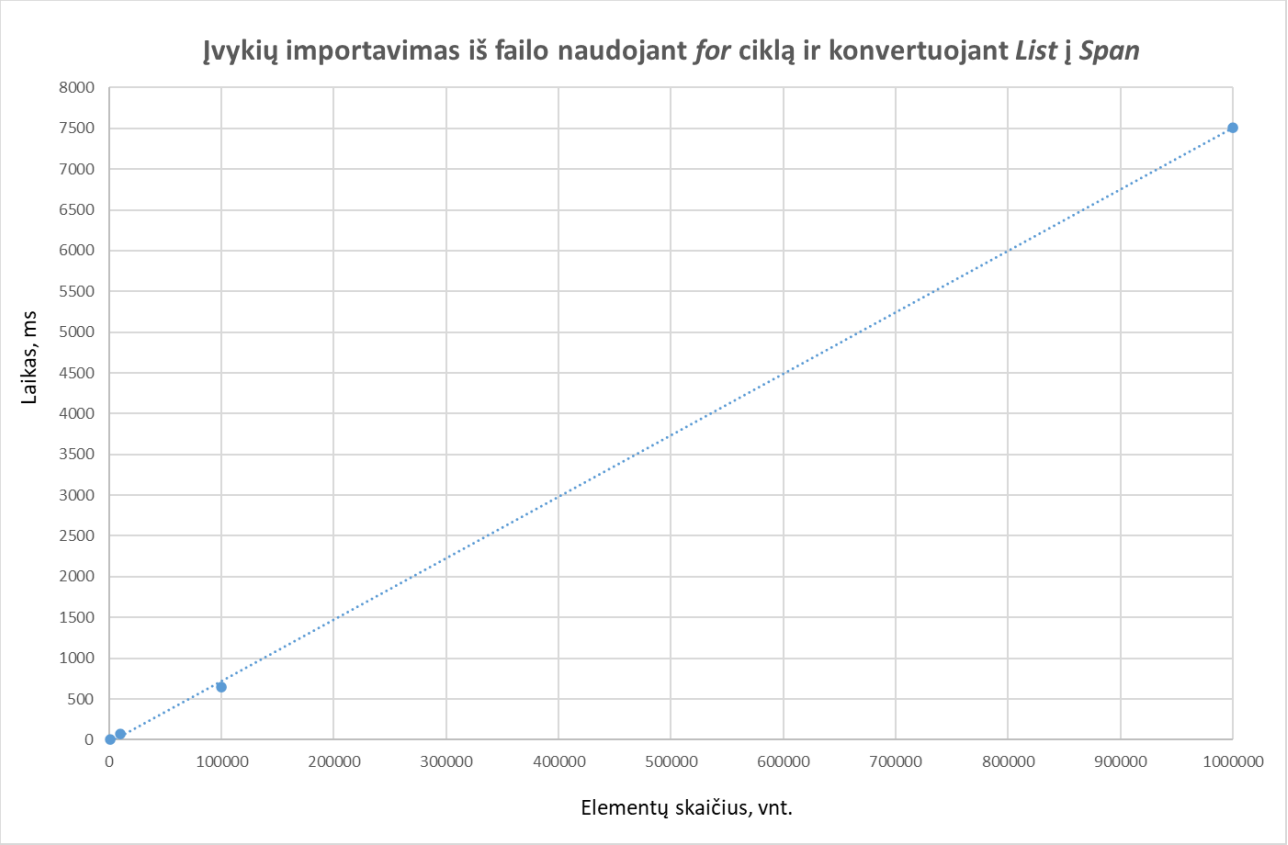
Įgyvendinus 3.1.2 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 67 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	15	105

67 pav. Įvykių importavimo iš failo naudojant *for* ciklą ir konvertuojant *List* į *Span* kodo metrikos

Šio patobulinimo palaikomumo indeksas yra 61, ciklomatinis sudėtingumas – 10, o paveldėjimo medžio gylis yra 1. Šios metrikos yra tokios pačios kaip ir paprasto *for* ciklo patobulinimo. Klasės susietumas yra 15, šį patobulinimą įgyvendinančios klasės dydis yra 105 kodo eilutės. Šios metrikos yra šiek tiek blogesnės nei *for* ciklo patobulinimo.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 68 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, ketvirtas dirbant su 10000 ir 100000 elementų bei aštuntas dirbant su 1 milijonu įvykių.



68 pav. Įvykių importavimo iš failo naudojant *for* ciklą ir konvertuojant *List* į *Span* greitaveika

4.1.3. Nesaugus *for* ciklas naudojant *Span* iš masyvo

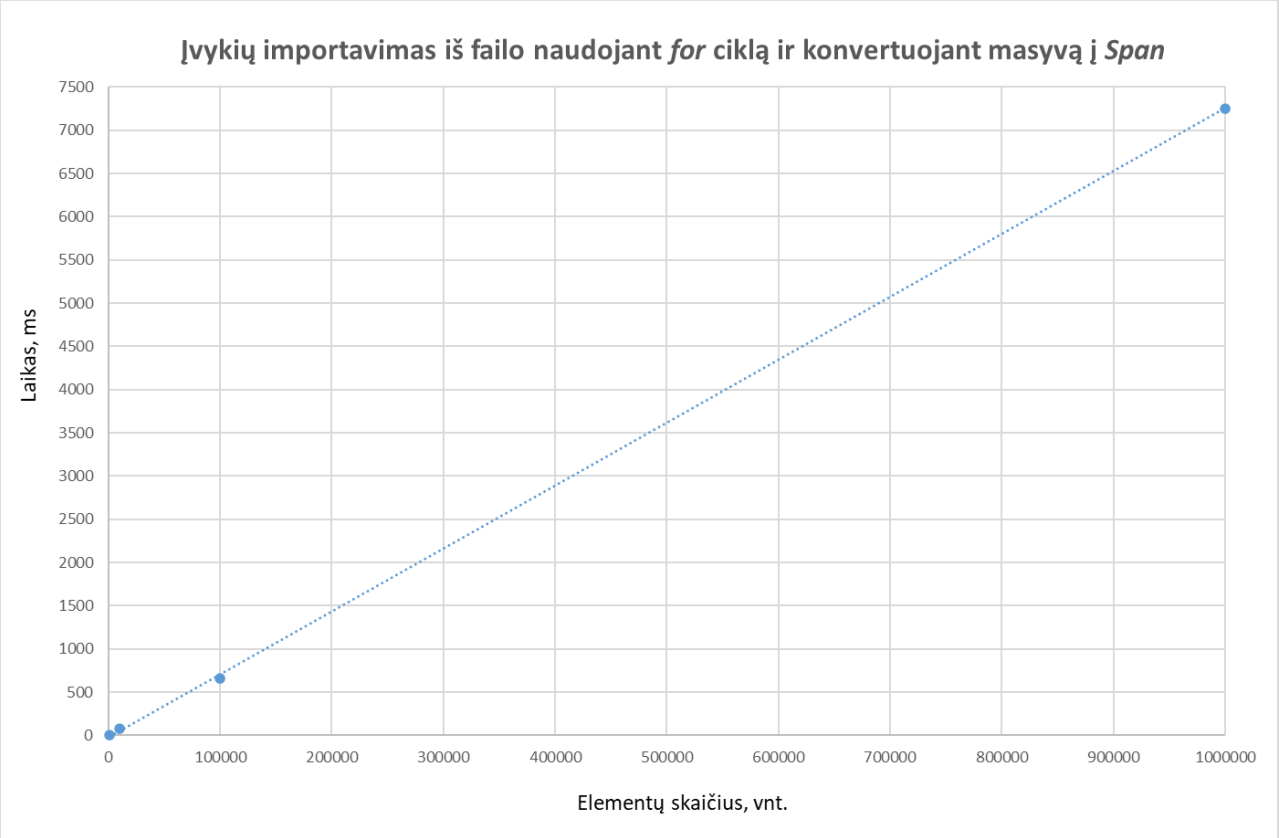
Įgyvendinus 3.1.3 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 69 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	15	105

69 pav. Įvykių importavimo iš failo naudojant *for* ciklą ir konvertuojant masyvą į *Span* kodo metrikos

Šio patobulinimo kodo metrikos yra tokios pačios, kaip ir naudojant *for* ciklą su *List* konvertavimu į *Span*. Taip yra dėl to, kad skiriasi tik *Span* objekto gavimo metodas.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 70 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, ketvirtas dirbant su 10000 ir 100000 elementų bei septintas dirbant su 1 milijonu įvykių.



70 pav. Įvykių importavimo iš failo naudojant *for* ciklą ir konvertuojant masyvą į *Span* greitaveika

4.1.4. Nesaugus *for* ciklas naudojant *MemoryMarshal*

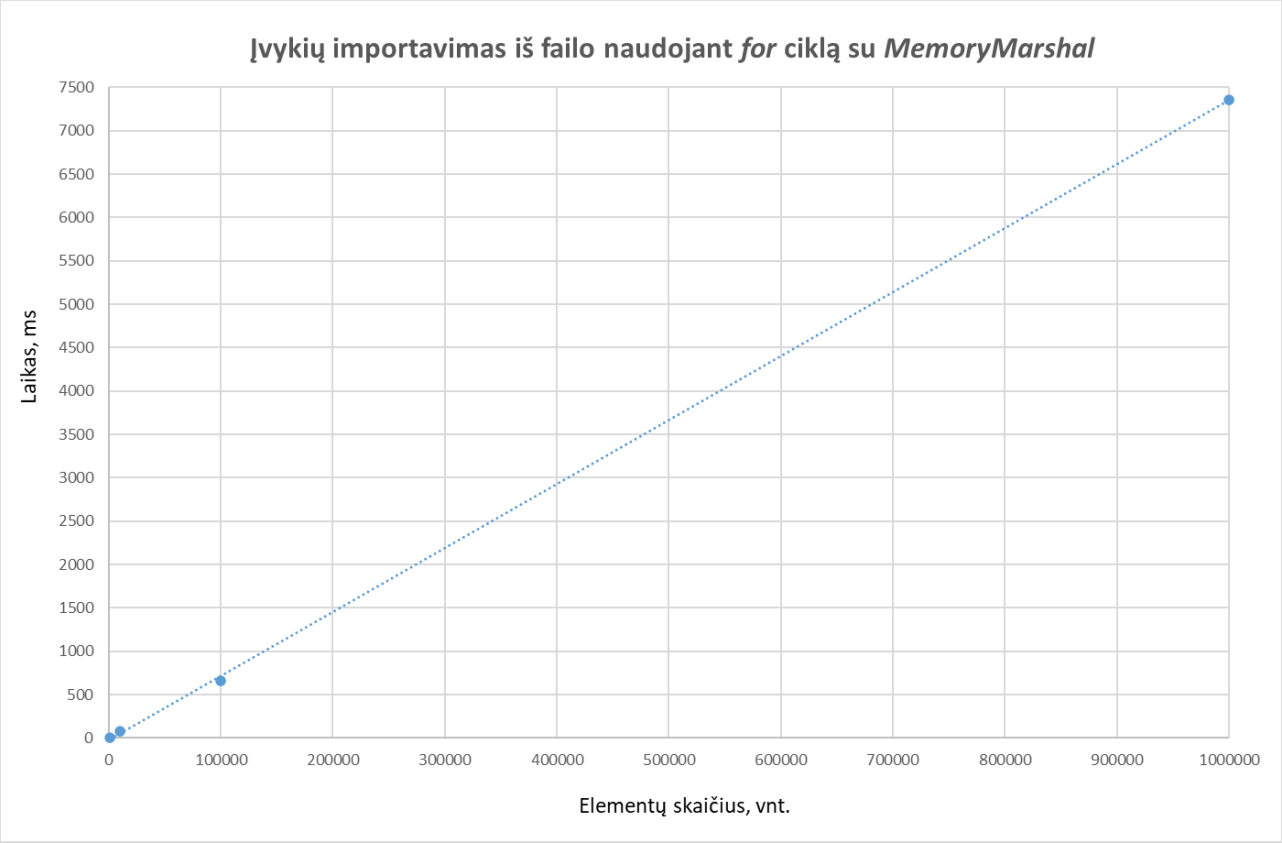
Įgyvendinus 3.1.4 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 71 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	15	106

71 pav. Įvykių importavimo iš failo naudojant *for* ciklą su *MemoryMarshal* kodo metrikos

Šio patobulinimo kodo metrikos yra beveik tokios pačios, kaip ir naudojant *for* ciklą su *List* ar masyvo konvertavimu į *Span*. Skirtumas yra tas, kad šis patobulinimas yra viena kodo eilute didesnis.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 72 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, penktas dirbant su 10000 elementų, ketvirtas dirbant su 100000 elementų bei septintas dirbant su 1 milijonu įvykių.



72 pav. Įvykių importavimo iš failo naudojant *for* ciklą su *MemoryMarshal* greitaveika

4.1.5. Lygiagretus *for* ciklas

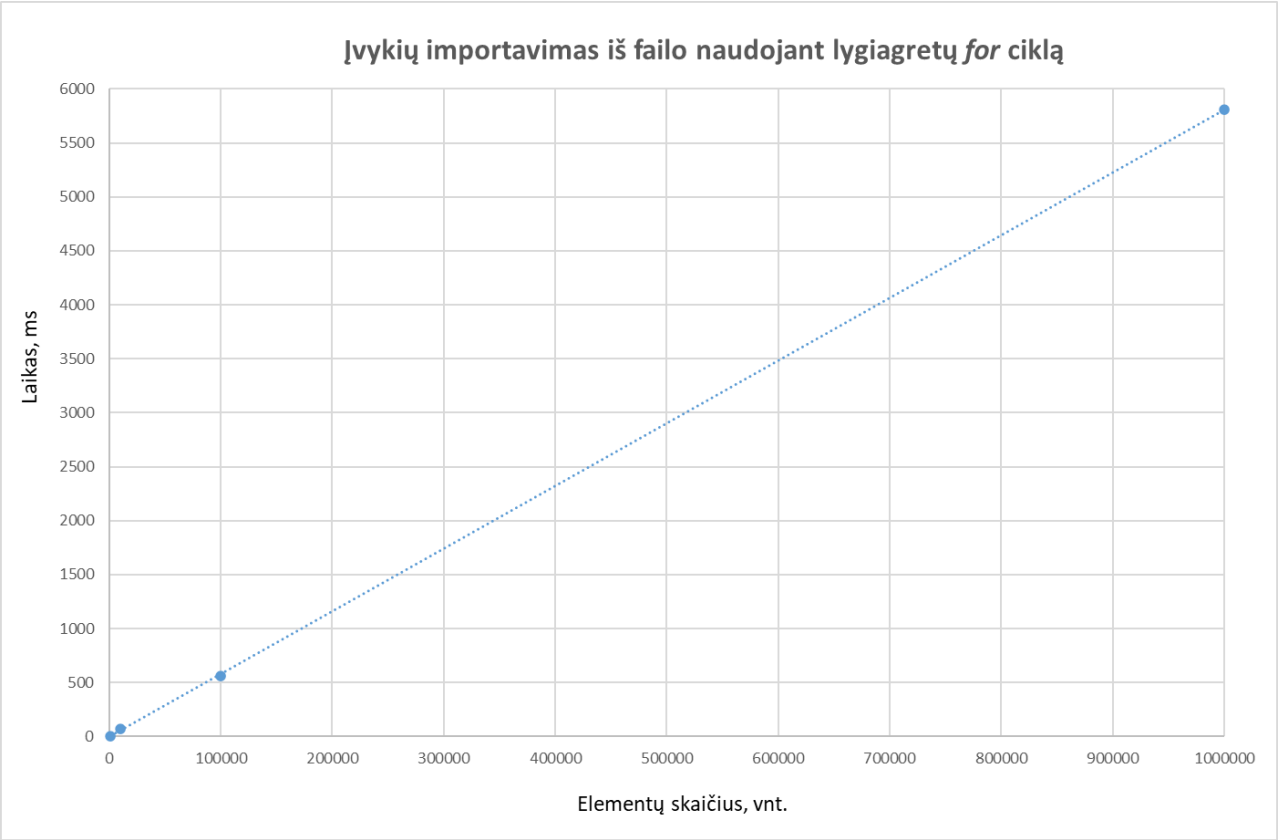
Įgyvendinus 3.1.5 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 73 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	9	1	16	108

73 pav. Įvykių importavimo iš failo naudojant lygiagretų *for* ciklą kodo metrikos

Beveik visos šio patobulinimo kodo metrikos yra šiek tiek prastesnės nei ankstesnių patobulinimų. Kita vertus, ciklomatinis sudėtingumas yra šiek tiek geresnis, o tai reiškia lengvesnį palaikymą.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 74 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra greičiausias dirbant su 1000, 100000 ir 1 milijonu elementų bei ketvirtas dirbant su 10000 įvykių.



74 pav. Įvykių importavimo iš failo naudojant lygiagretų *for* ciklą greitaveika

4.1.6. Paprastas *foreach* ciklas

Šiuo metu sistemoje įgyvendintam įvykių importavimui iš failo *Visual Studio* pateikė 75 pav. parodytas metrikas.

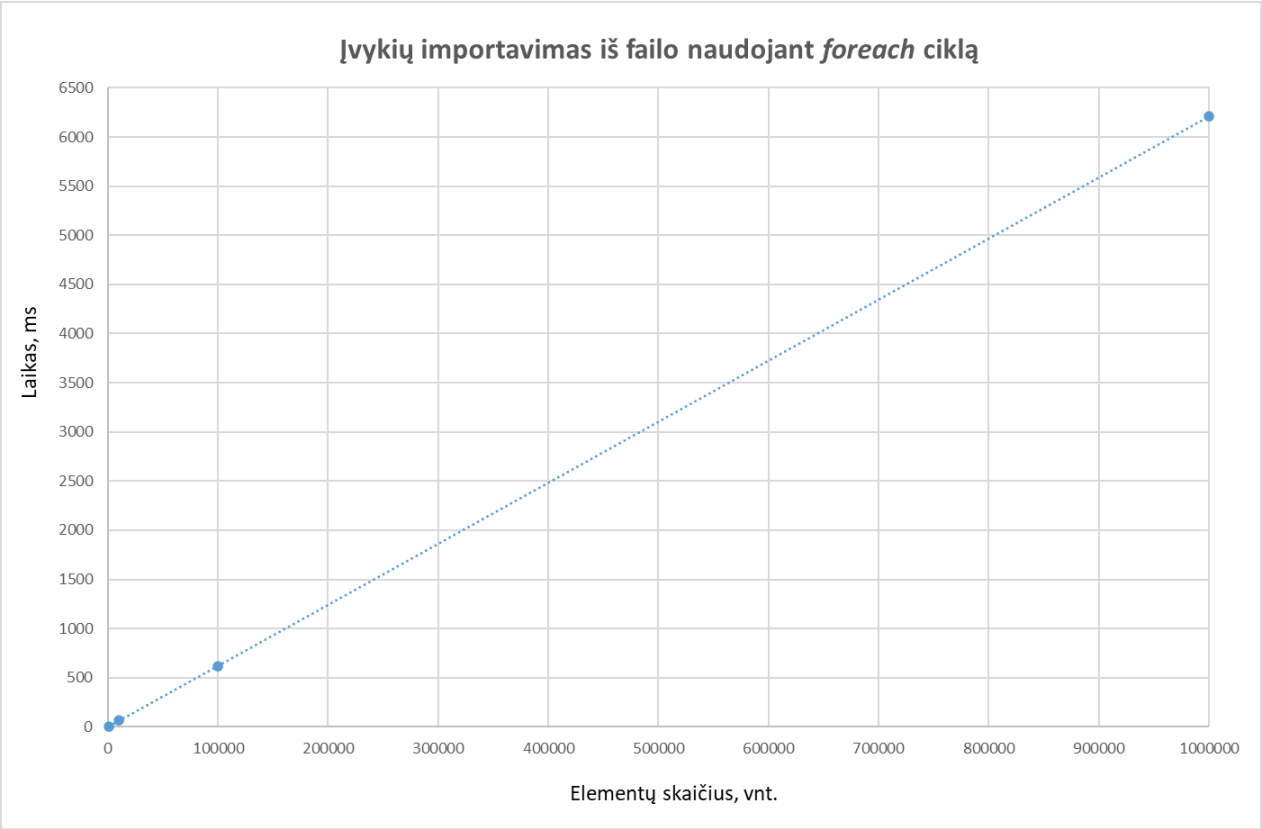
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	13	110

75 pav. Įvykių importavimo iš failo naudojant *foreach* ciklą kodo metrikos

Šio būdo kodo metrikos yra beveik tokios pačios, kaip ir naudojant paprastą *for* ciklą. Vienintelis skirtumas tarp jų yra tas, kad šis patobulinimas užima šiek tiek daugiau kodo eilučių.

Atlikus greitaveikos testavimą, buvo sudarytas 76 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų bei antras dirbant su 10000, 100000 ir 1 milijonu elementų.

Taip pat buvo išbandyta inicializuoti sąrašo dydį, bet tai nedavė jokios naudos. Kaip ir buvo tikėtasi, greitaveika sulėtėjo, o paskirtos atminties dydis padidėjo.



76 pav. Įvykių importavimo iš failo naudojant *foreach* ciklą greitaveika

4.1.7. Paprastas *foreach* ciklas naudojant LINQ

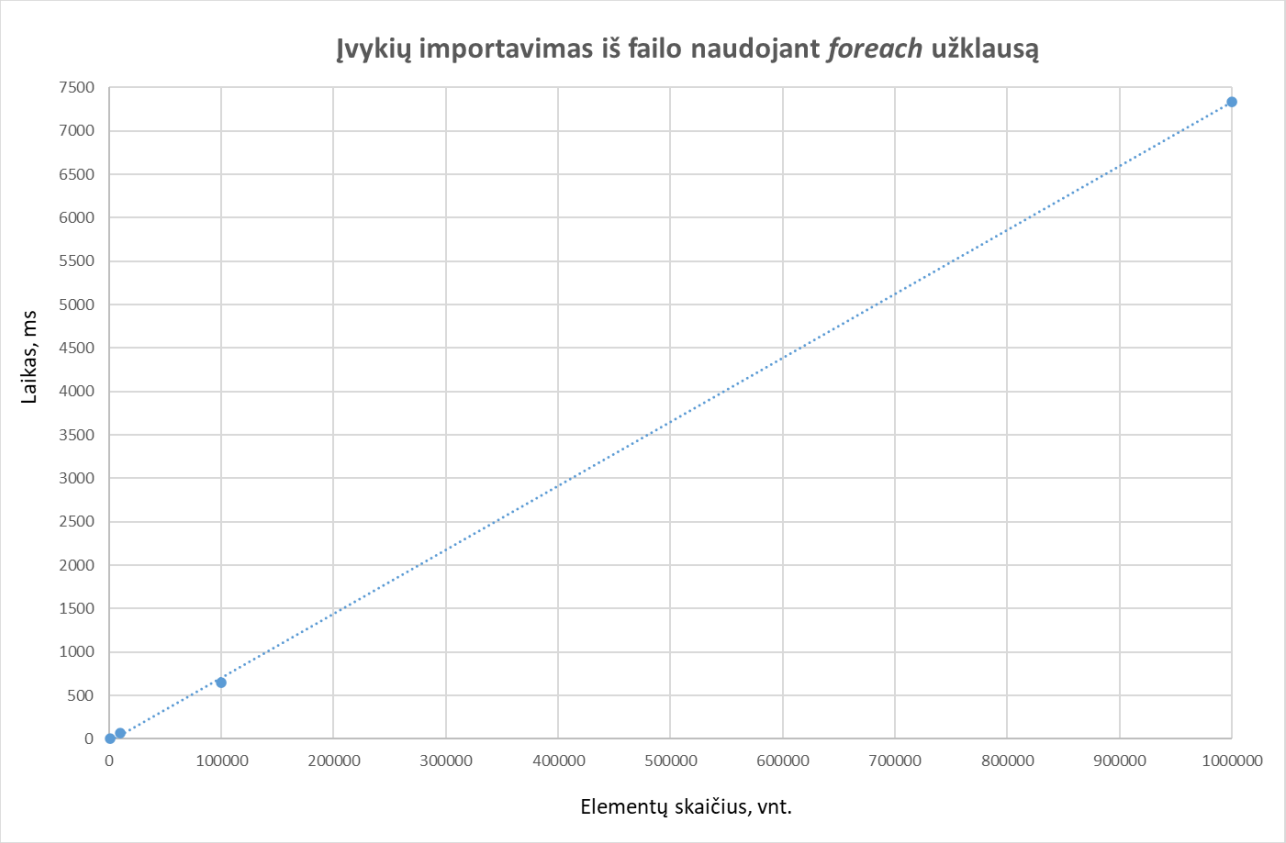
Įgyvendinus 3.1.7 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 77 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	9	1	14	110

77 pav. Įvykių importavimo iš failo naudojant *foreach* užklausą kodo metrikos

Lygindamas su ankstesniais patobulinimais, šis būdas turi geresnį ciklomatinį sudėtingumą nei dauguma kitų patobulinimų, tačiau kitose metrikose niekuo neišsiskiria.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 78 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, ketvirtas dirbant su 10000 ir 100000 elementų bei septintas dirbant su 1 milijonu elementų.



78 pav. Įvykių importavimo iš failo naudojant *foreach* užklausą greitaveika

4.1.8. Nesaugus *foreach* ciklas naudojant *Span* iš *List*

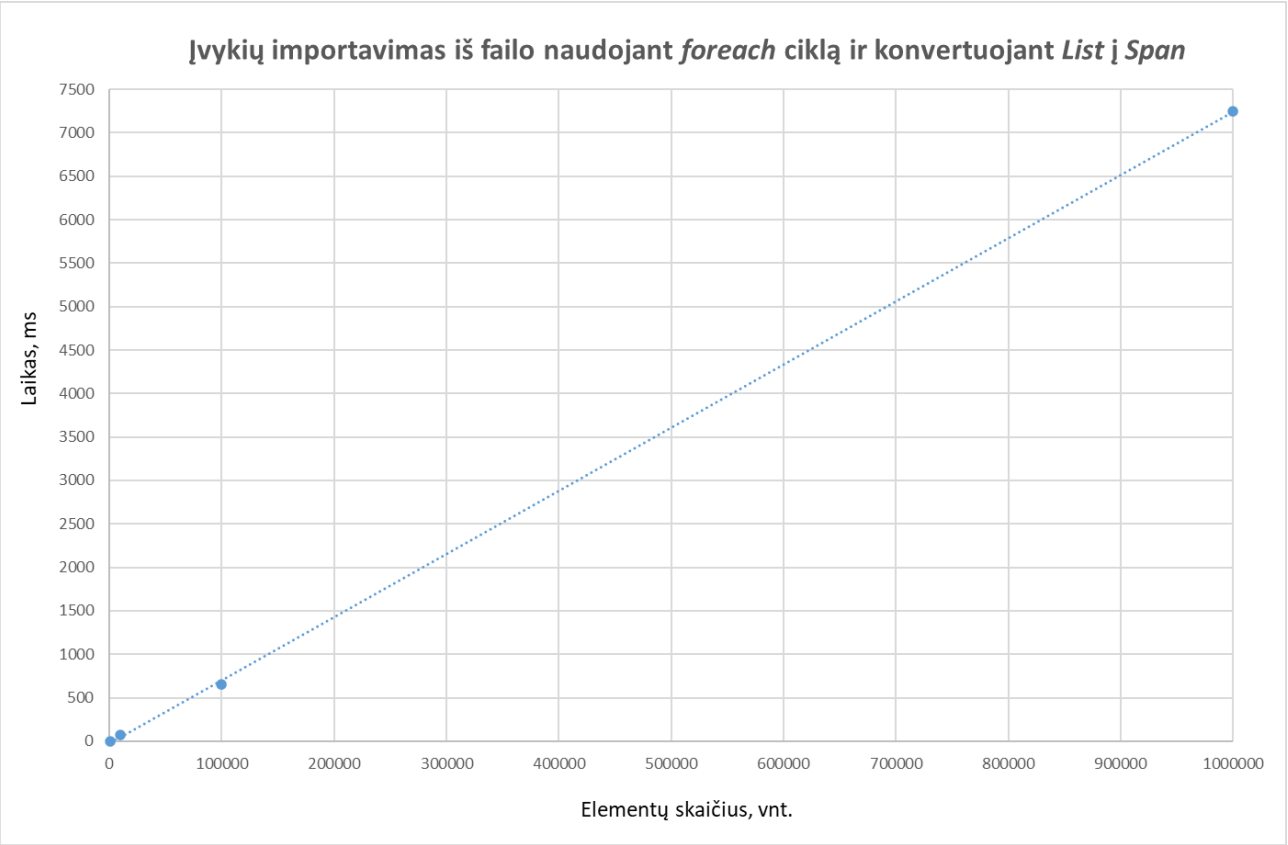
Įgyvendinus 3.1.8 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 79 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	15	110

79 pav. Įvykių importavimo iš failo naudojant *foreach* ciklą ir konvertuojant *List* į *Span* kodo metrikos

Šis patobulinimas yra panašus į *for* ciklo su *Span* patobulinimus, tačiau eilučių skaičiumi šis būdas yra didesnis. Kitos metrikos yra tokios pačios, nes skiriasi tik ciklo iteravimo būdas.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 80 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, ketvirtas dirbant su 10000 ir 100000 elementų bei septintas dirbant su 1 milijonu elementų.



80 pav. Įvykių importavimo iš failo naudojant *foreach* ciklą ir konvertuojant *List* į *Span* greitaveika

4.1.9. Nesaugus *foreach* ciklas naudojant *Span* iš masyvo

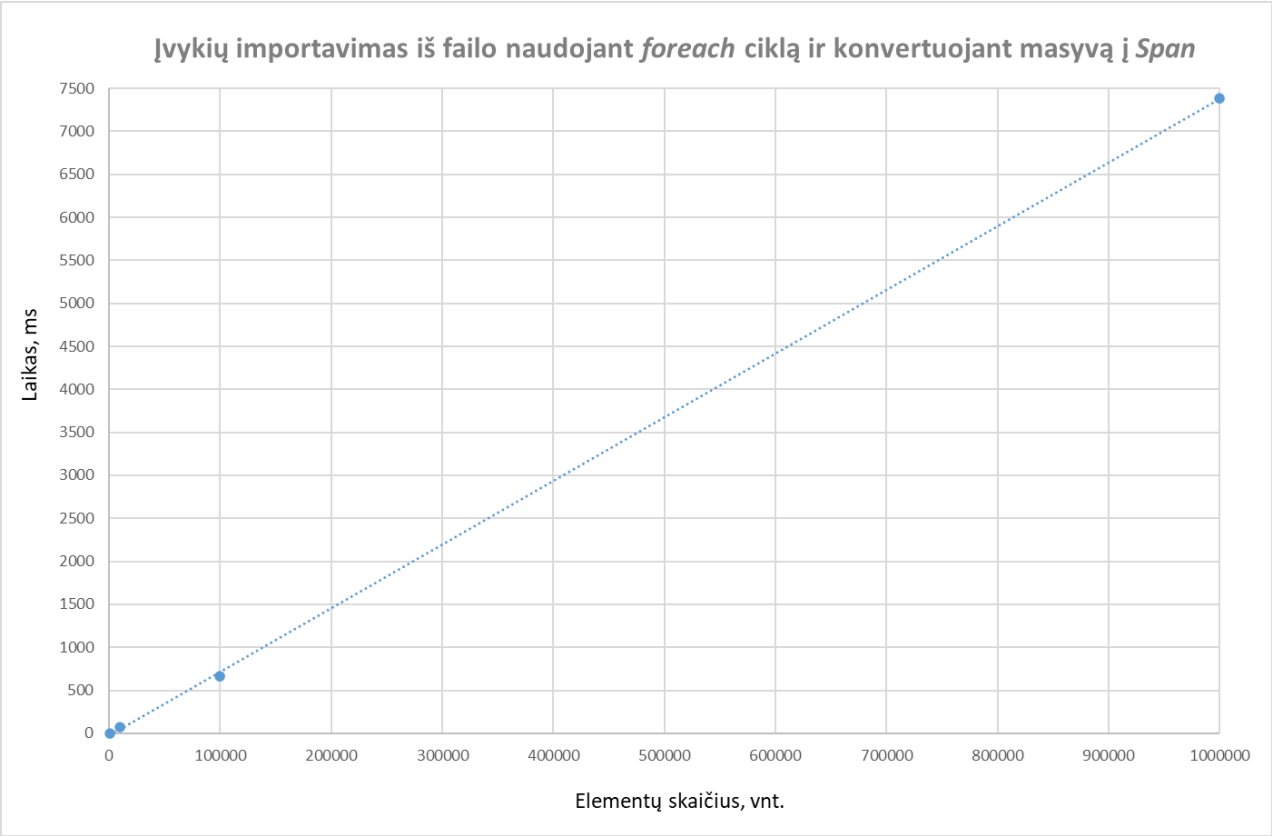
Įgyvendinus 3.1.9 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 81 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	10	1	15	110

81 pav. Įvykių importavimo iš failo naudojant *foreach* ciklą ir konvertuojant masyvą į *Span* kodo metrikos

Šio patobulinimo metrikos yra tokios pačios lyginant su *foreach* ciklu, kuriame yra naudojamas *Span* iš *List*. Taip yra dėl to, kad praktiškai skiriasi tik *Span* objekto paėmimas.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 82 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, ketvirtas dirbant su 10000 ir 100000 elementų bei septintas dirbant su 1 milijonu elementų.



82 pav. Įvykių importavimo iš failo naudojant *foreach* ciklą ir konvertuojant masyvą į *Span* greitaveika

4.1.10. Lygiagretus *foreach* ciklas

Įgyvendinus 3.1.10 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 83 pav. parodytas metrikas.

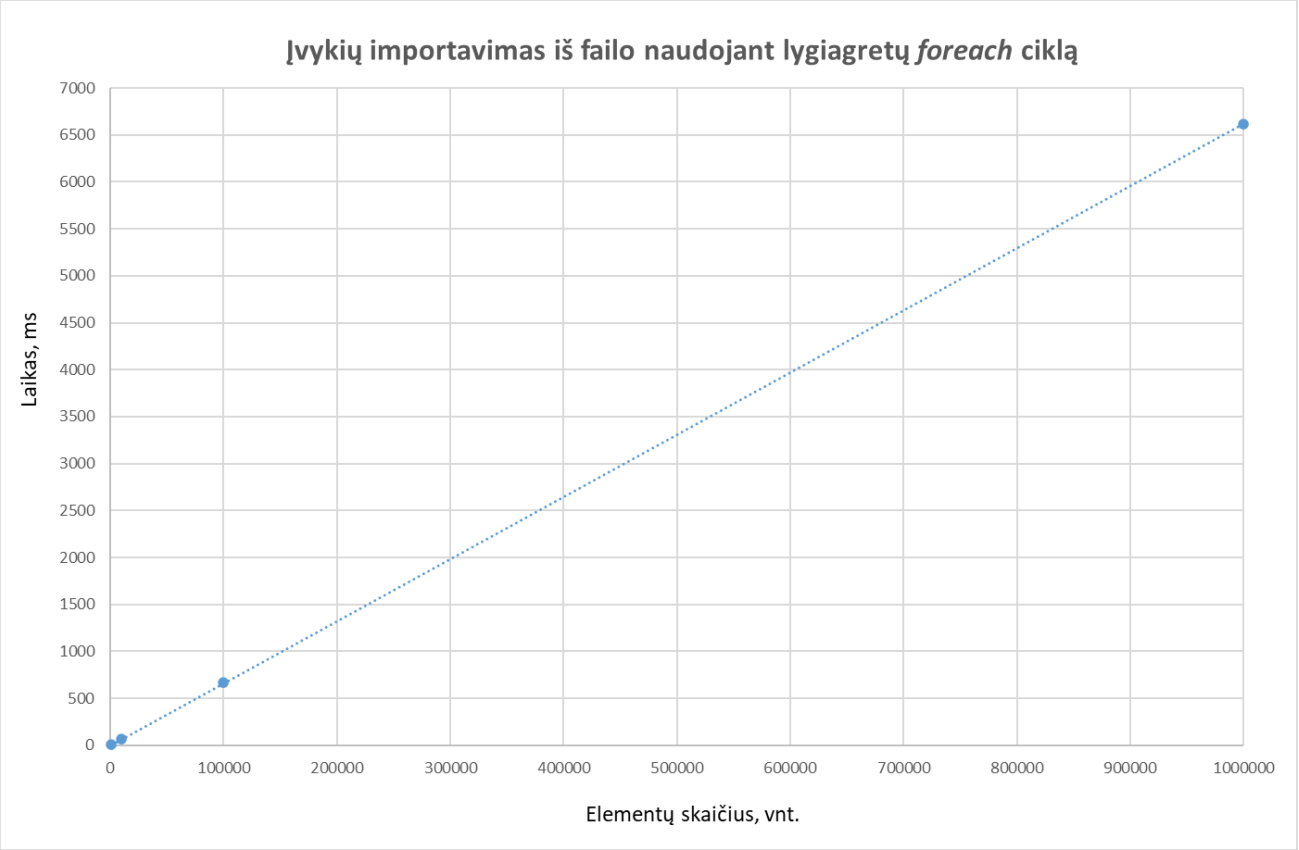
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	9	1	17	108

83 pav. Įvykių importavimo iš failo naudojant lygiagretų *foreach* ciklą kodo metrikos

Šis patobulinimas turi geriausią ciklomatinį sudėtingumą kartu su lygiagrečiu *for* ciklu ir *foreach* LINQ užklausa, tačiau klasių susietumas yra blogiausias lyginant su ankstesniais patobulinimais. Palaikomumo indeksas ir paveldėjimo medžio gylis yra tokie patys kaip ir kitų patobulinimų.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 84 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra šeštas pagal greitumą dirbant su 1000 elementų, trečias dirbant su 10000 elementų, ketvirtas dirbant su 100000 elementų bei penktas dirbant su 1 milijonu elementų.

Taip pat buvo išbandyta inicializuoti sąrašo dydį, bet tai nedavė jokios naudos. Kaip ir buvo tikėtasi, greitaveika sulėtėjo, o paskirtos atminties dydis padidėjo.



84 pav. Įvykių importavimo iš failo naudojant lygiagretų *foreach* ciklą greitaveika

4.1.11. Lygiagretus asinchroninis *foreach* ciklas

Įgyvendinus 3.1.11 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 85 pav. parodytas metrikas.

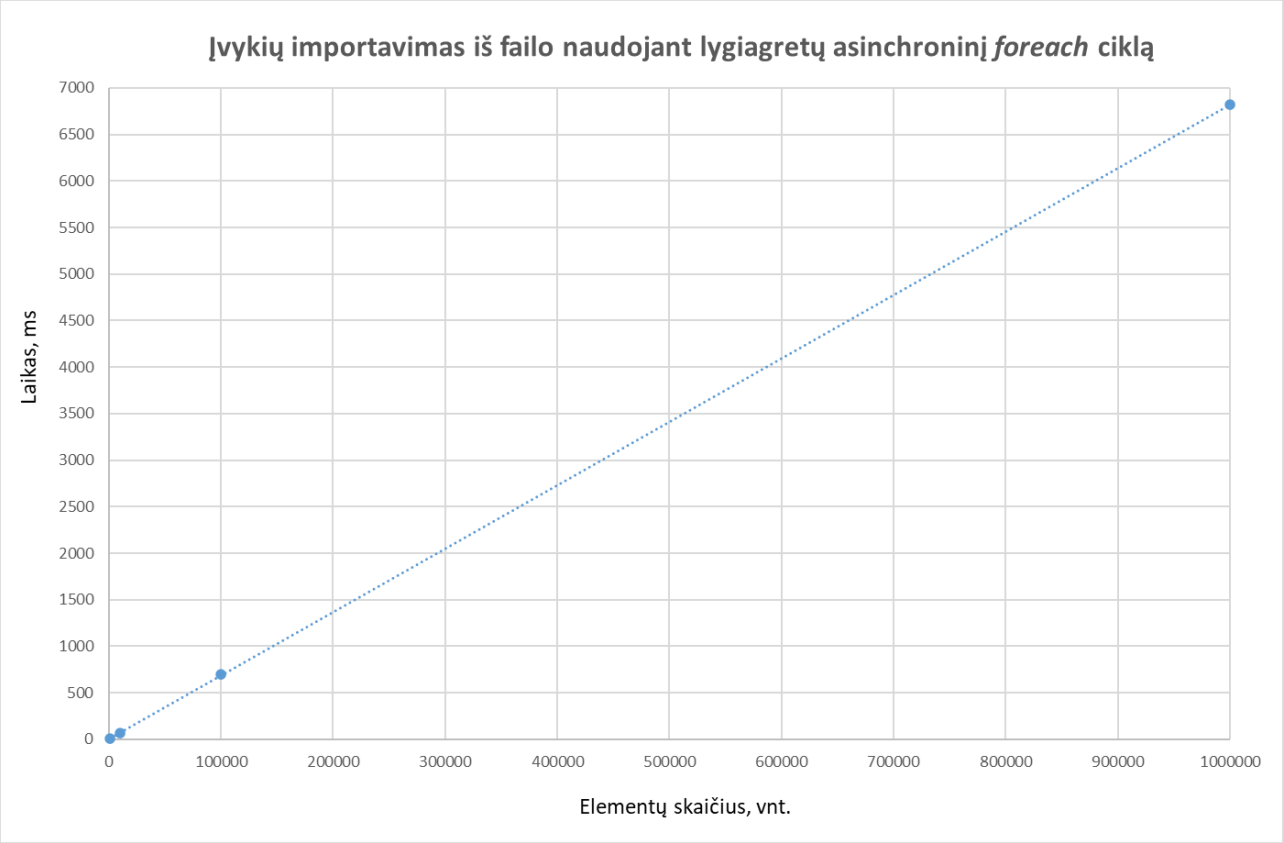
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	9	1	20	111

85 pav. Įvykių importavimo iš failo naudojant lygiagretų asinchroninį *foreach* ciklą kodo metrikos

Šis patobulinimas išsiskiria blogiausiu klasių susietumu iš visų patobulinimų. Jis taip pat yra didžiausias. Kita vertus, ciklomatinis sudėtingumas yra vienas geriausių.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 86 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra ketvirtas pagal greitumą dirbant su 1000 ir 10000 elementų, penktas dirbant su 100000 elementų bei šeštas dirbant su 1 milijonu elementų.

Taip pat buvo išbandyta inicializuoti sąrašo dydį, bet tai nedavė jokios naudos. Kaip ir buvo tikėtasi, greitaveika sulėtėjo, o paskirtos atminties dydis padidėjo.



86 pav. Įvykių importavimo iš failo naudojant lygiagretų asinchroninį *foreach* ciklą greitaveika

4.1.12. Paprastas *while* ciklas

Įgyvendinus 3.1.12 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 87 pav. parodytas metrikas.

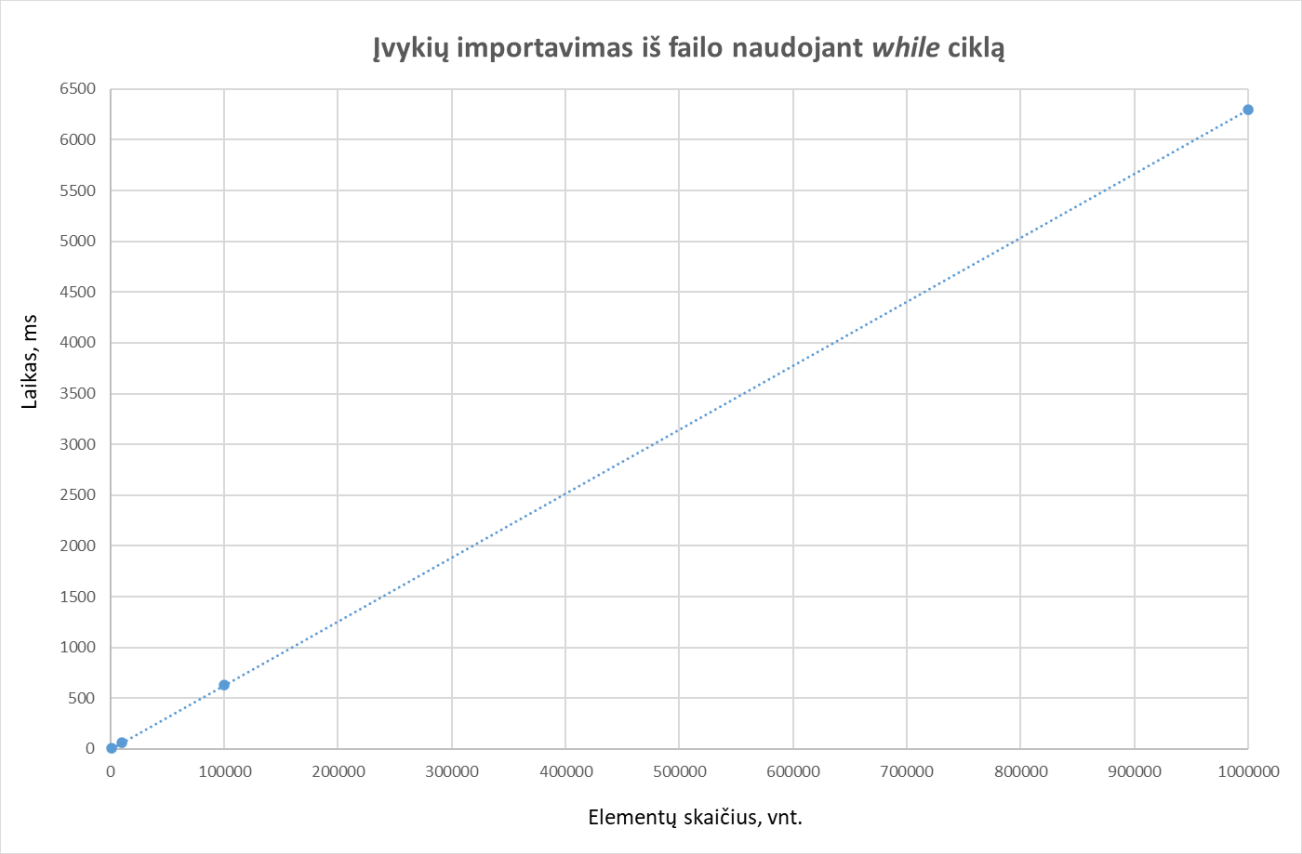
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	60	10	1	14	111

87 pav. Įvykių importavimo iš failo naudojant *while* ciklą kodo metrikos

Šis patobulinimas turi mažesnį palaikomumo indeksą nei kiti patobulinimai. Ciklomatinis sudėtingumas yra panašus į daugelio kitų patobulinimų. Kita vertus, klasių susietumas yra vienas iš mažiausių – 14.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 88 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000, 100000 ir 1 milijonu elementų bei greičiausias dirbant su 10000 įvykių.

Taip pat buvo išbandyta inicializuoti sąrašo dydį, bet tai nedavė jokios naudos. Kaip ir buvo tikėtasi, greitaveika sulėtėjo, o paskirtos atminties dydis padidėjo.



88 pav. Įvykių importavimo iš failo naudojant *while* ciklą greitaveika

4.1.13. Nesaugus *while* ciklas naudojant *MemoryMarshal*

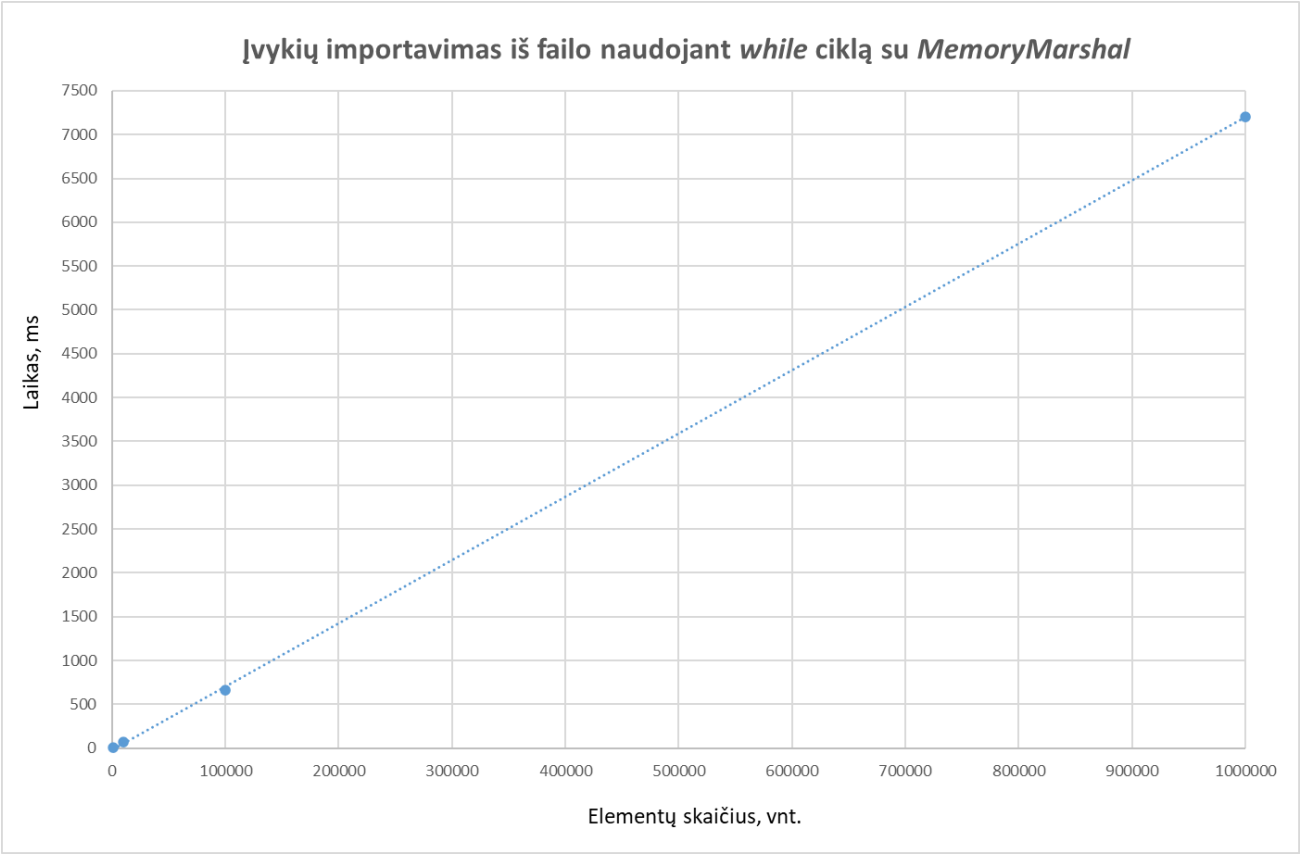
Įgyvendinus 3.1.13 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 89 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	60	10	1	15	114

89 pav. Įvykių importavimo iš failo naudojant *while* ciklą su *MemoryMarshal* kodo metrikos

Šis patobulinimas taip pat turi blogiausią palaikomumo indeksą. Kitos metrikos taip pat niekuo neišsiskiria iš kitų patobulinimų. Šis patobulinimas užima daugiausiai kodo eilučių.

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 90 pav. pateiktas grafikas. Grafike pavaizdutoa, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra trečias pagal greitumą dirbant su 1000 elementų, ketvirtas dirbant su 10000 ir 100000 elementų bei septintas dirbant su 1 milijonu įvykių.



90 pav. Įvykių importavimo iš failo naudojant *while* ciklą su *MemoryMarshal* greitaveika

4.1.14. Lygiagrečios užklausos

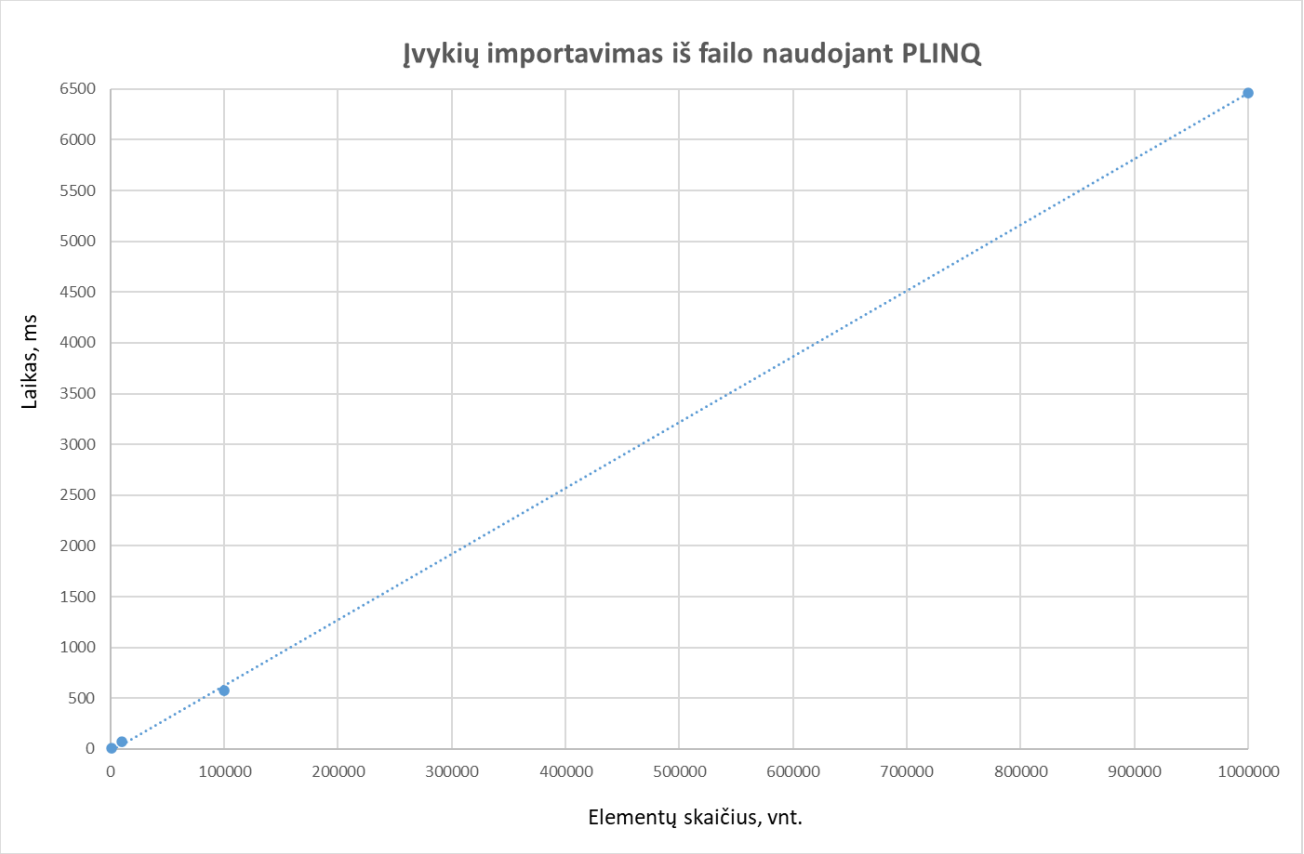
Įgyvendinus 3.1.14 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 91 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
L2tCSVEventsImporter	61	9	1	16	108

91 pav. Įvykių importavimo iš failo naudojant PLINQ kodo metrikos

Šio patobulinimo palaikomumo indeksas yra 61. Jis yra toks pat lyginant su dauguma kitų patobulinimų. Ciklomatinis sudėtingumas yra vienas iš mažiausių. Klasių susietumas yra 16, o tai yra beveik blogiausias klasių susietumo įvertinimas iš šių patobulinimų.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 92 pav. pateiktas grafikas. Grafike pavaizdutoa, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra antras pagal greitumą dirbant su 1000 elementų, trečias dirbant su 10000 elementų, greičiausias dirbant su 100000 elementų bei ketvirtas dirbant su 1 milijonu įvykių.



92 pav. Įvykių importavimo iš failo naudojant PLINQ greitaveika

4.1.15. Rezultatų apibendrinimas

Visų patobulinimų kodo metrikų įvertinimai yra pateikti 1 priede, o greitaveikos tyrimo rezultatai – 2 priede.

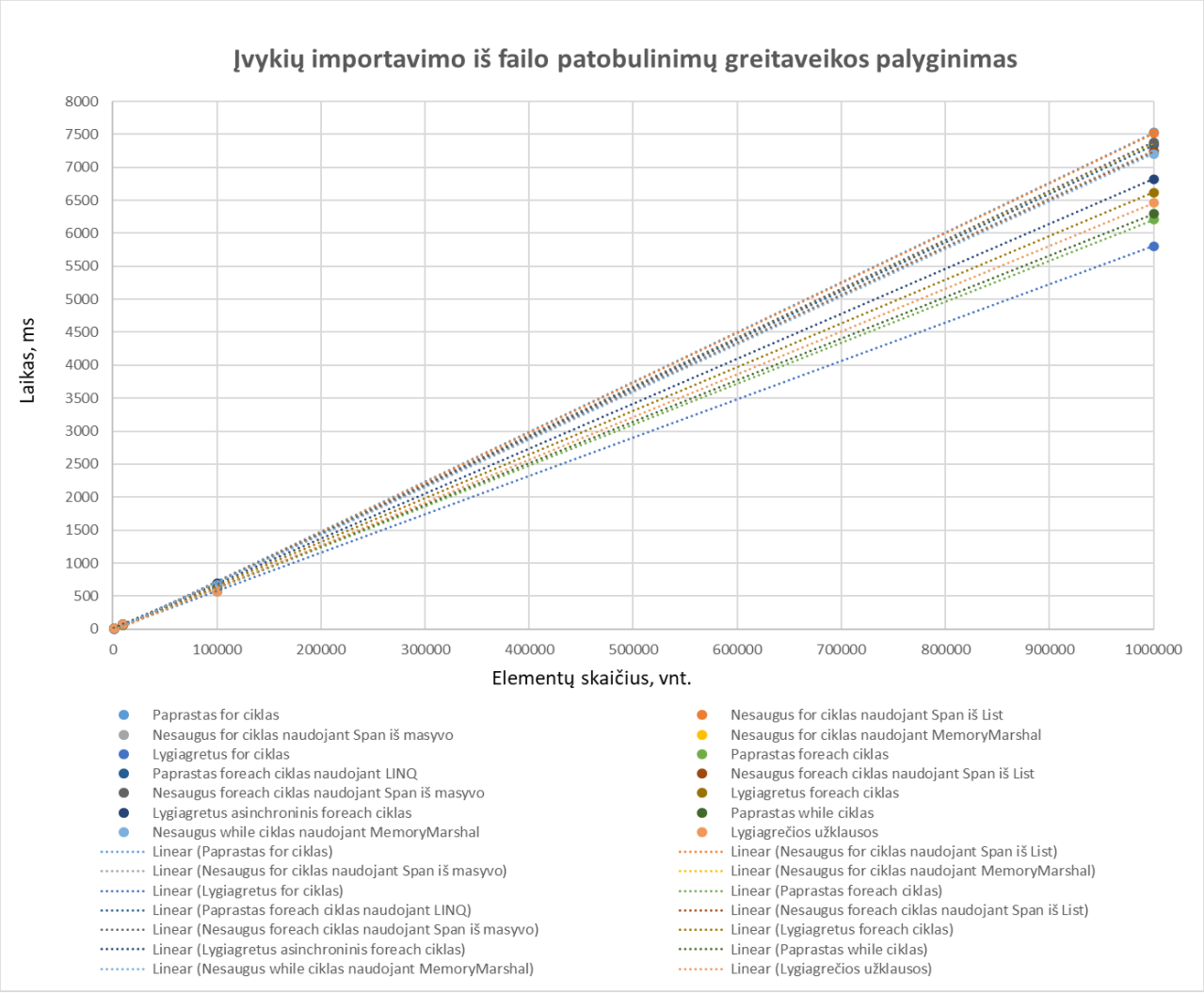
Geriausi įvykių importavimo iš failo patobulinimai pagal kiekvieną kodo metriką yra pateikiami 4 lentelėje. Nėra tokio patobulinimo, kuris būtų geriausias visose kodo metrikose. Paprastas *for* ciklas yra geriausias pagal keturias metrikas iš penkių. Jis nėra geriausias tik pagal ciklomatinį sudėtingumą, tačiau tas skirtumas yra nežymus. Kodo eilučių metrika čia taip pat neturėtų daryti įtakos, nes skirtumas tarp mažiausio ir didžiausio patobulinimų yra tik 10 kodo eilučių. Ignoruojant kodo eilučių metriką, galima išskirti septynis geriausius patobulinimus: paprastas *for* ciklas, lygiagretus *for* ciklas, paprastas *foreach* ciklas, paprastas *foreach* ciklas naudojant LINQ, lygiagretus *foreach* ciklas, lygiagretus asinchroninis *foreach* ciklas ir lygiagrečios užklausos.

4 lentelė. Geriausi įvykių importavimo iš failo patobulinimai pagal kodo metrikas

Kodo metrika	**Geriausi patobulinimai**
Palaikomumo indeksas	Paprastas *for* ciklas. Nesaugus *for* ciklas naudojant *Span* iš *List*. Nesaugus *for* ciklas naudojant *Span* iš masyvo. Nesaugus *for* ciklas naudojant *MemoryMarshal*. Lygiagretus *for* ciklas. Paprastas *foreach* ciklas. Paprastas *foreach* ciklas naudojant LINQ.

	Nesaugus *foreach* ciklas naudojant *Span* iš *List*. Nesaugus *foreach* ciklas naudojant *Span* iš masyvo. Lygiagretus *foreach* ciklas. Lygiagretus asinchroninis *foreach* ciklas. Lygiagrečios užklausos.
Ciklomatinis sudėtingumas	Lygiagretus *for* ciklas. Paprastas *foreach* ciklas naudojant LINQ. Lygiagretus *foreach* ciklas. Lygiagretus asinchroninis *foreach* ciklas. Lygiagrečios užklausos.
Paveldėjimo medžio gylis	Visi vienodi.
Klasių susietumas	Paprastas *for* ciklas. Paprastas *foreach* ciklas.
Kodo eilučių skaičius	Paprastas *for* ciklas.

Pagal greitaveiką (žr. 93 pav.), greičiausias patobulinimas yra lygiagretus *for* ciklas. Šiek tiek lėtesni yra paprasto *foreach* ciklo, paprasto *while* ciklo, lygiagrečių užklausų, lygiagretaus *foreach* ciklo bei lygiagretaus asinchroninio *foreach* ciklo patobulinimai. Likę patobulinimai yra dar lėtesni.



93 pav. Įvykių importavimo iš failo patobulinimų greitaveikos palyginimas

Įvertinus kodo metrikas ir greitaveikos tyrimą, geriausias įvykių importavimo iš failo metodas yra lygiagretus *for* ciklas. Kita vertus, lygiagretumas gali neleisti panaudoti asinchroniškumo, kad nebūtų užblokuota pagrindinė naudotojo sąsajos gija. Tokiu atveju, geriausias patobulinimas būtų paprastas *foreach* ciklas, kuris ir buvo įgyvendintas prieš kompiuterio įvykių laike atkūrimo sistemos tyrimą.

Galiausiai, pirma hipotezė yra tenkinama iš dalies. Lygiagretus *for* ciklas yra greitesnis už kitus patobulinimus, tačiau kiti lygiagretumą naudojantys patobulinimai yra toje pačioje grupėje kaip ir paprasti *for* ar *while* ciklai. Kita vertus, antra hipotezė yra teisinga. Per atitinkamo patobulinimo atskaitos taškus galima nubrėžti tiesę.

4.2. Darbo išsaugojimo eksperimentų rezultatai

Darbo išsaugojimo hipotezės:

1. Jei ciklas iteruoja per *Span* ar *MemoryMarshal* objektą, tai jis bus greitesnis nei iteruojant per sąrašą ar masyvą.
2. Jeigu patobulinimo teorinis laiko sudėtingumas yra tiesinis, tai per gautus greitaveikos atskaitos taškus bus galima nubrėžti tiesę.

4.2.1. Paprastas *for* ciklas

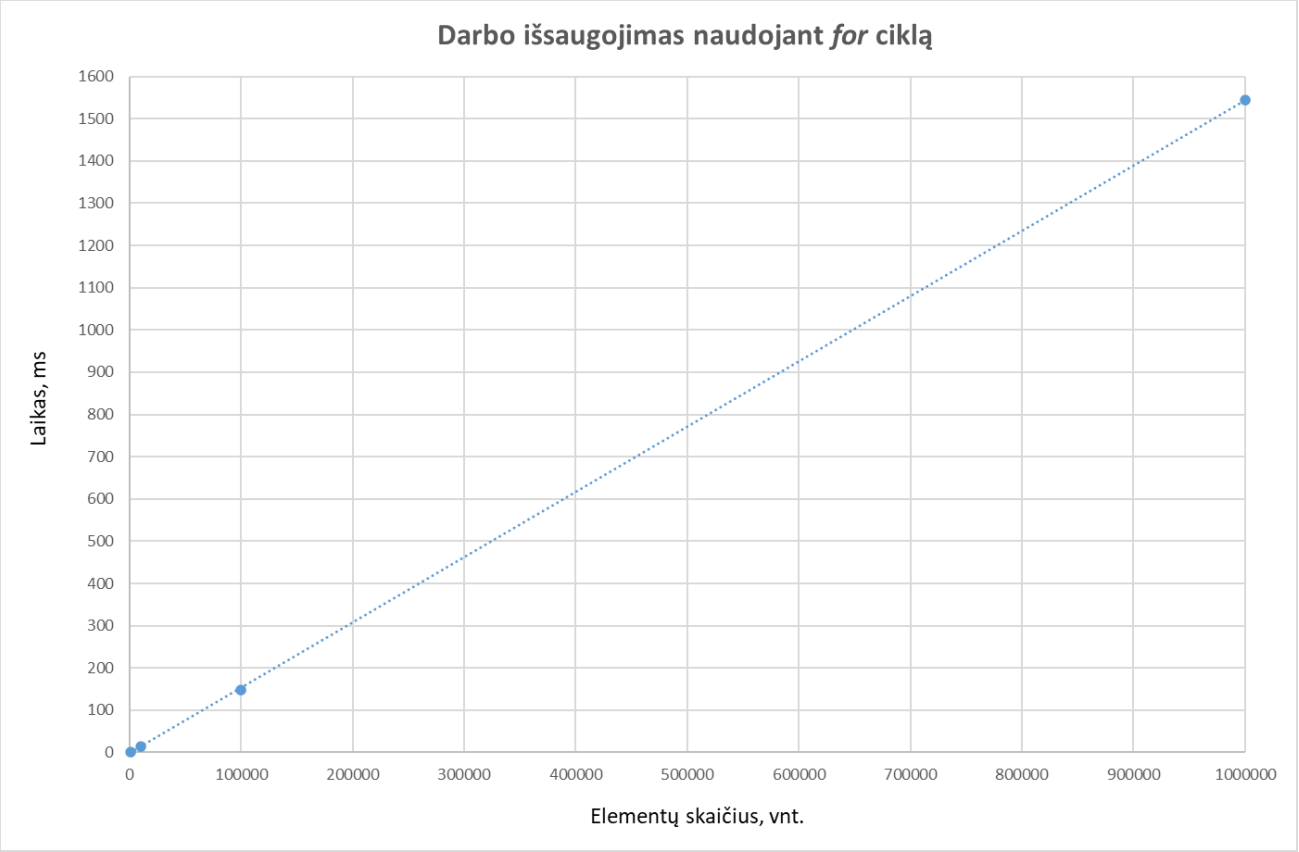
Įgyvendinus 3.2.1 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 94 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	67	5	1	9	38

94 pav. Darbo išsaugojimo naudojant *for* ciklą kodo metrikos

Šio patobulinimo palaikomumo indeksas yra 67, o tai yra trečias pagal gerumą rezultatas iš darbo išsaugojimo patobulinimų. Ciklomatinis sudėtingumas yra 5, tačiau beveik visi kiti patobulinimai taip pat turi tokį patį ciklomatinį sudėtingumą. Paveldėjimo medžio gylis yra 1, nes klasė įgyvendina vieną sąsają, kad būtų galima panaudoti priklausomybės inversiją. Klasės susietumas yra 9, o šį patobulinimą įgyvendinančios klasės dydis yra 38 kodo eilutės.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 95 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra antras pagal greitumą dirbant su 1000 ir 100000 elementų bei trečias dirbant su 10000 ir 1 milijonu objektų.



95 pav. Darbo išsaugojimo naudojant *for* ciklą greitaveika

4.2.2. Nesaugus *for* ciklas naudojant *Span* iš *List*

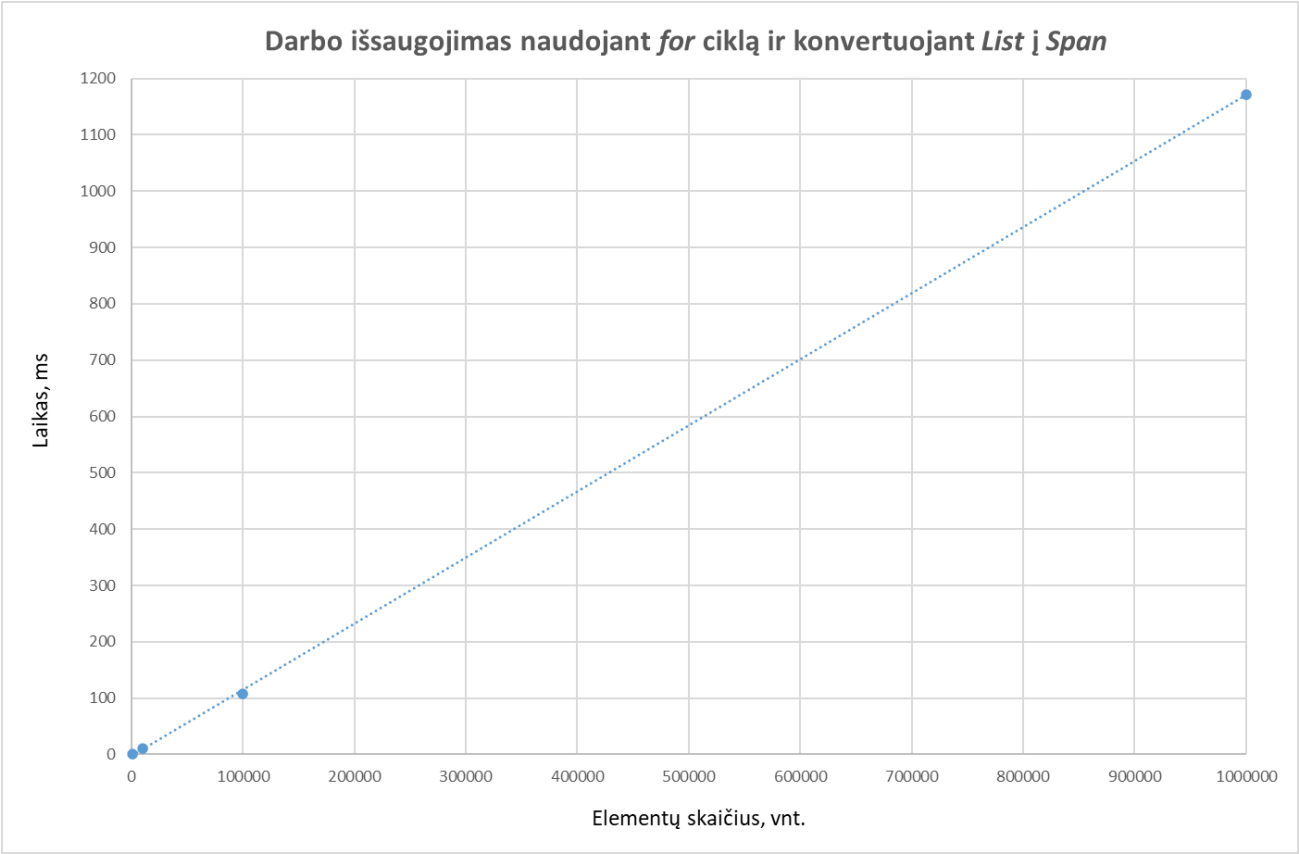
Įgyvendinus 3.2.2 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 96 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	67	5	1	10	42

96 pav. Darbo išsaugojimo naudojant *for* ciklą ir konvertuojant *List* į *Span* kodo metrikos

Šio patobulinimo palaikomumo indeksas yra 67, ciklomatinis sudėtingumas – 5, o paveldėjimo medžio gylis yra 1. Šios metrikos yra tokios pačios kaip ir paprasto *for* ciklo patobulinimo. Klasės susietumas yra 10, nes čia yra papildomai naudojama *Span* klasė. Šio patobulinimo realizacija taip pat yra šiek tiek didesnė nei paprasto *for* ciklo patobulinimo, nes reikia konvertuoti sąrašą į *Span* objektą.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 97 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas dalinasi pirmą vietą pagal greitumą dirbant su visais elementų skaičiais.



97 pav. Darbo išsaugojimo naudojant *for* ciklą ir konvertuojant *List* į *Span* greitaveika

4.2.3. Nesaugus *for* ciklas naudojant *Span* iš masyvo

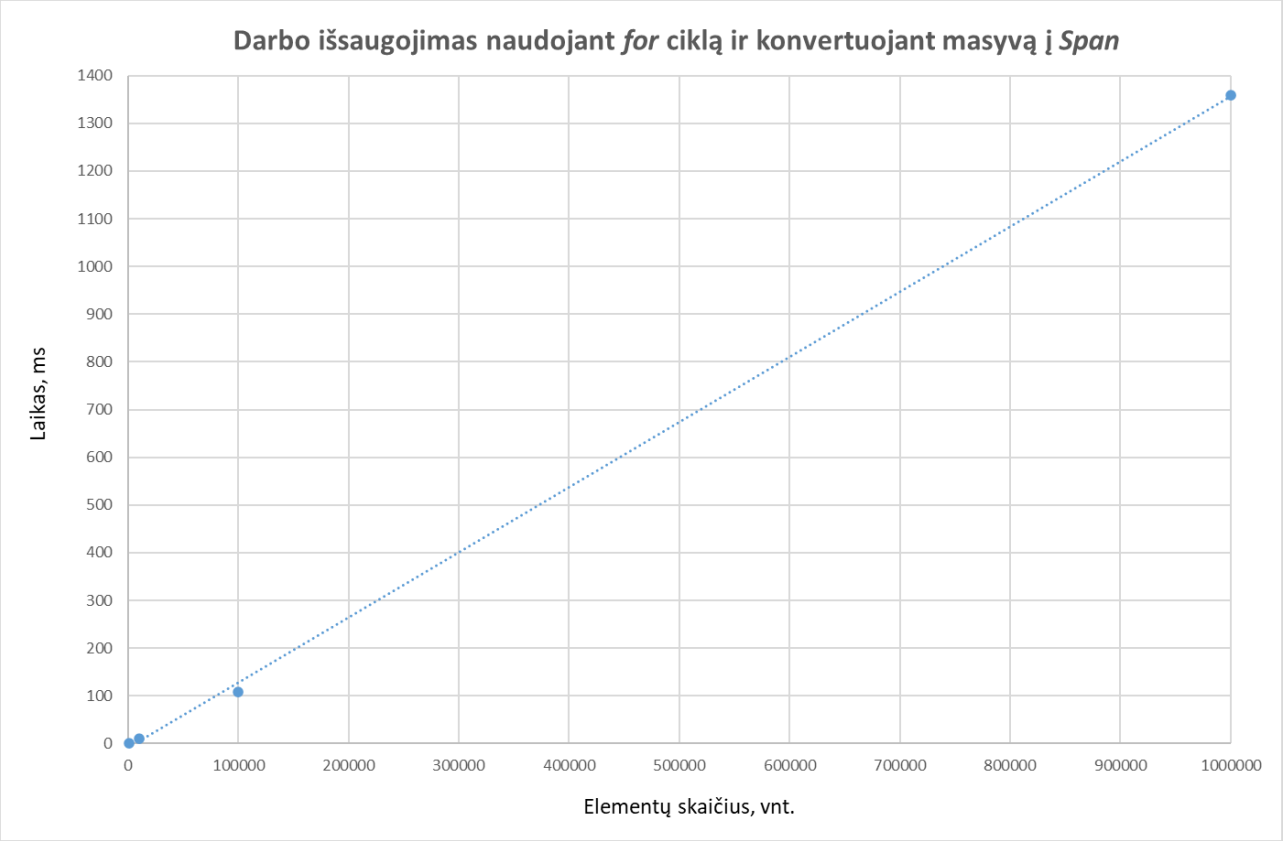
Įgyvendinus 3.2.3 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 98 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	67	5	1	10	42

98 pav. Darbo išsaugojimo naudojant *for* ciklą ir konvertuojant masyvą į *Span* kodo metrikos

Šio patobulinimo kodo metrikos yra tokios pačios lyginant su 3.2.2 skyrelyje aprašytu patobulinimu. Skirtumas yra tas, kad masyvo konvertavimui į *Span* objektą yra naudojamas masyvo metodas.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 99 pav. pateiktas grafikas. Grafike pavaizdutoa, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas dalinasi pirmą vietą dirbant su 1000 ir 100000 elementų bei yra antras dirbant su 10000 ir 1 milijonu objektų.



99 pav. Darbo išsaugojimo naudojant *for* ciklą ir konvertuojant masyvą į *Span* greitaveika

4.2.4. Nesaugus *for* ciklas naudojant *MemoryMarshal*

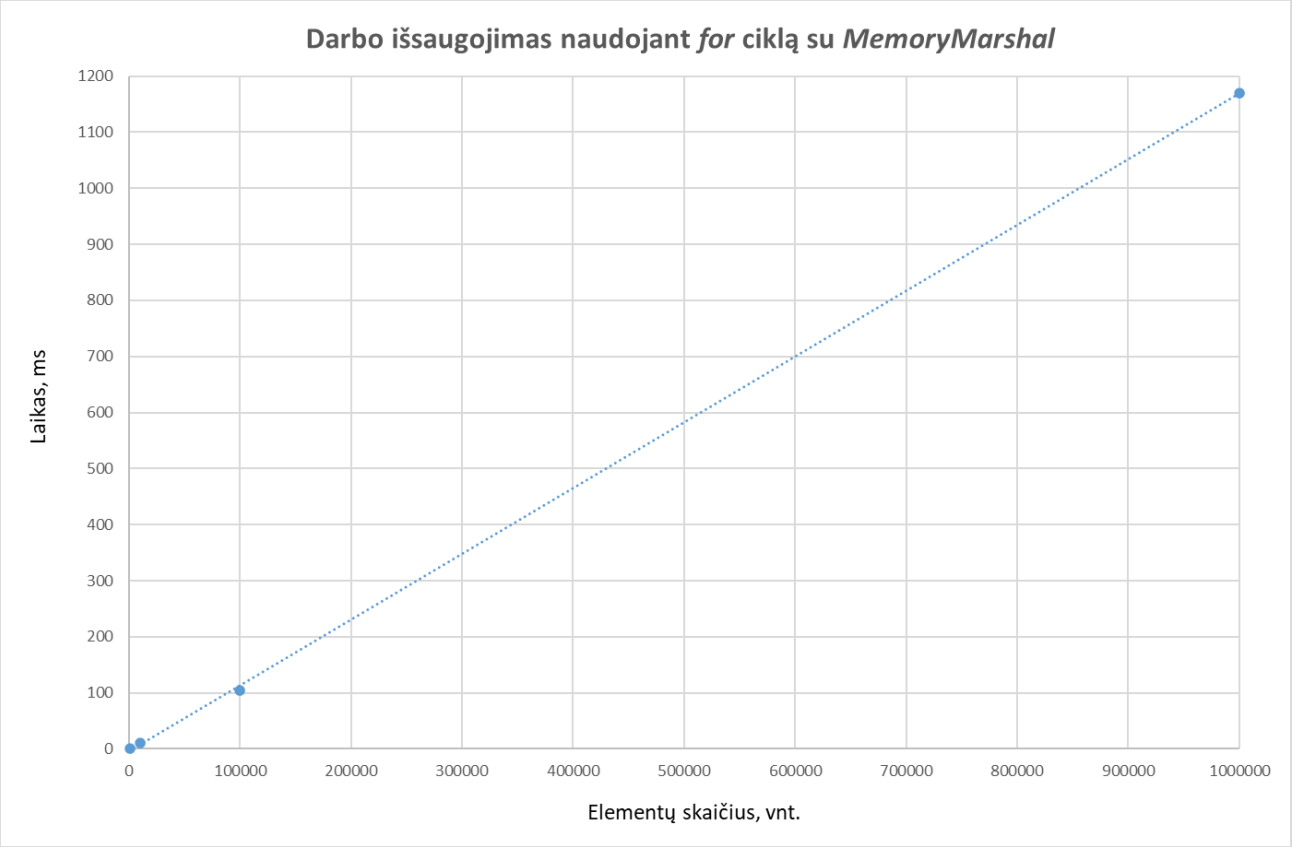
Įgyvendinus 3.2.4 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 100 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	65	5	1	10	44

100 pav. Darbo išsaugojimo naudojant *for* ciklą su *MemoryMarshal* kodo metrikos

Lyginant su *Span* klasę naudojančiais patobulinimais, šis būdas turi šiek tiek blogesnį palaikomumą. Jis taip pat yra keliomis eilutėmis didesnis, tačiau kitos kodo metrikos yra tokios pačios. Kitos metrikos yra tokios pačios kaip ir 4.2.2 ir 4.2.3 skyreliuose.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gauats 101 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra pirmas dirbant su 1000, 100000 ir 1 milijonu elementų bei yra antras dirbant su 10000 objektų.



101 pav. Darbo išsaugojimo naudojant *for* ciklą su *MemoryMarshal* greitaveika

4.2.5. Paprastas *foreach* ciklas

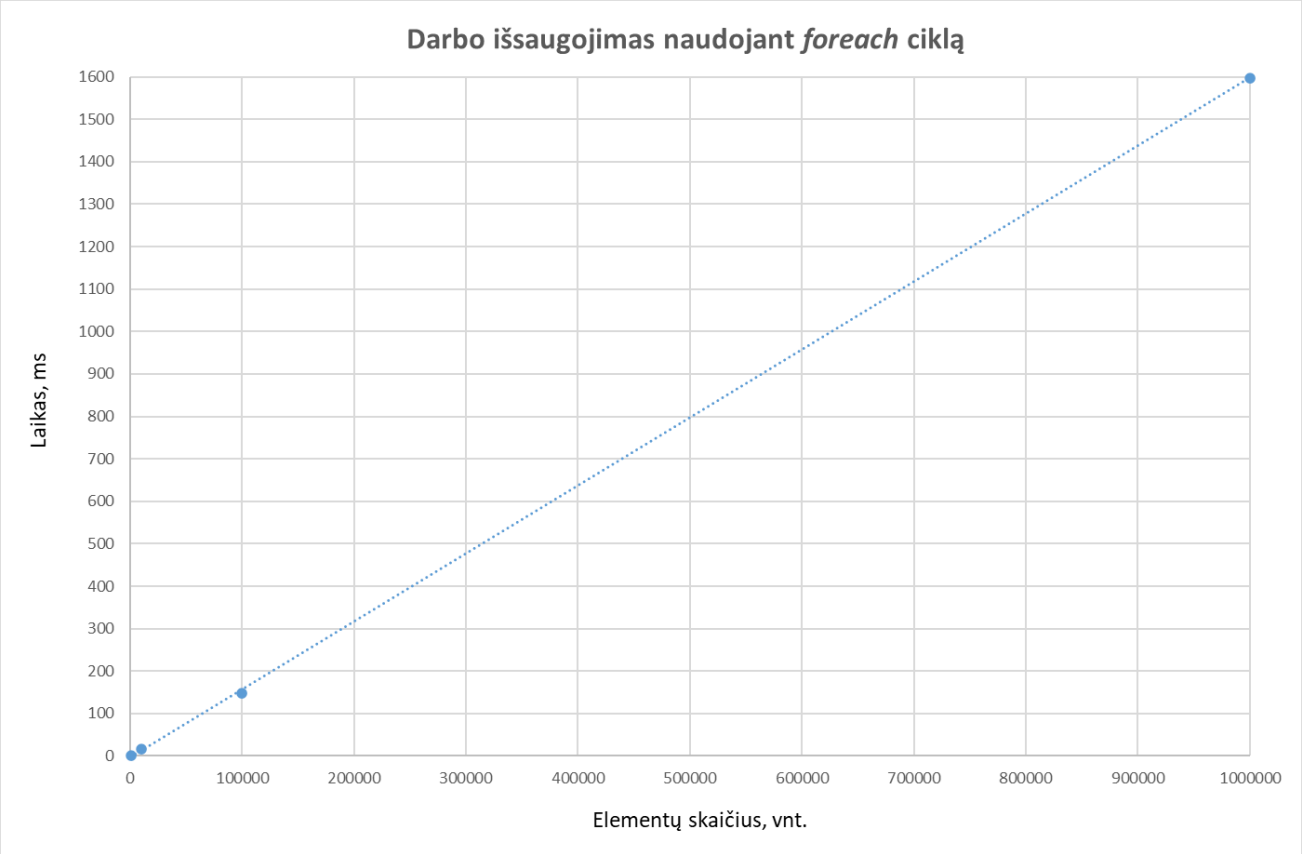
Šiuo metu sistemoje įgyvendintam darbo išsaugojimui *Visual Studio* pateikė 102 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	72	11	1	10	64

102 pav. Darbo išsaugojimo naudojant *foreach* ciklą kodo metrikos

Šis darbo išsaugojimo būdas yra vienintelis, kurio ciklomatinis sudėtingumas yra du kartus didesnis nei kitų patobulinimų. Kita vertus, jis turi geriausią palaikomumo įvertinimą, kuris yra bent keturiais punktais geresnis už kitus patobulinimus. Taip pat šio metodo įgyvendinimas užima daugiausiai kodo eilučių, nes yra kartojamas kodas. Klasių susietumas yra toks pat kaip ir daugelio kitų patobulinimų.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 103 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra antras dirbant su 1000 ir 100000 elementų, penktas dirbant su 10000 elementų bei ketvirtas dirbant su 1 milijonu objektų.



103 pav. Darbo išsaugojimo naudojant *foreach* ciklą greitaveika

4.2.6. Nesaugus *foreach* ciklas naudojant *Span* iš *List*

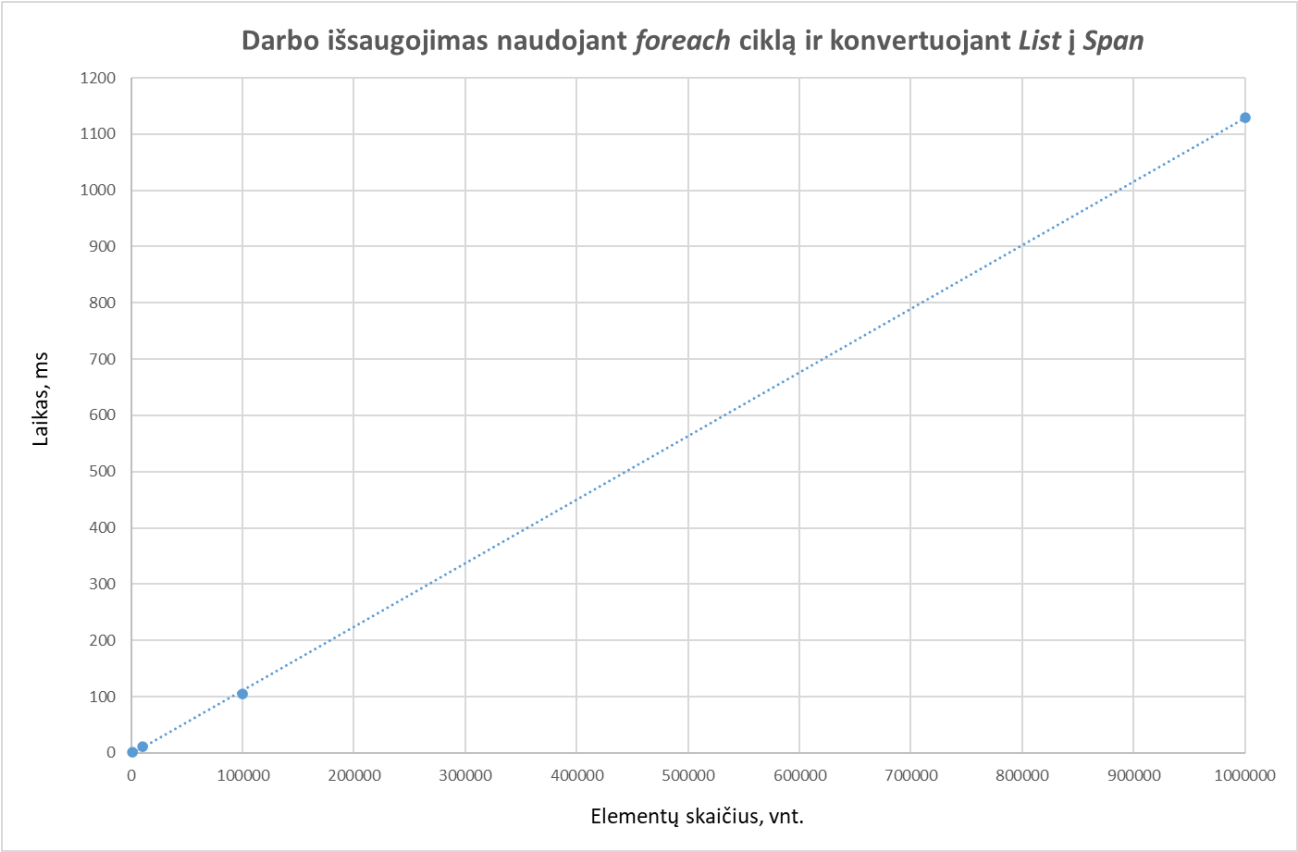
Įgyvendinus 3.2.6 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 104 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	68	5	1	11	38

104 pav. Darbo išsaugojimo naudojant *foreach* ciklą ir konvertuojant *List* į *Span* kodo metrikos

Lyginant su panašaus patobulinimo kodo metrikomis, aprašytomis 4.2.2 skyrelyje, šis būdas turi šiek tiek geresnį palaikomumo indeksą. Taip pat jis yra mažesnis kodo eilučių skaičiumi. Kita vertus, šio būdo įgyvendinimas turi blogiausią klasių susietumą iš visų patobulinimų.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 105 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra pirmas dirbant su 1000, 100000 ir 1 milijonu elementų bei antras dirbant su 10000 elementų.



105 pav. Darbo išsaugojimo naudojant *foreach* ciklą ir konvertuojant *List* į *Span* greitaveika

4.2.7. Nesaugus *foreach* ciklas naudojant *Span* iš masyvo

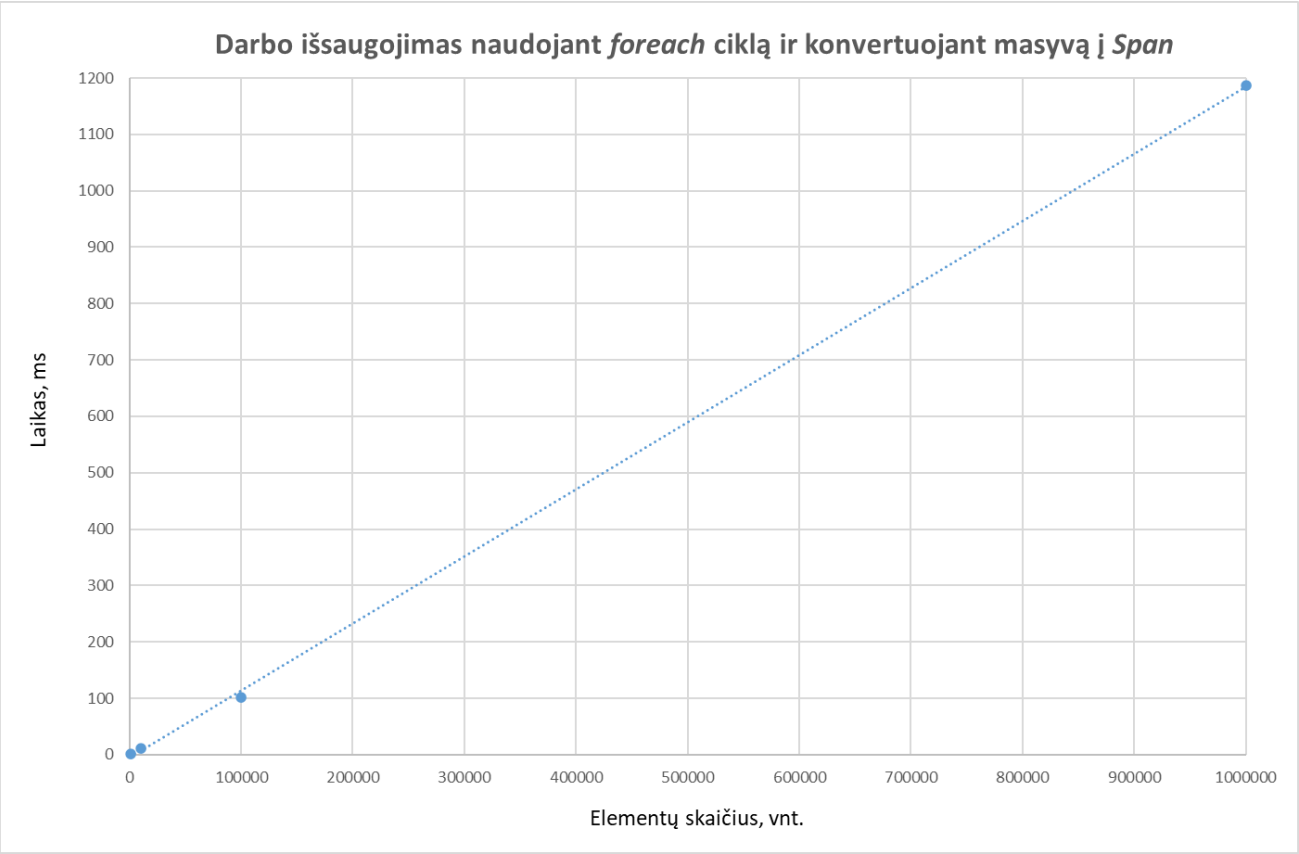
Įgyvendinus 3.2.7 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 106 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	68	5	1	10	38

106 pav. Darbo išsaugojimo naudojant *foreach* ciklą ir konvertuojant masyvą į *Span* kodo metrikos

Visos šio patobulinimo metrikos yra beveik tokios pačios lyginant su *foreach* ciklu, kuriame yra naudojamas *Span* iš sąrašo. Vienintelis skirtumas yra klasių susietumas, kuris šiame patobulinime yra šiek tiek geresnis.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 107 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra pirmas pagal greitumą dirbant su 1000, 100000 ir 1 milijonu elementų bei antras dirbant su 10000 objektų.



107 pav. Darbo išsaugojimo naudojant *foreach* ciklą ir konvertuojant masyvą į *Span* greitaveika

4.2.8. Paprastas *while* ciklas

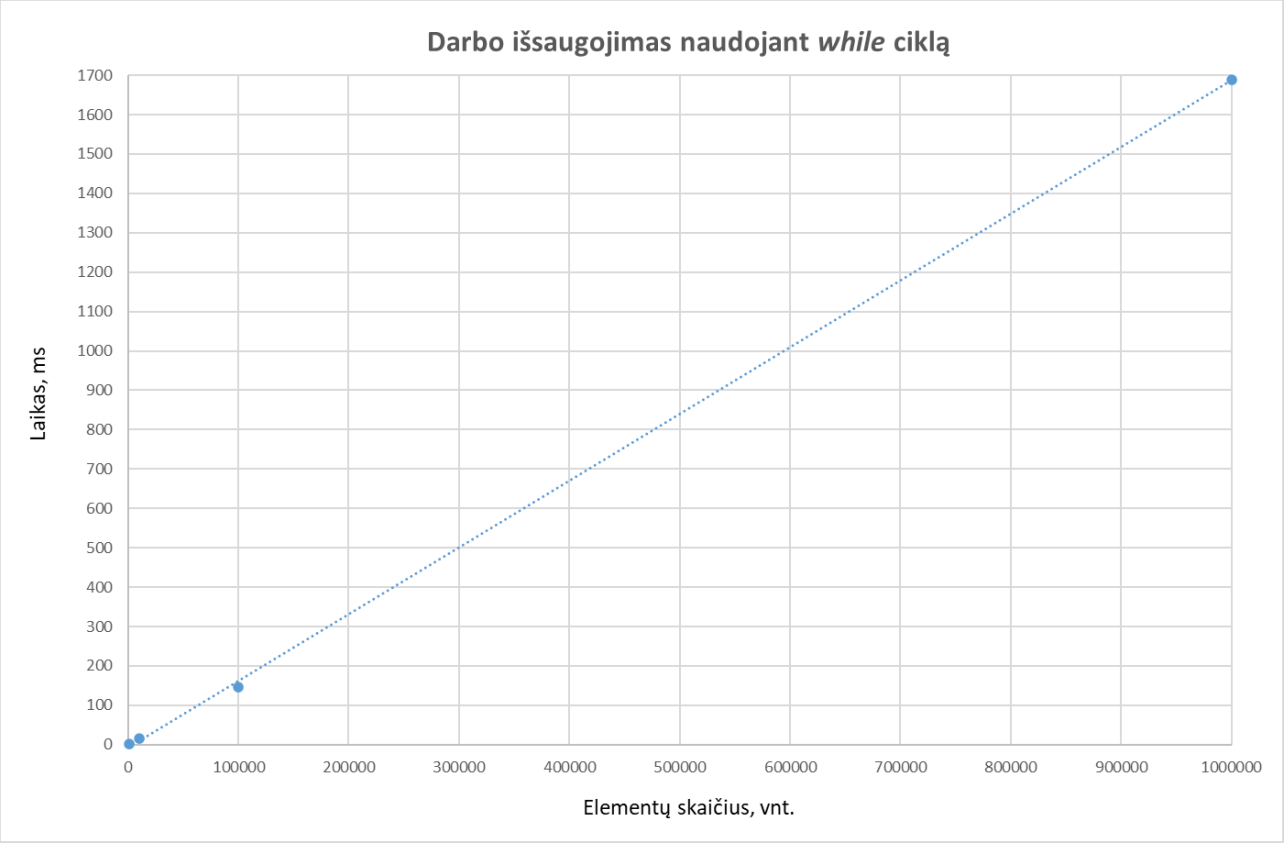
Įgyvendinus 3.2.8 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 108 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	67	5	1	8	42

108 pav. Darbo išsaugojimo naudojant *while* ciklą kodo metrikos

Šio patobulinimo palaikomumo indeksas ir ciklomatinis sudėtingumas yra panašūs į daugelio kitų patobulinimų. Taip pat verta paminėti, kad jis turi geriausią klasių susietumą – 8.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 109 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra antras pagal greitumą dirbant su 1000 ir 100000 elementų, ketvirtas dirbant su 10000 objektų bei penktas dirbant su 1 milijonu įvykių.



109 pav. Darbo išsaugojimo naudojant *while* ciklą greitaveika

4.2.9. Nesaugus *while* ciklas naudojant *MemoryMarshal*

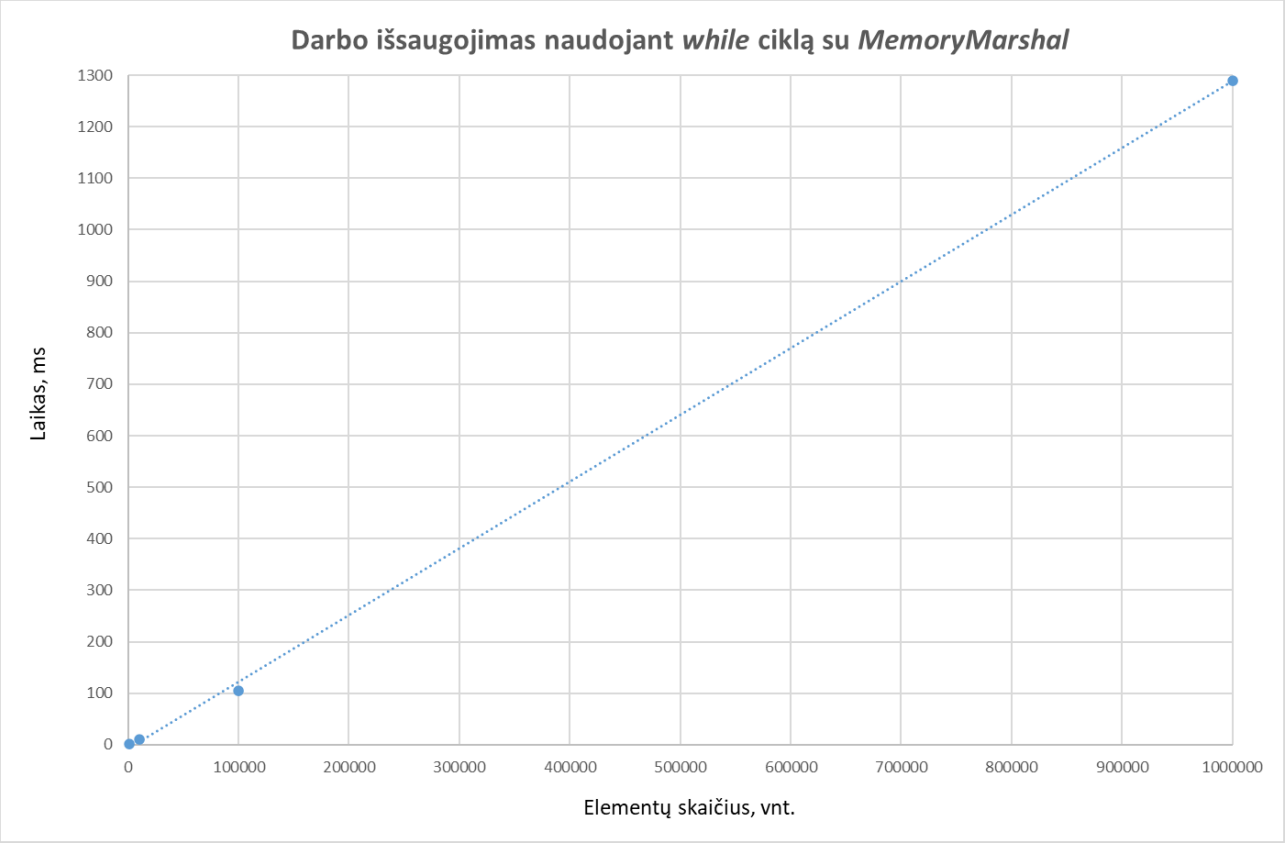
Įgyvendinus 3.2.9 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 110 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkSaver	64	5	1	10	48

110 pav. Darbo išsaugojimo naudojant *while* ciklą su *MemoryMarshal* kodo metrikos

Šis patobulinimas turi blogiausią palaikomumo indeksą. Jis atsilieka vienu punktu nuo kito patobulinimo, taip pat naudojančio *MemoryMarshal* klasę. Kitos kodo metrikos neišsiskiria iš kitų patobulinimų. Kita vertus, šis patobulinimas taip pat užima daugiausiai kodo eilučių, išskyrus jau sistemoje realizuotą paprastą *foreach* būdą.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 111 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja tiesinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis metodas yra pirmas pagal greitumą dirbant su 1000 ir 100000 elementų bei antras dirbant su 10000 ir 1 milijonu įvykių.



111 pav. Darbo išsaugojimo naudojant *while* ciklą su *MemoryMarshal* greitaveika

4.2.10. Rezultatų apibendrinimas

Visų patobulinimų kodo metrikų įvertinimai yra pateikti 3 priede, o greitaveikos tyrimo rezultatai – 4 priede.

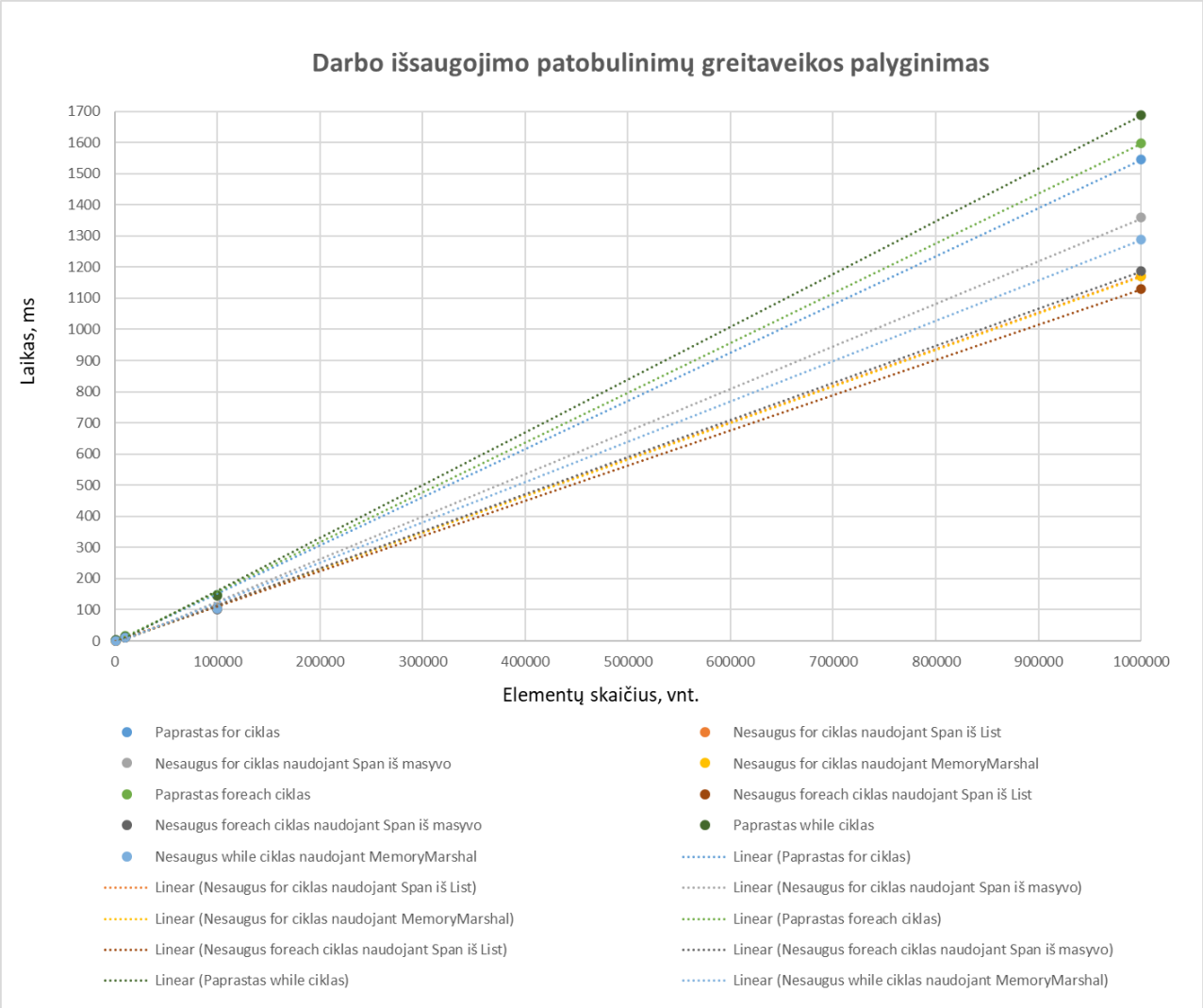
Geriausi darbo išsaugojimo patobulinimai pagal kiekvieną kodo metriką yra pateikiami 5 lentelėje. Nė vienas patobulinimas nėra geriausias visose kodo metrikose. Geriausias patobulinimas pagal palaikomumą yra paprastas *foreach* ciklas, kurio palaikomumo indeksas yra 72, o blogiausias patobulinimas yra *while* ciklas su *MemoryMarshal*. Visi patobulinimai turi tokį patį ciklomatinį sudėtingumą, išskyrus sistemoje šiuo metu įgyvendintą *foreach* ciklą, kurio ciklomatinis sudėtingumas yra du kartus didesnis. Taip gali būti dėl kodo kartojimo, nes pats darbo išsaugojimo algoritmas nebuvo keičiamas. Patobulinimų klasių susietumo metrikos skiriasi tik keliomis klasėmis. Geriausi patobulinimai įvertinant visas kodo metrikas yra paprastas *for* ciklas, nesaugus *foreach* ciklas naudojant *Span* iš *List*, nesaugus *foreach* ciklas naudojant *Span* iš masyvo bei paprastas *while* ciklas. Jeigu ignoruosime kodo eilučių metrikas, tai geriausias patobulinimas yra paprastas *while* ciklas.

5 lentelė. Geriausi darbo išsaugojimo patobulinimai pagal kodo metrikas

Kodo metrika	**Geriausi patobulinimai**
Palaikomumo indeksas	Paprastas *foreach* ciklas.
Ciklomatinis sudėtingumas	Visi, išskyrus paprastą *foreach* ciklą.
Paveldėjimo medžio gylis	Visi vienodi.

Klasių susietumas	Paprastas *while* ciklas.
Kodo eilučių skaičius	Paprastas *foreach* ciklas. Nesaugus *foreach* ciklas naudojant *Span* iš *List*. Nesaugus *foreach* ciklas naudojant *Span* iš masyvo.

Pagal greitaveiką (žr. 112 pav.), greičiausias patobulinimas yra *foreach* ciklas konvertuojant *List* į *Span* objektą. Netoli atsilieka ir kiti patobulinimai, naudojantys *Span* bei *MemoryMarshal* klases. Lėčiausi patobulinimai yra paprastas *foreach* ciklas, paprastas *while* ciklas bei šiuo metu sistemoje įgyvendintas paprastas *foreach* ciklas.



112 pav. Darbo išsaugojimo patobulinimų greitaveikos palyginimas

Įvertinus kodo metrikas ir greitaveikos tyrimą, geriausias darbo išsaugojimo būdas yra *foreach* ciklas naudojant *Span* objektą, gautą iš *List* objekto. Nors paprastas *while* ciklas yra geriausias pagal kodo metrikas, tačiau jis yra daug lėtesnis nei *Span* ar *MemoryMarshal* klases naudojantys patobulinimai. Be to, visų metodų kodo metrikos yra labai panašios, todėl prioritetas yra teikiamas greitaveikai.

Galiausiai, pirma hipotezė yra teisinga. 112 pav. pateiktame grafike galima matyti, kad patobulinimai, naudojantys *Span* bei *MemoryMarshal*, yra greitesni už *List* ar masyvą naudojančius metodus. Antra hipotezė taip pat yra teisinga, nes per atitinkamo patobulinimo atskaitos taškus galima nubrėžti tiesę.

4.3. Darbo įkėlimo eksperimentų rezultatai

Darbo įkėlimo hipotezės:
1. Jei yra naudojamas *Factory* projektavimo šablonas, tai kodo metrikos bus geresnės.
2. Jei failas yra skaitomas naudojant *StreamReader*, tai jis bus skaitomas ilgiau nei naudojant sisteminį metodą, grąžinantį failo eilučių *IEnumerable* objektą.

4.3.1. Failo skaitymas po vieną eilutę naudojant *StreamReader*

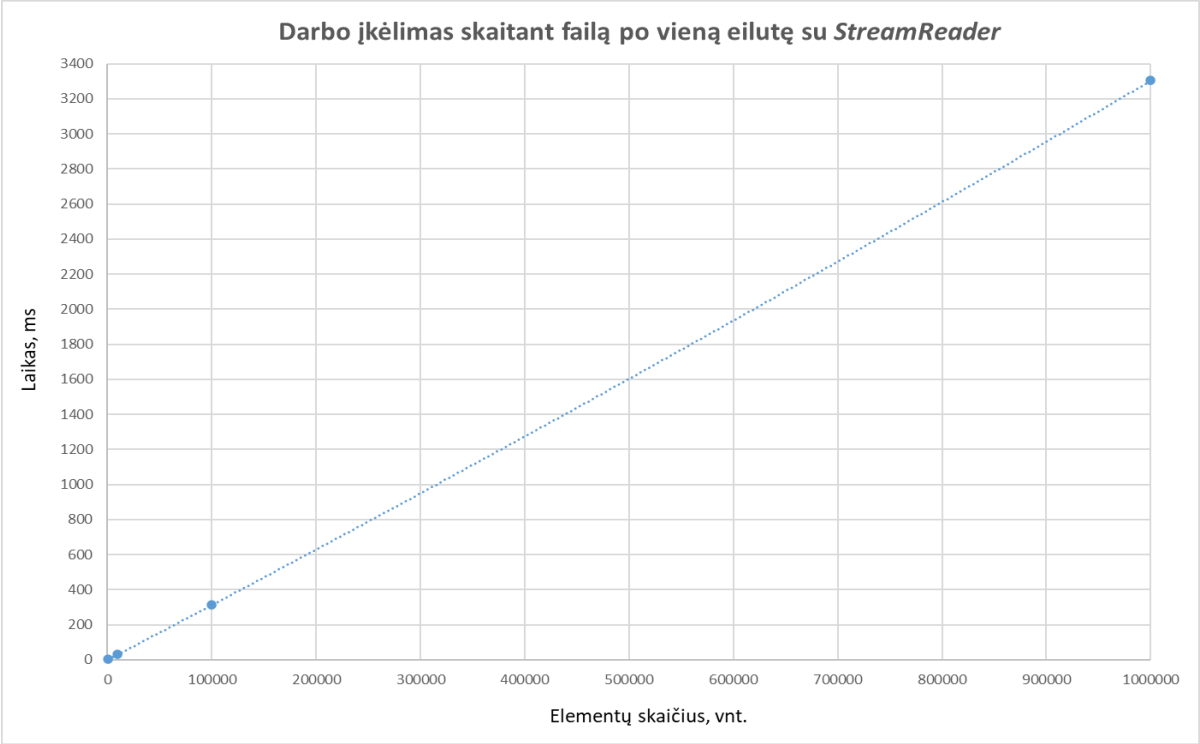
Šiuo metu sistemoje įgyvendintam darbo įkėlimui *Visual Studio* pateikė 113 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkLoader	61	31	1	24	322

113 pav. Darbo įkėlimo skaitant failą po vieną eilutę su *StreamReader* kodo metrikos

Šio būdo palaikomumo indeksas ir klasių susietumas yra gana neblogi, tačiau ciklomatinis sudėtingumas yra per didelis. Idealiu atveju jis neturėtų viršyti 20, tačiau šis būdas yra įvertintas 31. Taip pat dėl kodo kartojimo klasės dydis smarkiai išaugo.

Atlikus greitaveikos testavimą, buvo sudarytas 114 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja antros eilės priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis darbo įkėlimo metodas su visais elementų kiekiais buvo lėtesnis nei kitas patobulinimas.



114 pav. Darbo įkėlimo skaitant failą po vieną eilutę su *StreamReader* greitaveika

4.3.2. Darbo įkėlimas naudojant *Factory* projektavimo šabloną

Įgyvendinus 3.3.2 skyrelyje aprašytą patobulinimą, *Visual Studio* apskaičiavo 115 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
FileWorkLoader	62	22	1	26	181

115 pav. Darbo įkėlimo naudojant *Factory* projektavimo šabloną kodo metrikos

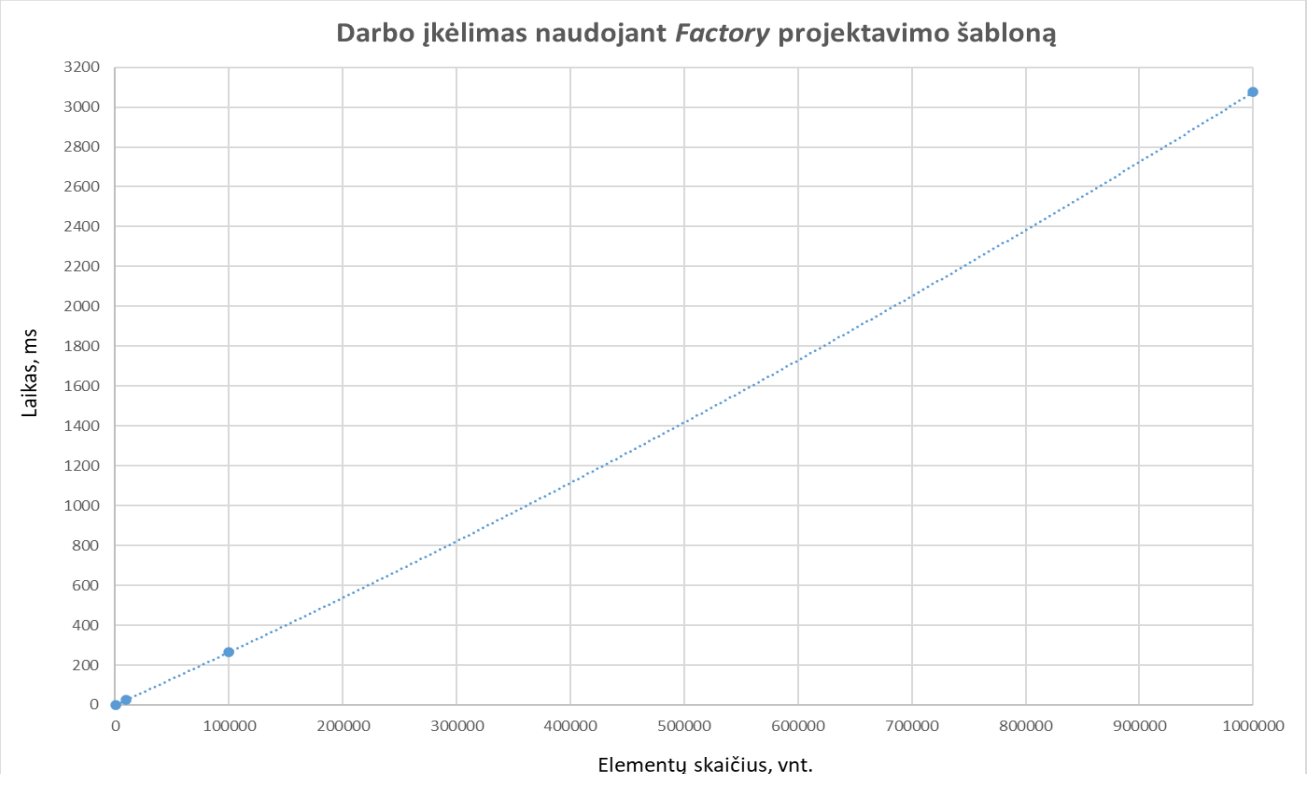
Lyginant su jau sistemoje įgyvendintu darbo įkėlimo metodu, šis patobulinimas turi daug geresnį ciklomatinį sudėtingumą bei palaikomumo indeksas yra šiek aukštesnis, o tai reiškia šiek tiek lengvesnį palaikomumą. Šis patobulinimas taip pat yra beveik du kartus mažesnis, nes buvo panaikintas kodo kartojimas. Kita vertus, klasių susietumo metrika šiek tiek pablogėjo, nes reikia daugiau klasių, kad būtų galima panaudoti *Factory* projektavimo šabloną.

Factory projektavimo šabloną realizuojančios klasės kodo metrikos yra pateiktos 116 pav. jos palaikomumo indeksas yra labai aukštas, o ciklomatinis sudėtingumas rodo, kad yra tik vienas nepriklausomas kodo vykdymo kelias.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
AbstractionLevelFactory	87	1	1	9	14

116 pav. *Factory* klasės kodo metrikos

Atlikus šio patobulinimo greitaveikos testavimą, buvo sudarytas 117 pav. pateiktas grafikas. Grafike pavaizdutoa, kad egzistuoja antros eilės priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis patobulinimas su visais elementų kiekiais yra greitesnis nei šiuo metu sistemoje įgyvendintas metodas.



117 pav. Darbo įkėlimo naudojant *Factory* projektavimo šabloną greitaveika

4.3.3. Rezultatų apibendrinimas

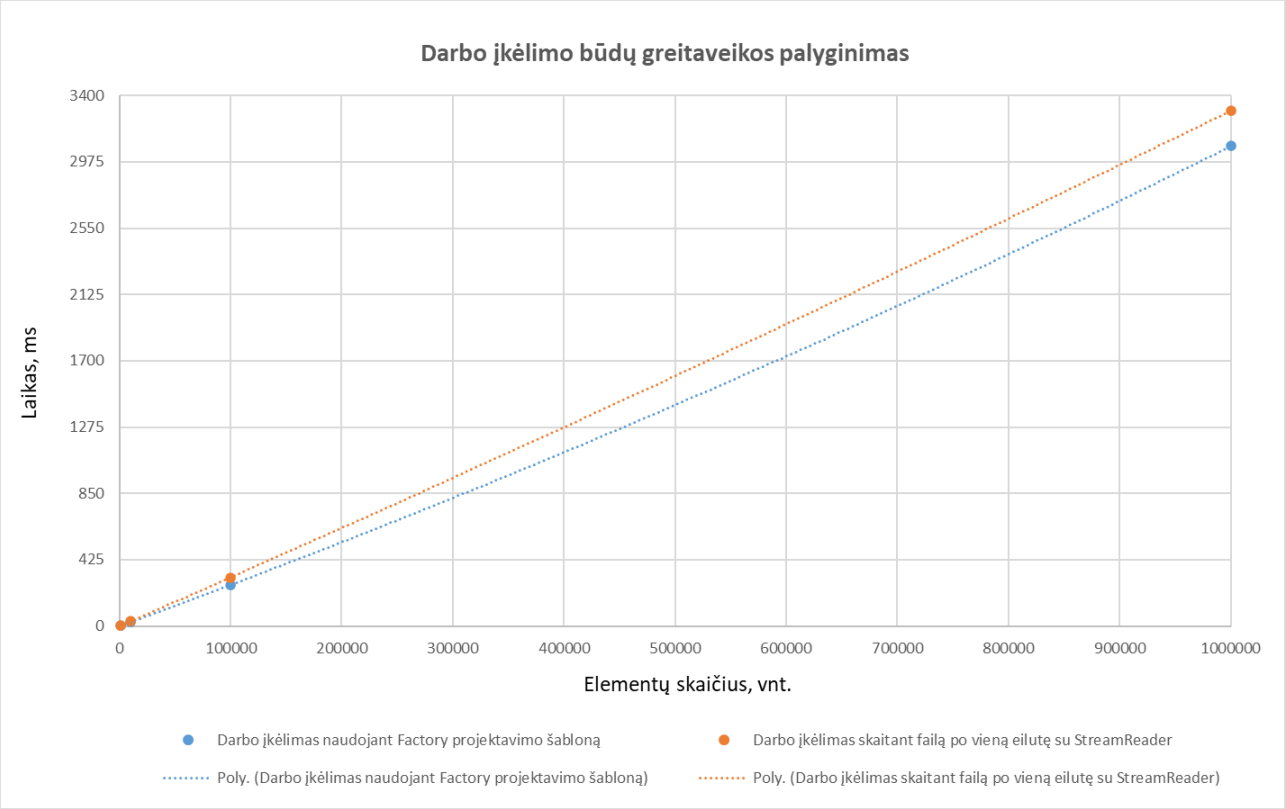
Darbo įkėlimo būdų kodo metrikų įvertinimai yra pateikti 5 priede, o greitaveikos tyrimo rezultatai – 6 priede.

Geriausias darbo įkėlimo būdas pagal kiekvieną kodo metriką yra pateikiamas 6 lentelėje. Geriausias būdas pagal metrikas yra *Factory* šablono naudojimas. Jis turi šiek tiek didesnį palaikomumo indeksą, bet tai nesuteikia didelio pranašumo. Didžiausias skirtumas yra ciklomatinio sudėtingumo metrikoje. *Factory* projektavimo šablonas ir *IEnumerable* objekto naudojimas beveik padėjo sumažinti ciklomatinį sudėtingumą iki rekomenduojamo lygio. Taip pat vienas iš projektavimo šablonų naudojimo privalumų yra sumažintas kodo kartojimas, dėl ko klasės kodo eilučių skaičius sumažėjo apie 44%. Kita vertus, klasių susietumas šiek tiek padidėjo, nes reikia kreiptis į *Factory* klasę, kad ji sukurtų reikiamo tipo objektą.

6 lentelė. Geriausias darbo įkėlimo būdas pagal kodo metrikas

Kodo metrika	**Geriausias būdas**
Palaikomumo indeksas	Darbo įkėlimas naudojant Factory projektavimo šabloną.
Ciklomatinis sudėtingumas	Darbo įkėlimas naudojant Factory projektavimo šabloną.
Paveldėjimo medžio gylis	Abu vienodi.
Klasių susietumas	Failo skaitymas po vieną eilutę naudojant *StreamReader*.
Kodo eilučių skaičius	Darbo įkėlimas naudojant Factory projektavimo šabloną.

Pagal greitaveiką (žr. 118 pav.), greičiausias darbo įkėlimo būdas yra *Factory* projektavimo šablono naudojimas. Dirbant su 1000 elementų, patobulinimas, naudojantis *Factory* projektavimo šabloną, yra 23% greitesnis ir naudoja 5% mažiau atminties, dirbant su 10000 elementų yra 14% greitesnis ir užima 5% mažiau atminties, dirbant su 100000 elementų yra 15% greitesnis ir užima 4% mažiau atminties, o dirbant su 1 milijonu įvykių yra 6% greitesnis ir užima 4% mažiau atminties. Dirbant su 1 milijonu įvykių, patobulinimas gali sutaupyti 227,969ms bei 75,13MB atminties.



118 pav. Darbo įkėlimo būdų greitaveikos palyginimas

Įvertinus kodo metrikas ir greitaveikos palyginimą, *Factory* projektavimo šabloną naudojantis darbo įkėlimo būdas yra vienareikšmiškai geresnis.

Galiausiai, pirma hipotezė yra teisinga, nes visos, išskyrus klasių susietumo, metrikos yra geresnės naudojant projektavimo šabloną. Kodo susietumo metrika taip pat būtų pagerėjusi, jei būtų daugiau abstrakcijos lygmenų. Antra hipotezė taip pat yra teisinga. Taip gali būti dėl to, kad .NET karkasas gali būti optimizaveęs viso failo skaitymą eilutėmis.

4.4. Abstrakcijos lygmenų formavimo eksperimentų rezultatai

Abstrakcijos lygmenų formavimo hipotezės:

1. Jei yra naudojamas *Chain of Responsibility* projektavimo šablonas, tai kodo metrikos bus geresnės.
2. Jeigu patobulinimo teorinis laiko sudėtingumas yra kvadratinis, tai per gautus greitaveikos atskaitos taškus bus galima nubrėžti antros eilės kreivę.

4.4.1. Abstrakcijos lygmenų formavimas naudojant *switch* sakinį

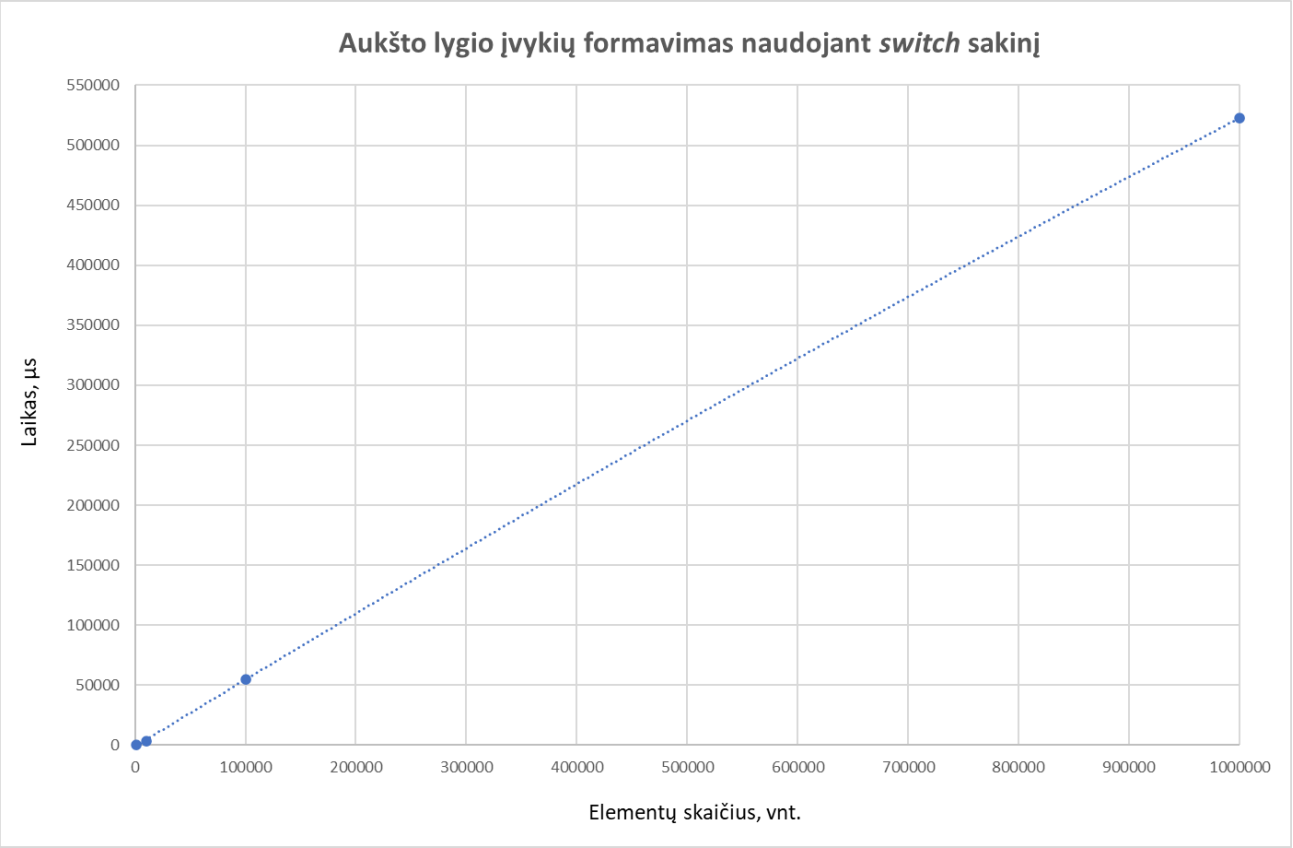
Šiuo metu sistemoje įgyvendintam aukšto lygio įvykių formavimui *Visual Studio* pateikė 119 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
HighLevelEventsAbstractor	66	44	1	10	308

119 pav. Aukšto lygio įvykių formavimo naudojant *switch* sakinį kodo metrikos

Šio būdo palaikomumo indeksas yra 66, o tai reiškia neblogą šio komponento palaikymą. Kita vertus, ciklomatinis sudėtingumas yra 44, o tai yra dvigubai didesnis nei yra rekomenduojama. Taip pat šią klasę sudaro gana daug kodo eilučių, dėl ko gali sumažėti jos suprantamumas.

Atlikus greitaveikos testavimą, buvo gautas 120 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis aukšto lygio įvykių formavimo būdas su visais elementų skaičiais yra šiek tiek greitesnis nei tiriamas patobulinimas.



120 pav. Aukšto lygio įvykių formavimo naudojant *switch* sakinį greitaveika

Sistemoje įgyvendintam žemo lygio įvykių formavimui *Visual Studio* pateikė 121 pav. pateiktas metrikas.

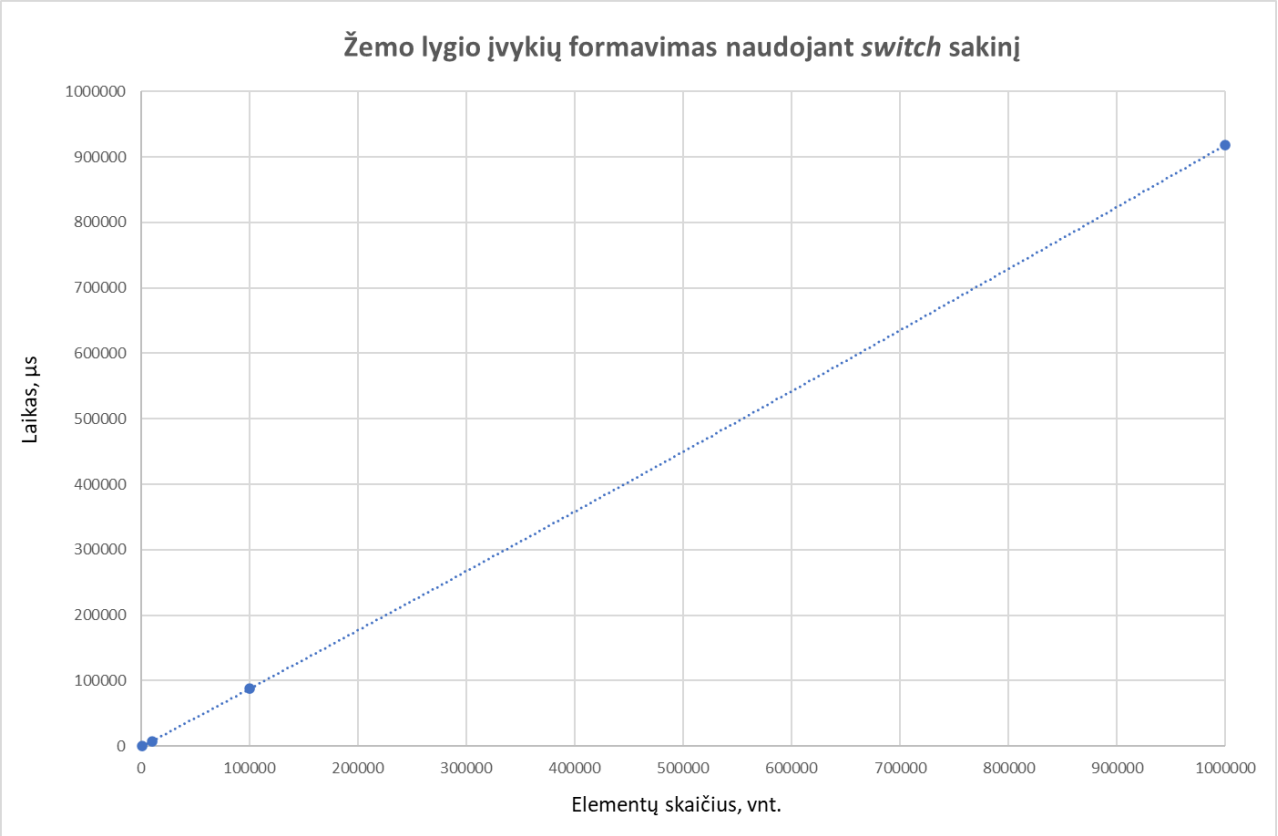
Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
LowLevelEventsAbstractor	63	104	1	12	555

121 pav. Žemo lygio įvykių formavimo naudojant *switch* sakinį kodo metrikos

Tokio būdo palaikomumo indeksas yra 63, tačiau ciklomatinis sudėtingumas yra 104, o tai ypač apsunkina funkcionalumo testavimą ir suprantamumą. Šią klasę yra būtina pertvarkyti, kad būtų sumažintas ciklomatinis sudėtingumas. Taip pat ši klasė yra labai didelė, nes ją sudaro 555 kodo eilutės.

Atlikus greitaveikos testavimą, buvo gauats 122 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis žemo lygio įvykių

formavimo būdas su visais elementų skaičiais taip pat yra šiek tiek greitesnis nei tiriamas patobulinimas.



122 pav. . Žemo lygio įvykių formavimo naudojant *switch* sakinį greitaveika

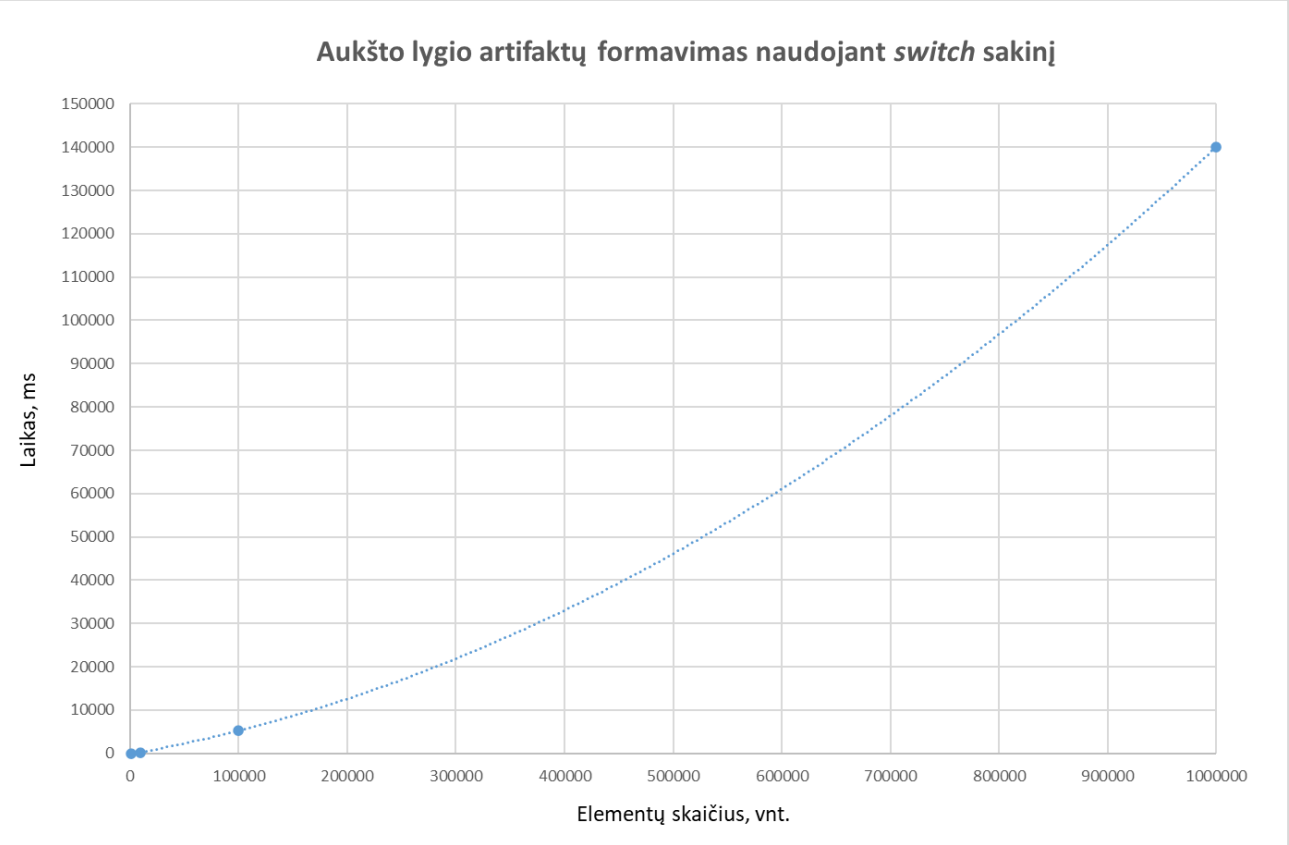
Šiuo metu sistemoje įgyvendintam aukšto lygio artifaktų formavimui *Visual Studio* pateikė 123 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
HighLevelArtefactsAbstractor	63	94	1	14	610

123 pav. Aukšto lygio artifaktų formavimo naudojant *switch* sakinį kodo metrikos

Šio būdo palaikomumo indeksas yra 63, o ciklomatinis sudėtingumas taip pat yra labai didelis. Aukšto lygio artifaktų formavimo klasė yra dar didesnė už žemo lygio įvykių formavimo klasę. Šios metrikos taip pat verčia pertvarkyti šią klasę.

Atlikus greitaveikos testavimą, buvo gautas 124 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis aukšto lygio artifaktų formavimo būdas su visais elementų skaičiais yra greitesnis nei tiriamas patobulinimas. Taip pat verta paminėti, kad šis lygio formavimas yra 1000 kartų lėtesnis nei kitų lygių formavimas.



124 pav. Aukšto lygio artifaktų formavimo naudojant *switch* sakinį greitaveika

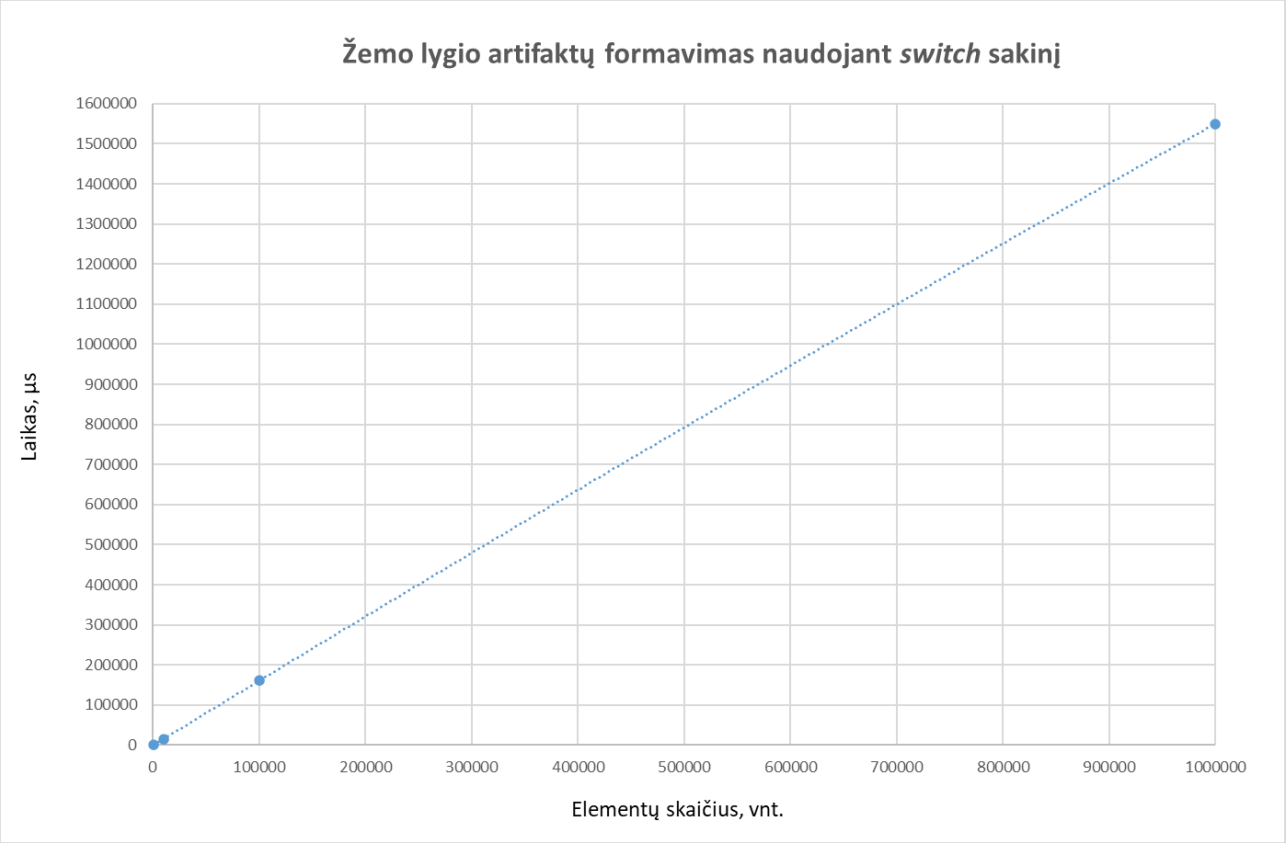
Sistemoje įgyvendintam žemo lygio artifaktų formavimui *Visual Studio* pateikė 125 pav. pateiktas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
LowLevelArtefactsAbstractor	62	40	1	11	208

125 pav. Žemo lygio artifaktų formavimo naudojant *switch* sakinį kodo metrikos

Šio būdo palaikomumo indeksas yra 62, o ciklomatinis sudėtingumas yra šiek tiek geresnis nei žemo lygio įvykių ir aukšto lygio artifaktų formavimų. Žemo lygio artifaktų formavimo klasė taip pat yra mažesnė už kitas abstrakcijos lygių formavimo klases, tačiau vis tiek yra per didelė.

Atlikus greitaveikos testavimą, buvo gautas 126 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis žemo lygio artifaktų formavimo būdas su visais elementų skaičiais užtrunka apytiksliai tiek pat laiko lyginant su tiriamu patobulinimu.



126 pav. Žemo lygio artifaktų formavimo naudojant *switch* sakinį greitaveika

4.4.2. Abstrakcijos lygmenų formavimas naudojant *Chain of Responsibility* projektavimo šabloną

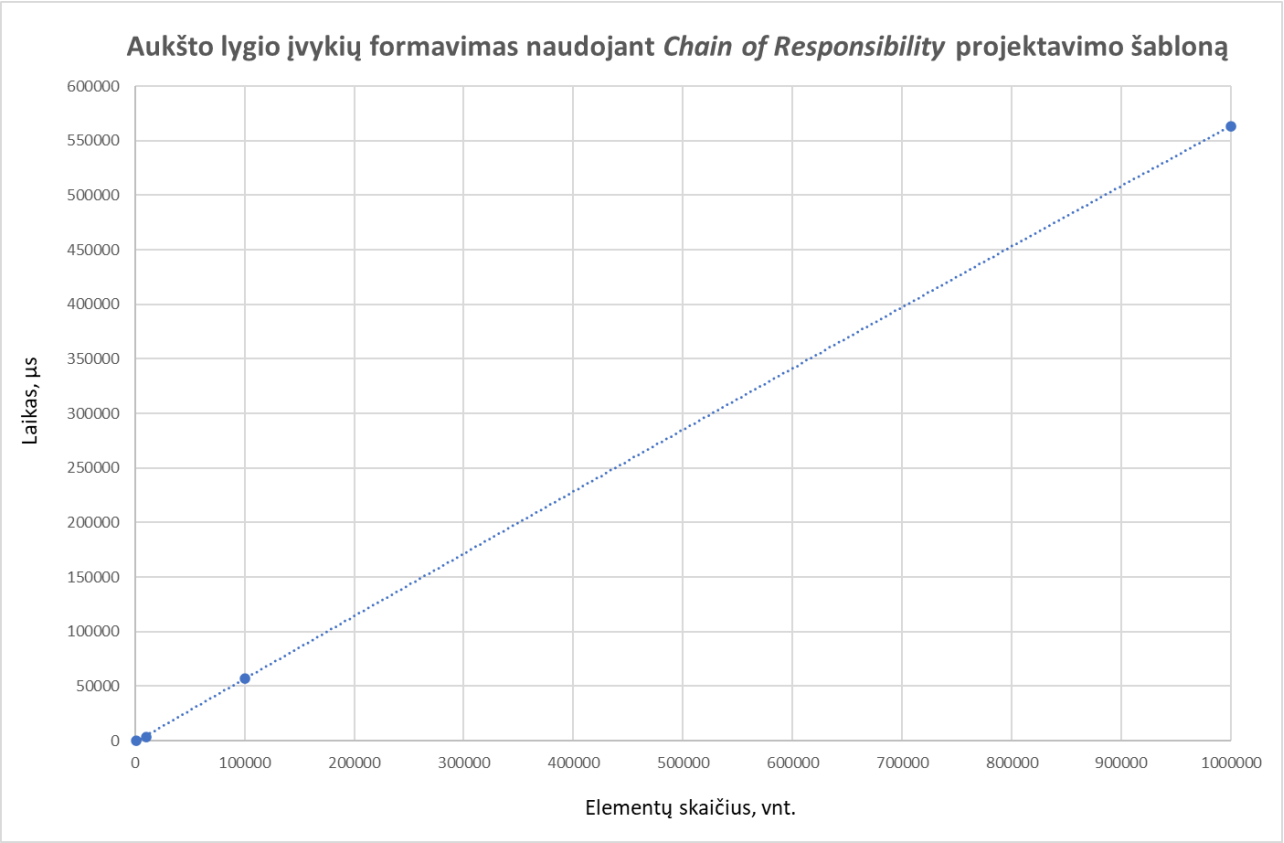
Įgyvendinus 3.4.2 skyrelyje aprašytą aukšto lygio įvykių patobulinimą, *Visual Studio* apskaičiavo 127 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
HighLevelEventsAbstractor	68	4	1	11	37

127 pav. Aukšto lygio įvykių formavimo naudojant *Chain of Responsibility* šabloną kodo metrikos

Lyginant su jau sistemoje įgyvendintu aukšto lygio įvykių formavimu, šis patobulinimas turi daug geresnį ciklomatinį sudėtingumą, o tai reiškia lengvesnį funkcionalumo testavimą. Taip pat keliais punktais pagerėjo sistemos palaikomumo indeksas. Klasių susietumas šiek tiek pablogėjo, tačiau šios klasės dydis sumažėjo daugiau nei aštuonis kartus, nes didelė dalis funkcionalumo buvo iškelta į atskiras klases.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 128 pav. pateiktas grafikas. Grafike pavaizdutoa, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis patobulinimas su visais elementų kiekiais yra šiek tiek lėtesnis nei šiuo metu sistemoje įgyvendintas būdas.



128 pav. Aukšto lygio įvykių formavimo naudojant *Chain of Responsibility* šabloną greitaveika

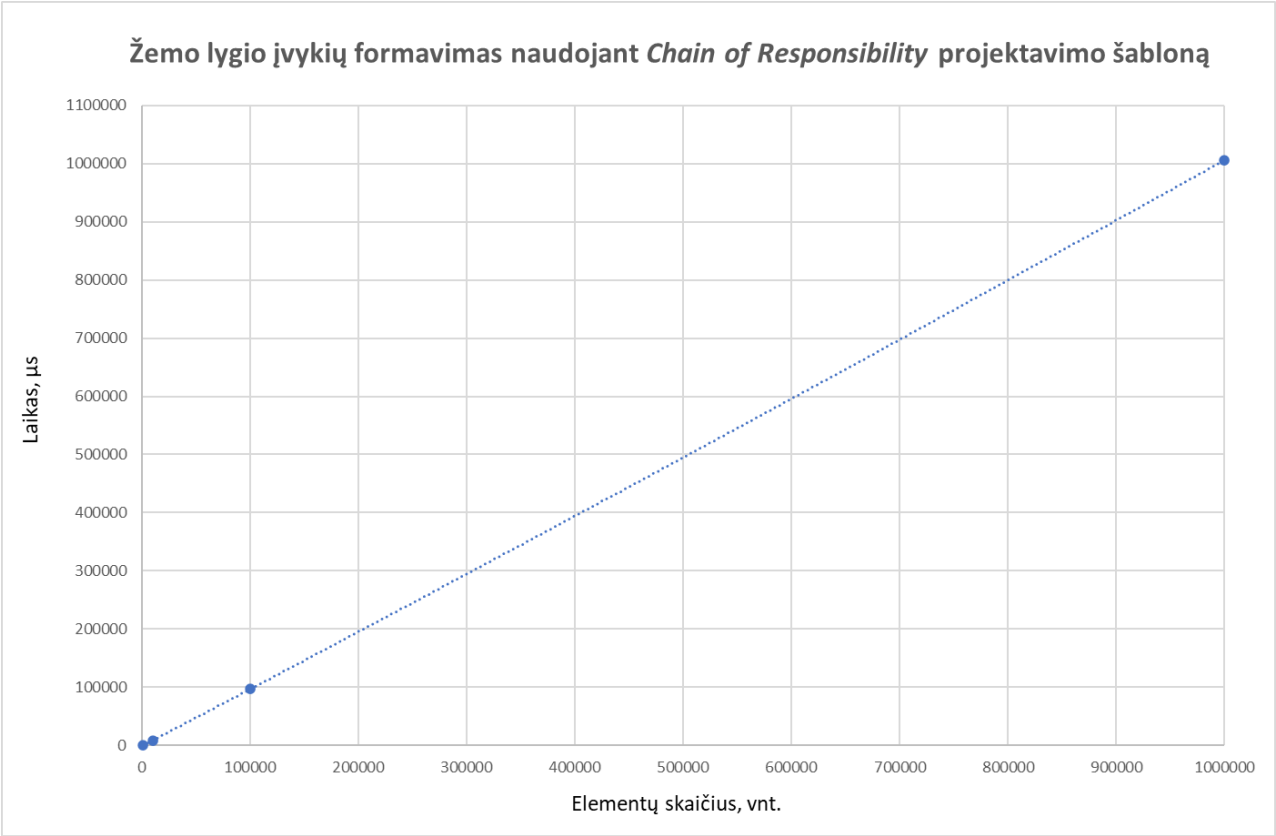
Įgyvendinus 3.4.2 skyrelyje aprašytą žemo lygio įvykių patobulinimą, *Visual Studio* apskaičiavo 129 pav. pateiktas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
LowLevelEventsAbstractor	59	65	1	18	247

129 pav. Žemo lygio įvykių formavimo naudojant *Chain of Responsibility* šabloną kodo metrikos

Lyginant su jau sistemoje įgyvendintu žemo lygio įvykių formavimu, šis patobulinimas turi geresnį ciklomatinį sudėtingumą, tačiau jis vis dar yra per didelis, nes nebuvo galima iškelti viso funkcionalumo į kitas klases. Palaikomumo indeksas ir klasių susietumas taip pat pablogėjo keliais punktais. Kita vertus, klasės dydis dvigubai sumažėjo.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gauats 130 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis patobulinimas taip pat yra lėtesnis nei šiuo metu sistemoje įgyvendintas būdas.



130 pav. Žemo lygio įvykių formavimo naudojant *Chain of Responsibility* šabloną greitaveika

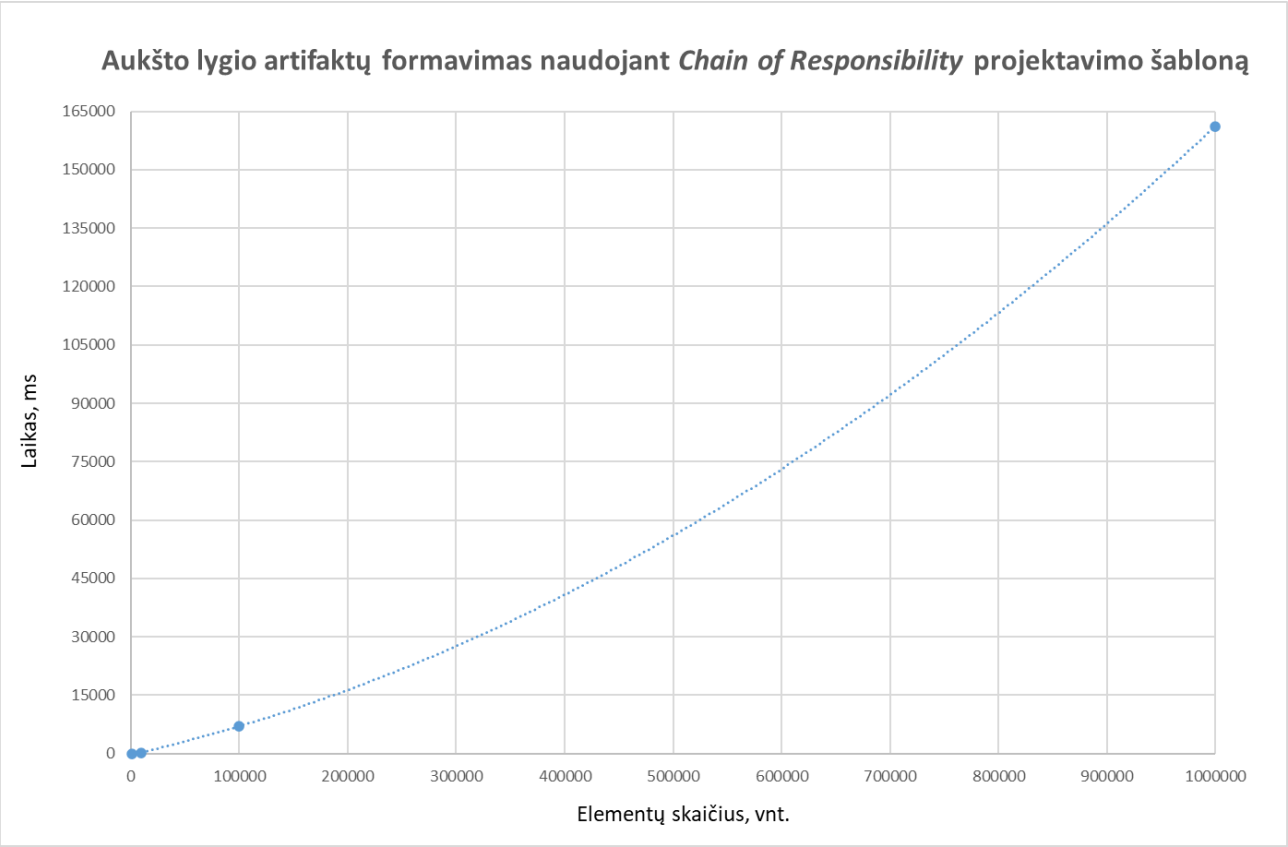
Įgyvendinus 3.4.2 skyrelyje aprašytą aukšto lygio artifaktų patobulinimą, *Visual Studio* apskaičiavo 131 pav. parodytas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
HighLevelArtefactsAbstractor	63	15	1	20	96

131 pav. Aukšto lygio artifaktų formavimo naudojant *Chain of Responsibility* šabloną kodo metrikos

Lyginant su jau sistemoje įgyvendintu aukšto lygio artifaktų formavimu, šis patobulinimas turi daug geresnį ciklomatinį sudėtingumą, kuris neviršija rekomenduojamos ribos. Palaikomumo indeksas nepasikeitė, nes klasių susietumas pablogėjo keliais punktais. Klasės eilučių skaičius taip pat sumažėjo daugiau nei šešis kartus.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 132 pav. pateiktas grafikas. Grafike pavaizdutoa, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Šis patobulinimas beveik visais elementų kiekiais yra šiek tiek lėtesnis nei šiuo metu sistemoje įgyvendintas būdas. Dirbant su mažais elementų skaičiais, skirtumas tarp šiuo metu sistemoje įgyvendinto būdo ir šio patobulinimo yra nežymus.



132 pav. Aukšto lygio artifaktų formavimo naudojant *Chain of Responsibility* šabloną greitaveika

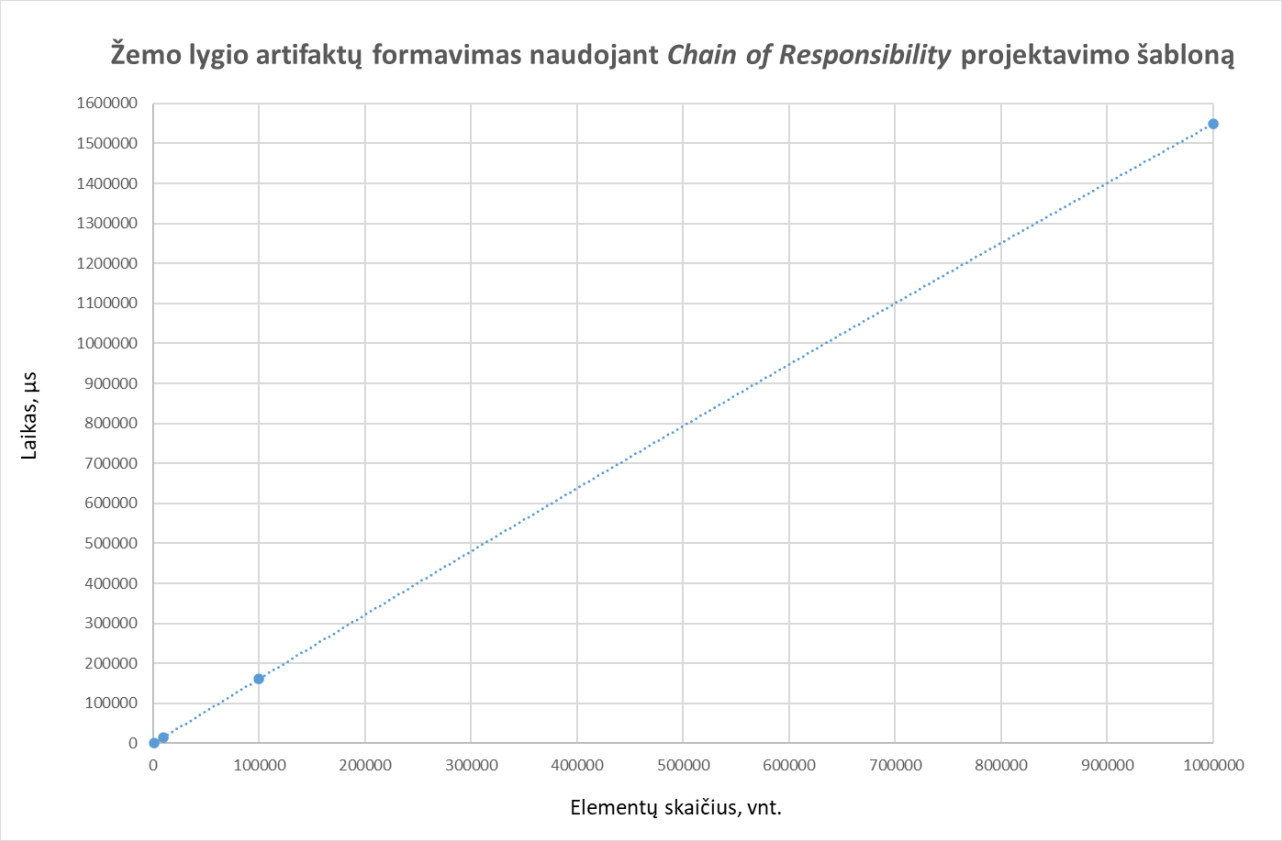
Įgyvendinus 3.4.2 skyrelyje aprašytą žemo lygio artifaktų patobulinimą, *Visual Studio* apskaičiavo 133 pav. pateiktas metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
LowLevelArtefactsAbstractor	68	17	1	10	73

133 pav. Žemo lygio artifaktų formavimo naudojant *Chain of Responsibility* šabloną kodo metrikos

Lyginant su jau sistemoje įgyvendintu žemo lygio artifaktų formavimu, šis patobulinimas turi du kartus geresnį ciklomatinį sudėtingumą, kuris dabar neviršija rekomenduojamos ciklomatinio sudėtingumo ribos. Klasės palaikomumas taip pat pagerėjo keliais punktais, dėl ko palengvėja sistemos palaikymas. Taip pat pavyko sumažinti klasės dydį beveik trimis kartais.

Atlikus šio patobulinimo greitaveikos testavimą, buvo gautas 134 pav. pateiktas grafikas. Grafike pavaizduota, kad egzistuoja kvadratinė priklausomybė tarp elementų skaičiaus ir vykdymo laiko. Tiek šis patobulinimas, tiek šiuo sistemoje įgyvendintas būdas su visais elementų kiekiais dirba panašiai.



134 pav. Žemo lygio artifaktų formavimo naudojant *Chain of Responsibility* šabloną greitaveika

4.4.3. Rezultatų apibendrinimas

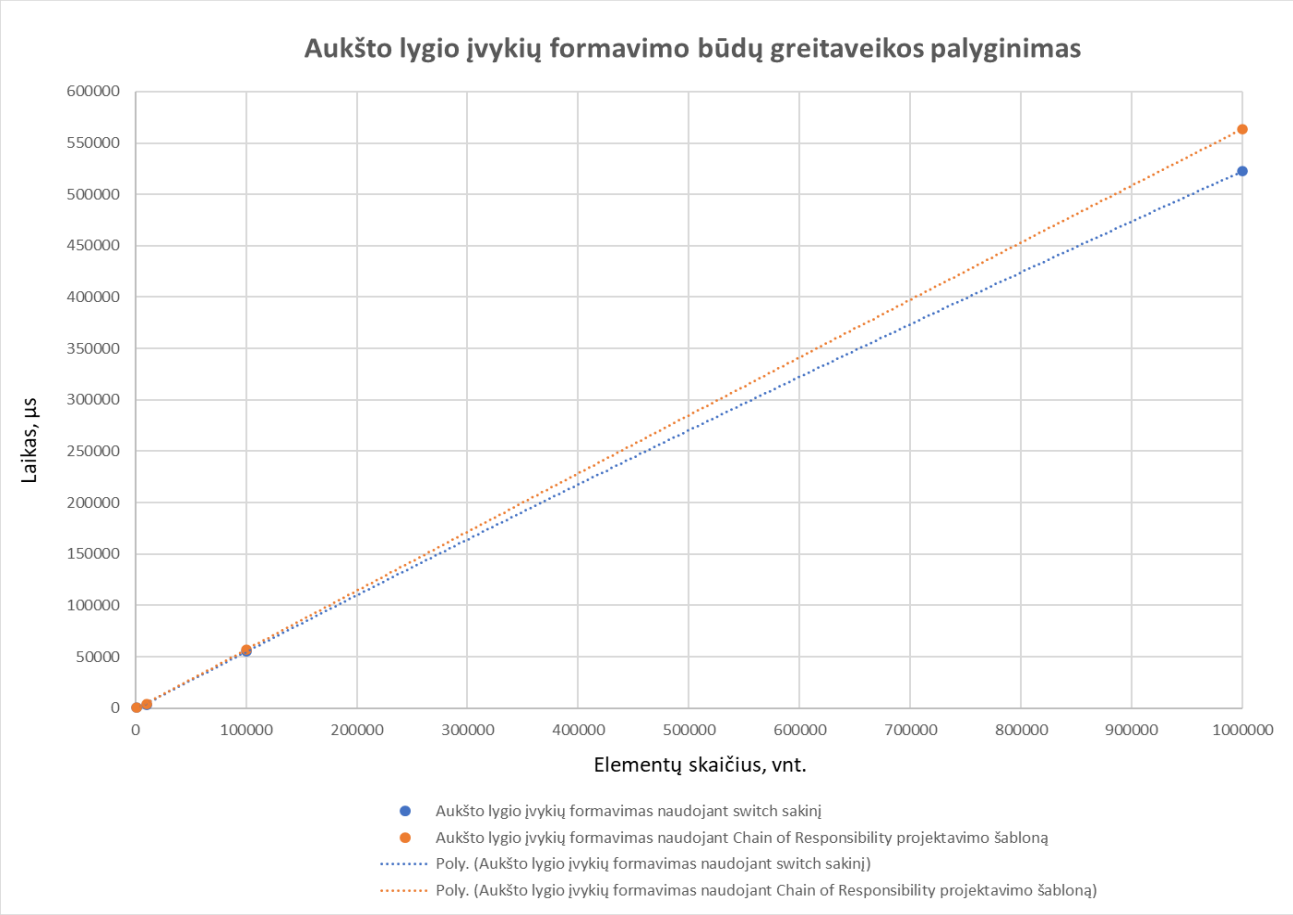
Aukšto lygio įvykių formavimo būdų kodo metrikų įvertinimai yra pateikti 7 priede, o greitaveikos tyrimo rezultatai – 8 priede.

Geriausias aukšto lygio įvykių formavimo būdas pagal kiekvieną kodo metriką yra pateikiamas 7 lentelėje. Geriausias būdas pagal metrikas yra *Chain of Responsibility* šablono naudojimas. Jis turi keliais punktais geresnį palaikomumo indeksą, bet didžiausias šio patobulinimo privalumas yra mažas ciklomatinis sudėtingumas, kuris yra 11 kartų mažesnis. Klasių susietumas yra šiek tiek blogesnis, bet tai nėra didelis trūkumas. Taip pat ženkliai sumažėjo klasės dydis, tačiau bendras sistemos kodo eilučių skaičius išaugo, nes reikėjo sukurti daug naujų klasių, įgyvendinančių projektavimo šabloną.

7 lentelė. Geriausias aukšto lygio įvykių formavimo būdas pagal kodo metrikas

Kodo metrika	**Geriausias būdas**
Palaikomumo indeksas	Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Ciklomatinis sudėtingumas	Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Paveldėjimo medžio gylis	Abu vienodi.
Klasių susietumas	Aukšto lygio įvykių formavimas naudojant *switch* sakinį.
Kodo eilučių skaičius	Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną.

Pagal greitaveiką (žr. 135 pav.), greičiausias aukšto lygio įvykių formavimo būdas yra šiuo metu sistemoje naudojamas *switch* sakinys. Dirbant su 1000 elementų, patobulinimas, naudojantis *Chain of Responsibility* projektavimo šabloną, yra 14% lėtesnis, dirbant su 10000 elementų yra 8% lėtesnis, dirbant su 100000 elementų yra 4% lėtesnis, o dirbant su 1 milijonu įvykių yra 8% lėtesnis. Abu būdai naudoja tiek pat atminties.



135 pav. Aukšto lygio įvykių formavimo būdų greitaveikos palyginimas

Žemo lygio įvykių formavimo būdų kodo metrikų įvertinimai yra pateikti 9 priede, o greitaveikos tyrimo rezultatai – 10 priede.

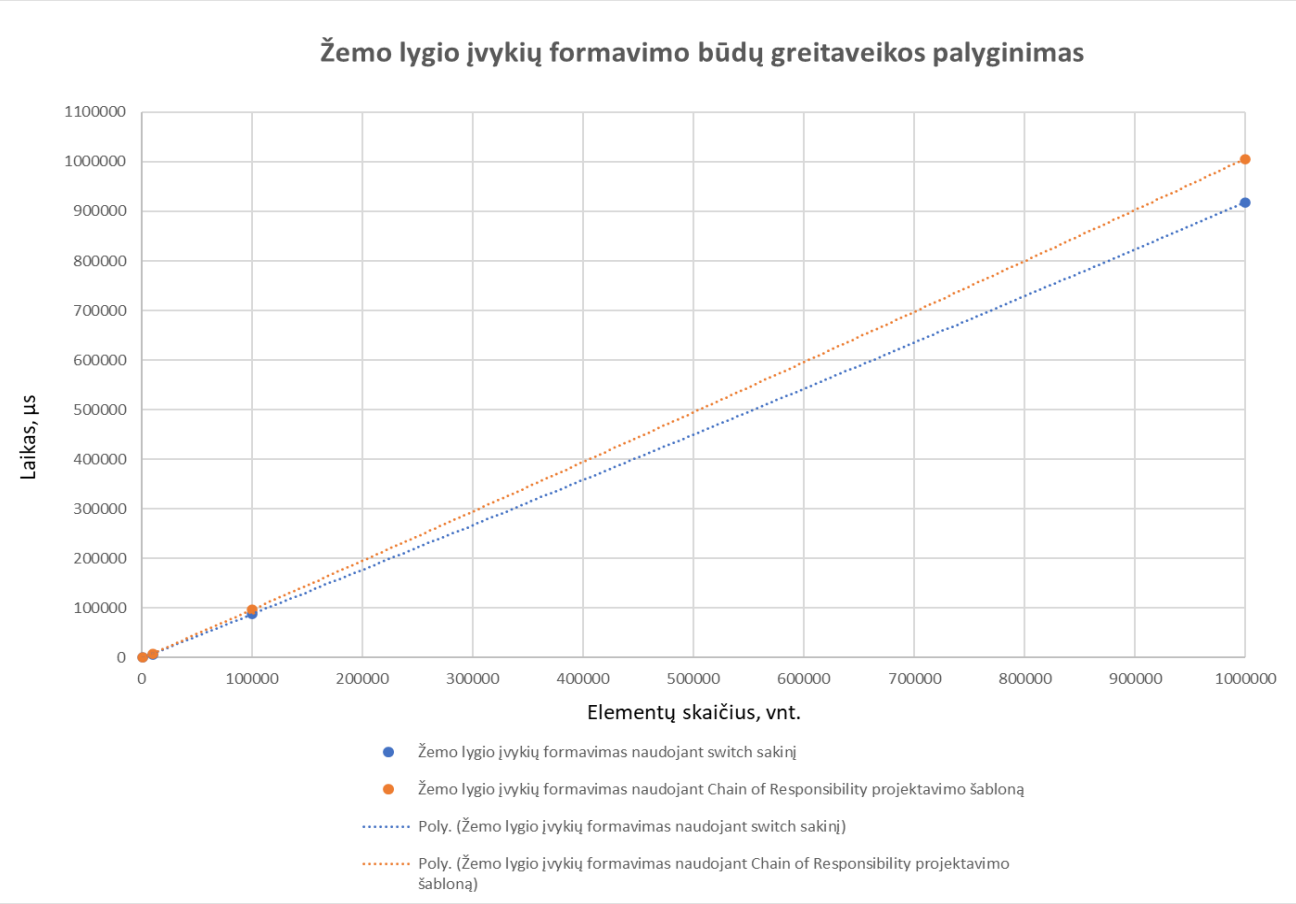
Geriausias žemo lygio įvykių formavimo būdas pagal kiekvieną kodo metriką yra pateikiamas 8 lentelėje. Geriausias būdas pagal metrikas taip pat yra *Chain of Responsibility* šablono naudojimas. Jis turi šiek tiek blogesnį palaikomumo indeksą ir klasių susietumą, tačiau ženklus ciklomatinio sudėtingumo pagerėjimas leidžia lengviau ištestuoti funkcionalumą. Klasė taip pat sumažėjo du kartus, nes funkcionalumas buvo iškeltas į naujas klases.

8 lentelė. Geriausias žemo lygio įvykių formavimo būdas pagal kodo metrikas

Kodo metrika	**Geriausias būdas**
Palaikomumo indeksas	Žemo lygio įvykių formavimas naudojant *switch* sakinį.
Ciklomatinis sudėtingumas	Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Paveldėjimo medžio gylis	Abu vienodi.

Klasių susietumas	Žemo lygio įvykių formavimas naudojant *switch* sakinį.
Kodo eilučių skaičius	Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną.

Pagal greitaveiką (žr. 136 pav.), greičiausias žemo lygio įvykių formavimo būdas yra dabar sistemoje naudojamas *switch* sakinys. Dirbant su 1000 elementų, patobulinimas, naudojantis *Chain of Responsibility* projektavimo šabloną, yra 23% lėtesnis, dirbant su 10000 elementų yra 14% lėtesnis, dirbant su 100000 elementų yra 10% lėtesnis, o dirbant su 1 milijonu įvykių yra 9% lėtesnis. Abu būdai naudoja tiek pat atminties.



136 pav. Žemo lygio įvykių formavimo būdų greitaveikos palyginimas

Aukšto lygio artifaktų formavimo būdų kodo metrikų įvertinimai yra pateikti 11 priede, o greitaveikos tyrimo rezultatai – 12 priede.

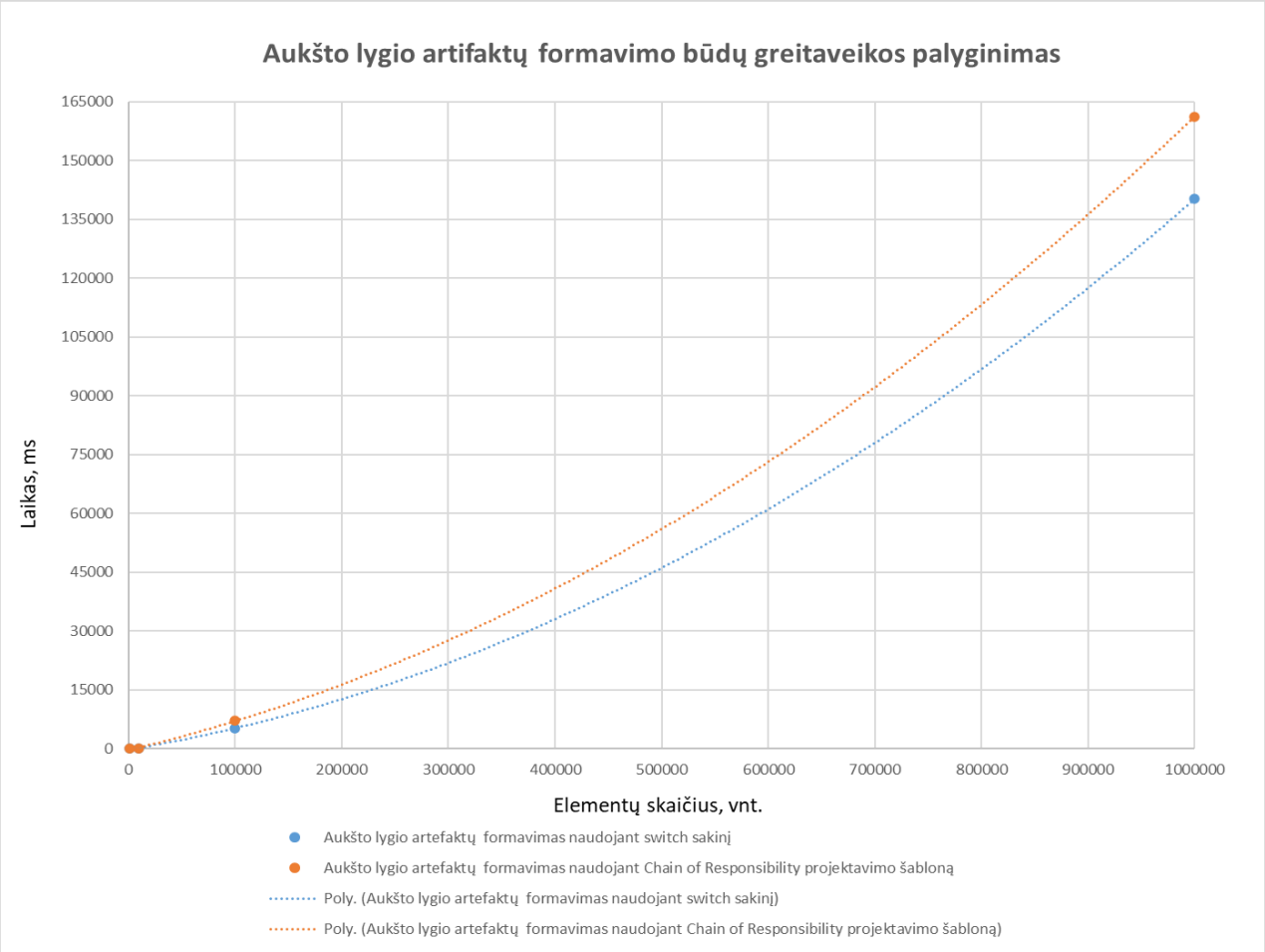
Geriausias aukšto lygio artifaktų formavimo būdas pagal kiekvieną kodo metriką yra pateikiamas 9 lentelėje. Geriausias būdas pagal metrikas yra *Chain of Responsibility* šablono naudojimas dėl ženkliai geresnio ciklomatinio sudėtingumo. Šis patobulinimas taip pat turi beveik šešis kartus mažiau kodo eilučių. Kita vertus, klasių susietumas yra blogesnis, o palaikomumo indeksas liko nepakitęs.

9 lentelė. Geriausias aukšto lygio artifaktų formavimo būdas pagal kodo metrikas

Kodo metrika	**Geriausias būdas**
Palaikomumo indeksas	Abu vienodi.

Ciklomatinis sudėtingumas	Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Paveldėjimo medžio gylis	Abu vienodi.
Klasių susietumas	Aukšto lygio artifaktų formavimas naudojant *switch* sakinį.
Kodo eilučių skaičius	Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną.

Pagal greitaveiką (žr. 137 pav.), greičiausias aukšto lygio artifaktų formavimo būdas yra šiuo metu sistemoje naudojamas *switch* sakinys. Dirbant su 1000 elementų, patobulinimas, naudojantis *Chain of Responsibility* projektavimo šabloną, užtrunka tiek pat laiko, kiek ir šiuo metu sistemoje įgyvendintas būdas, dirbant su 10000 elementų yra 6% lėtesnis, dirbant su 100000 elementų yra 34% lėtesnis, o dirbant su 1 milijonu įvykių yra 15% lėtesnis. Abu būdai naudoja tiek pat atminties.



137 pav. Aukšto lygio artifaktų formavimo būdų greitaveikos palyginimas

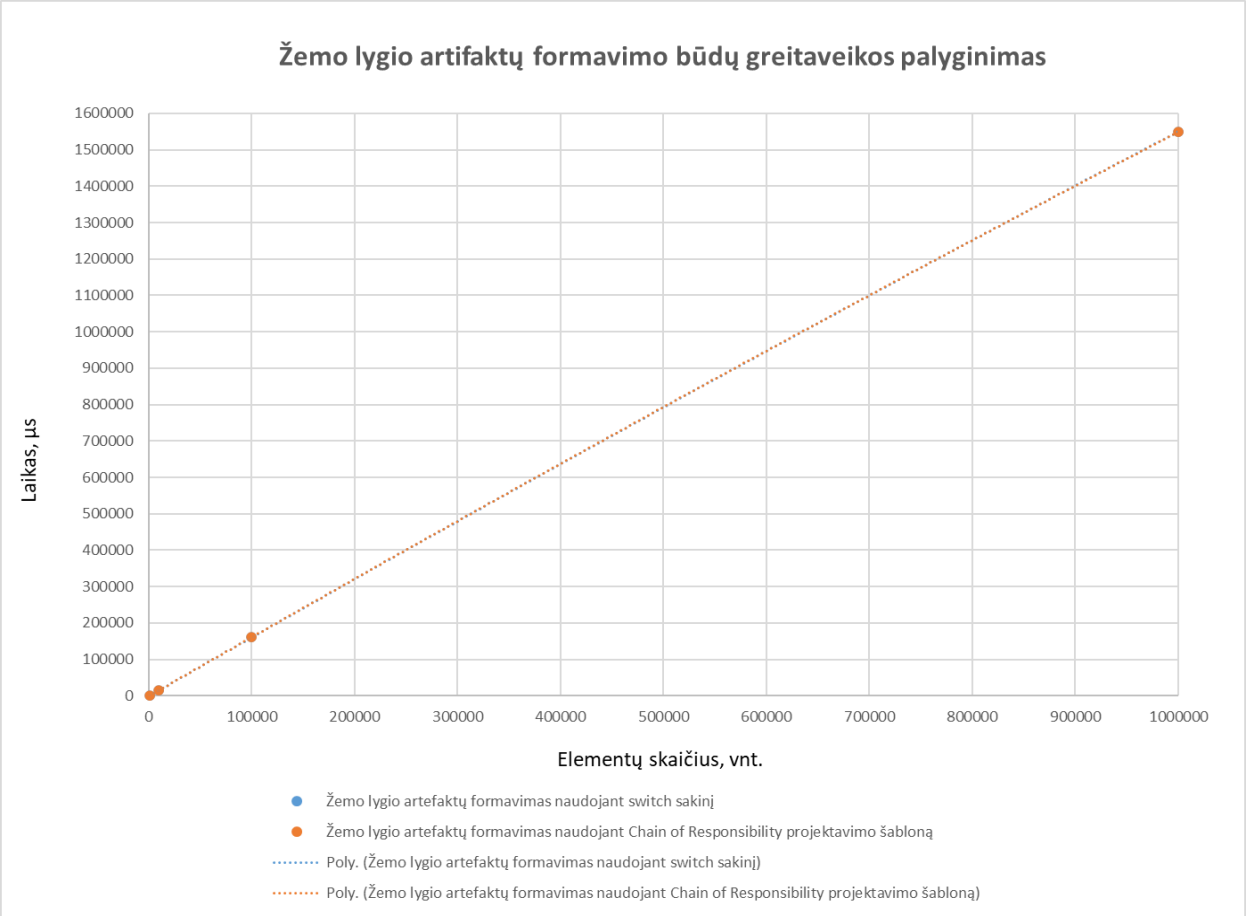
Žemo lygio artifaktų formavimo būdų kodo metrikų įvertinimai yra pateikti 13 priede, o greitaveikos tyrimo rezultatai – 14 priede.

Geriausias žemo lygio artifaktų formavimo būdas pagal kiekvieną kodo metriką yra pateikiamas 10 lentelėje. Geriausias būdas pagal visas kodo metrikas yra *Chain of Responsibility* šablono naudojimas. Jis turi šešiais punktais geresnį palaikomumo indeksą, dvigubai mažesnį ciklomatinį sudėtingumą, šiek tiek mažesnį klasių susietumą bei yra beveik tris kartus mažesnis.

10 lentelė. Geriausias žemo lygio artifaktų formavimo būdas pagal kodo metrikas

Kodo metrika	**Geriausias būdas**
Palaikomumo indeksas	Žemo lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Ciklomatinis sudėtingumas	Žemo lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Paveldėjimo medžio gylis	Abu vienodi.
Klasių susietumas	Žemo lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną.
Kodo eilučių skaičius	Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną.

Pagal greitaveiką (žr. 138 pav.), abu žemo lygio artifaktų formavimo būdai yra labai panašūs. Jie skiriasi tik 1% greičio atžvilgiu dirbant su visais elementų skaičiais. Abu būdai naudoja tiek pat atminties.



138 pav. Žemo lygio artifaktų formavimo būdų greitaveikos palyginimas

Įvertinus kodo metrikas ir greitaveikos palyginimus, *Chain of Responsibility* projektavimo šabloną naudojantys abstrakcijos formavimo būdai yra geresni. Nors greičio atžvilgiu jie ir nusileidžia *switch* sakinį naudojantiems metodams, tačiau projektavimo šablonas ženkliai palengvina kodo testavimo ir keitimo galimybes. Tai leistų ir toliau tobulinti sistemą įdedant mažiau pastangų.

Galiausiai, pirma hipotezė yra teisinga, nes beveik visų patobulinimų, naudojančių *Chain of Responsibility* projektavimo šabloną, kodo metrikos yra geresnės. Tai akivaizdžiai galima pamatyti žiūrint į ciklomatinį sudėtingumą ir kodo eilučių skaičių. Klasių susietumas šiek tiek pablogėjo, nes reikėjo naudoti papildomų klasių. Palaikomumo indeksas pagerėjo formuojant aukšto lygio įvykius bei žemo lygio artifaktus, liko toks pat formuojant aukšto lygio artifaktus ir sumažėjo formuojant žemo lygio įvykius. Antra hipotezė taip pat yra teisinga, nes per atitinkamo metodo atskaitos taškus galima nubrėžti antros eilės daugianarį.

4.5. Geriausių patobulinimų rezultatai

Po geriausių patobulinimų įgyvendinimo, *Visual Studio* apskaičiavo 139 pav. parodytas visos sistemos metrikas.

Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Source code
EventTimelineReconstruction (Release)	82	1,626	9	293	10,228

139 pav. Kompiuterio įvykių laike sistemos metrikos po geriausių patobulinimų įgyvendinimo

Lyginant su ankstesne sistemos versija, palaikomumo indeksas pagerėjo vienu punktu, tačiau kitos metrikos pablogėjo. Taip yra dėl to, kad čia yra pateikiama kiekvienos metrikos suma, o projektavimo šablonų naudojimas privertė sukurti daug naujų klasių. Norint gauti tikslesnius įvertinimus, galima apskaičiuoti kiekvienos klasės vidutinę metriką. Prieš patobulinimus, kompiuterio įvykių laike atkūrimo sistemą sudarė 95 klasės. Apskaičiuotos sistemos vidutinės klasės metrikos yra pateiktos 11 lentelėje. Palaikomumo indeksas ir paveldėjimo medžio gylis nėra įtraukiamas, nes jie nepriklauso nuo klasių skaičiaus.

11 lentelė. Vidutinė klasės kodo metrika prieš patobulinimus

Kodo metrika	**Vidutinė klasės kodo metrika**
Ciklomatinis sudėtingumas	15,38
Klasių susietumas	2,72
Kodo eilučių skaičius	98,76

Po patobulinimų, kompiuterio įvykių laike atkūrimo sistemą sudaro 122 klasės. Apskaičiuotos patobulintos sistemos vidutinės klasės metrikos yra pateiktos 12 lentelėje.

12 lentelė. Vidutinė klasės kodo metrika po patobulinimų

Kodo metrika	**Vidutinė klasės kodo metrika**
Ciklomatinis sudėtingumas	13,33
Klasių susietumas	2,4
Kodo eilučių skaičius	83,84

Vidutinės klasių kodo metrikos rodo, kad įgyvendinti patobulinimai padėjo sumažinti vidutinį klasės ciklomatinį sudėtingumą dviem punktais, klasių susietumas vidutiniškai sumažėjo trimis dešimtosiomis, o kodo eilučių skaičius klasėje sumažėjo 15 eilučių. Pasirinkti sistemos patobulinimo būdai leido pagerinti sistemos palaikymą, testavimo galimybes bei greitaveiką.

Išvados

1. Įvykių laike atkūrimo srities analizės metu buvo nustatytos didelio įvykių kiekio bei sudėtingos skaitmeninių incidentų analizės problemos, kurios neleidžia efektyviai atlikti skaitmeninių tyrimų.
2. Svarbiausi reikalavimai yra susiję su sistemos greitaveika, išplečiamumu ir panaudojamumu, kad sistema galėtų pagreitinti ir palengvinti skaitmeninius tyrimus.
3. Kompiuterio įvykių laike atkūrimo sistema buvo sukurta pagal MVVM architektūrą, nes ją yra paprasta įgyvendinti pasirinktam WPF projekto tipui, skirtam kurti naudotojo sąsają *Windows* operacinei sistemai.
4. Geriausias įvykių importavimo iš failo būdas yra pirminėje sistemos versijoje realizuotas paprastas *foreach* ciklas, nes jis gali būti iškviestas asinchroniškai neužblokuodamas pagrindinės UI gijos, yra antras pagal greitumą iš visų patobulinimų bei turi panašias kodo metrikas į kitus metodus.
5. Geriausias darbo išsaugojimo patobulinimas yra *foreach* ciklas konvertuojant *List* į *Span* tipo objektą, nes jis yra greičiausias iš visų tirtų patobulinimų, o pagal kodo metrikas smarkiai nenusileidžia kitiems patobulinimams.
6. Geriausias darbo įkėlimo būdas yra *Factory* projektavimo šablono naudojimas, nes šablono naudojimas leido sumažinti kodo kartojimą, dėl ko ženkliai pagerėjo kodo metrikos, bei leido pagreitinti įkėlimo procesą.
7. Geriausias abstrakcijos lygmenų formavimo būdas yra *Chain of Responsibility* projektavimo šablono naudojimas, nes šablono naudojimas ženkliai sumažino ciklomatinį sudėtingumą, nors dėl to šiek tiek sulėtėjo abstrakcijos lygių formavimas.

Literatūros sąrašas

[1] F. Olajide, N. Savage, D. Ndzi ir H. Al-Sinani, „Forensic Live Response and Event Reconstruction Methods in Linux Systems“, *PGNET 2009: The 10th Annual PostGraduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting,* pp. 141-146, 2009.

[2] Y. Chabot, A. Bertaux, C. Nicolle ir M.-T. Kechadi, „A complete formalized knowledge representation model for advanced digital forensics timeline analysis“, *Digital Investigation,* nr. 11, pp. S95-S105, 2014.

[3] S. Bhandari ir V. Jusas, „An Abstraction Based Approach for Reconstruction of TimeLine in Digital Forensics“, *Symmetry,* t. 12, p. 104, 2020.

[4] S. Kalber, A. Dewald ir F. C. Freiling, „Forensic Application-Fingerprinting Based on File System Metadata“, įtraukta *2013 Seventh International Conference on IT Security Incident Management and IT Forensics,* Nuremberg, 2013.

[5] J. I. James, P. Gladyshev ir Y. Zhu, „Signature Based Detection of User Events for Post-Mortem Forensic Analysis“, *Digital Forensics and Cyber Crime,* pp. 96-109, 2011.

[6] J. I. James ir P. Gladyshev, „Automated inference of past action instances in digital investigations“, *International Journal of Information Security,* t. 14, pp. 249-261, 2015.

[7] M. N. Khan ir I. Wakeman, „Machine Learning for Post-Event Timeline Reconstruction“, 2006.

[8] M. N. Khan, C. Chatwin ir R. Young, „A Framework for Post-Event Timeline Reconstruction Using Neural Networks“, *Digital Investigation,* t. 4, nr. 3, pp. 146-157, 2007.

[9] J. Bang, B. Yoo, J. Kim ir S. Lee, „Analysis of Time Information for Digital Investigation“, įtraukta *2009 Fifth International Joint Conference on INC, IMS and IDC,* Washington, D.C., 2009.

[10] G. S. Cho, „An Intuitive Computer Forensic Method by Timestamp Changing Pattern“, įtraukta *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing,* Birmingham, 2014.

[11] S. Soltani, S. A. H. Seno ir H. S. Yazdi, „Event Reconstruction Using Temporal Pattern of File System Modification“, *IET Information Security,* t. 13, pp. 201-212, 2019.

[12] S. A. Awawdeh, I. Baggili, A. Marrington ir F. Iqbal, „CAT Record (Computer Activity Timeline Record): A Unified Agent Based Approach for Real Time Computer Forensic Evidence Collection“, *2013 8th International Workshop on Systematic Approaches to Digital Forensics Engineering (SADFE),* pp. 1-8, 2013.

[13] F. Buchholz ir C. Falk, „Design and Implementation of Zeitline: a Forensic Timeline Editor“, įtraukta *Refereed Proceedings of the 5th Annual Digital Forensic Research Workshop,* New Orleans, 2005.

[14] C. Hargreaves ir J. Patterson, „An Automated Timeline Reconstruction Approach for Digital Forensic Investigations“, *Digital Investigation,* t. 9, pp. S69-S79, 2012.

[15] K. Guðjónsson, „Mastering the Super Timeline With log2timeline“, SANS Institute, 2010.

[16] M. Debinski, F. Breitinger ir P. Mohan, „Timeline2GUI: A Log2Timeline CSV Parser and Training Scenarios“, *Digital Investigation,* t. 28, pp. 34-43, 2019.

[17] G. S. Cho, „A Computer Forensic Method for Detecting Timestamp Forgery in NTFS“, *Computers & Security,* t. 34, pp. 36-46, 2013.

[18] A. Marrington, I. Baggili, G. Mohay ir A. Clark, „CAT Detect (Computer Activity Timeline Detection): A Tool for Detecting Inconsistency in Computer Activity Timelines“, *Digital Investigation,* t. 8, pp. S52-S61, 2011.

[19] B. Wagner ir P. Kulikov, „Iteration statements - for, foreach, do, and while“, Microsoft Learn, 13 02 2023. [Tinkle]. Prieiga: https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements. [Kreiptasi 22 03 2023].

[20] SingleAccretion, „Use normal VNs when adding NRE exception sets by SingleAccretion · Pull Request #76639 · dotnet/runtime“, GitHub, 07 10 2022. [Tinkle]. Prieiga: https://github.com/dotnet/runtime/pull/76639. [Kreiptasi 22 03 2023].

[21] „File.ReadAllLines Method (System.IO) | Microsoft Learn“, [Tinkle]. Prieiga: https://learn.microsoft.com/en-us/dotnet/api/system.io.file.readalllines?view=net-7.0. [Kreiptasi 08 04 2023].

[22] „File.ReadLines Method (System.IO) | Microsoft Learn“, [Tinkle]. Prieiga: https://learn.microsoft.com/en-us/dotnet/api/system.io.file.readlines?redirectedfrom=MSDN&view=net-7.0#System_IO_File_ReadLines_System_String_. [Kreiptasi 08 04 2023].

[23] „CollectionsMarshal Class (System.Runtime.InteropServices) | Microsoft Learn“, [Tinkle]. Prieiga: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.collectionsmarshal?source=recommendations&view=net-7.0. [Kreiptasi 08 04 2023].

[24] „MemoryMarshal Class (System.Runtime.InteropServices) | Microsoft Learn“, [Tinkle]. Prieiga: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.memorymarshal?view=net-7.0. [Kreiptasi 08 04 2023].

[25] „Unsafe.IsAddressLessThan<T>(T, T) Method (System.Runtime.CompilerServices) | Microsoft Learn“, [Tinkle]. Prieiga: https://learn.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.unsafe.isaddresslessthan?view=net-8.0. [Kreiptasi 11 04 2023].

Priedai

1 priedas. Įvykių importavimo iš failo patobulinimų kodo metrikos

13 lentelė. Įvykių importavimo iš failo patobulinimų kodo metrikos

Patobulinimas	**Palaikomumo indeksas**	**Ciklomatinis sudėtingumas**	**Paveldėjimo medžio gylis**	**Klasių susietumas**	**Kodo eilučių skaičius**
Paprastas *for* ciklas	61	10	1	13	104
Nesaugus *for* ciklas naudojant *Span* iš *List*	61	10	1	15	105
Nesaugus *for* ciklas naudojant *Span* iš masyvo	61	10	1	15	105
Nesaugus *for* ciklas naudojant *MemoryMarshal*	61	10	1	15	106
Lygiagretus *for* ciklas	61	9	1	16	108
Paprastas *foreach* ciklas	61	10	1	13	110
Paprastas *foreach* ciklas naudojant LINQ	61	9	1	14	110
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	61	10	1	15	110
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	61	10	1	15	110
Lygiagretus *foreach* ciklas	61	9	1	17	108
Lygiagretus asinchroninis *foreach* ciklas	61	9	1	20	111
Paprastas *while* ciklas	60	10	1	14	111
Nesaugus *while* ciklas naudojant *MemoryMarshal*	60	10	1	15	114
Lygiagrečios užklausos	61	9	1	16	108

2 priedas. Įvykių importavimo iš failo patobulinimų greitaveikos tyrimo rezultatai

14 lentelė. Įvykių importavimo iš failo patobulinimų greitaveikos tyrimo rezultatai

Patobulinimas	**Elementų skaičius**	**Vidurkis**	**Santykis**	**Vieta**	**Paskirta atmintis**	**Santykis**
Lygiagretus *for* ciklas	1000	3.213 ms	0.76	1	4.26 MB	1
Lygiagrečios užklausos	1000	3.483 ms	0.83	2	4.26 MB	1
Paprastas *for* ciklas	1000	4.115 ms	0.98	3	4.26 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	1000	4.115 ms	0.98	3	4.26 MB	1
Nesaugus *for* ciklas naudojant *Span* iš *List*	1000	4.116 ms	0.98	3	4.26 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	1000	4.186 ms	1	3	4.26 MB	1
Nesaugus *for* ciklas naudojant *MemoryMarshal*	1000	4.196 ms	1	3	4.26 MB	1

Nesaugus *for* ciklas naudojant *Span* iš masyvo	1000	4.198 ms	1	3	4.26 MB	1
Paprastas *foreach* ciklas naudojant LINQ	1000	4.200 ms	1	3	4.26 MB	1
Nesaugus *while* ciklas naudojant *MemoryMarshal*	1000	4.201 ms	1	3	4.26 MB	1
Paprastas *foreach* ciklas	1000	4.205 ms	1	3	4.25 MB	1
Paprastas *while* ciklas	1000	4.237 ms	1.01	3	4.25 MB	1
Lygiagretus asinchroninis *foreach* ciklas	1000	5.354 ms	1.27	4	4.61 MB	1.08
Paprastas *while* ciklas su sąrašo dydžio inicializavimu	1000	5.473 ms	1.3	5	5.95 MB	1.4
Paprastas *foreach* ciklas su sąrašo dydžio inicializavimu	1000	5.581 ms	1.33	6	5.95 MB	1.4
Lygiagretus *foreach* ciklas	1000	5.925 ms	1.19	6	4.29 MB	1.01
Lygiagretus *foreach* ciklas su sąrašo dydžio inicializavimu	1000	6.279 ms	1.49	6	5.98 MB	1.41
Lygiagretus asinchroninis *foreach* ciklas su sąrašo dydžio inicializavimu	1000	6.848 ms	1.65	7	6.31 MB	1.48
Paprastas *while* ciklas	10000	60.351 ms	0.97	1	42.55 MB	1
Paprastas *foreach* ciklas	10000	63.294 ms	1	2	42.55 MB	1
Lygiagretus *foreach* ciklas	10000	66.226 ms	1	3	42.68 MB	1
Lygiagrečios užklausos	10000	67.704 ms	1.07	3	42.64 MB	1
Paprastas *foreach* ciklas su sąrašo dydžio inicializavimu	10000	68.882 ms	1.1	4	59.35 MB	1.39
Paprastas *while* ciklas su sąrašo dydžio inicializavimu	10000	69.142 ms	1.11	4	59.35 MB	1.39
Lygiagretus asinchroninis *foreach* ciklas	10000	69.188 ms	1.09	4	46.15 MB	1.08
Lygiagretus *for* ciklas	10000	69.861 ms	1.1	4	42.65 MB	1
Paprastas *foreach* ciklas naudojant LINQ	10000	71.613 ms	1.14	4	42.63 MB	1
Nesaugus *for* ciklas naudojant *Span* iš *List*	10000	71.941 ms	1.14	4	42.63 MB	1
Lygiagretus *foreach* ciklas su sąrašo dydžio inicializavimu	10000	71.998 ms	1.14	4	59.4 MB	1.4
Paprastas *for* ciklas	10000	72.386 ms	1.16	4	42.63 MB	1
Nesaugus *while* ciklas naudojant *MemoryMarshal*	10000	72.804 ms	1.16	4	42.58 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	10000	73.110 ms	1.17	4	42.63 MB	1
Nesaugus *for* ciklas naudojant *Span* iš masyvo	10000	73.222 ms	1.17	4	42.58 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	10000	73.636 ms	1.17	4	42.58 MB	1
Nesaugus *for* ciklas naudojant *MemoryMarshal*	10000	75.509 ms	1.21	5	42.58 MB	1

Lygiagretus asinchroninis *foreach* ciklas su sąrašo dydžio inicializavimu	10000	78.068 ms	1.25	6	62.94 MB	1.48
Lygiagretus *for* ciklas	100000	562.443 ms	0.91	1	425.75 MB	1
Lygiagrečios užklausos	100000	572.077 ms	0.92	1	425.72 MB	1
Paprastas *foreach* ciklas	100000	617.522 ms	1	2	424.95 MB	1
Paprastas *while* ciklas	100000	632.759 ms	1.02	3	424.95 MB	1
Paprastas *foreach* ciklas naudojant LINQ	100000	642.975 ms	1.04	4	425.71 MB	1
Nesaugus *for* ciklas naudojant *Span* iš *List*	100000	644.720 ms	1.04	4	425.72 MB	1
Paprastas *for* ciklas	100000	649.803 ms	1.05	4	425.72 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	100000	659.011 ms	1.07	4	425.72 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	100000	660.280 ms	1.07	4	425.48 MB	1
Nesaugus *for* ciklas naudojant *Span* iš masyvo	100000	660.842 ms	1.07	4	425.48 MB	1
Nesaugus *for* ciklas naudojant *MemoryMarshal*	100000	661.237 ms	1.07	4	425.48 MB	1
Nesaugus *while* ciklas naudojant *MemoryMarshal*	100000	665.441 ms	1.08	4	425.48 MB	1
Lygiagretus *foreach* ciklas	100000	666.169 ms	0.96	4	425.04 MB	1
Lygiagretus asinchroninis *foreach* ciklas	100000	698.257 ms	1.12	5	460.83 MB	1.08
Paprastas *foreach* ciklas su sąrašo dydžio inicializavimu	100000	728.716 ms	1.18	6	593.37 MB	1.4
Lygiagretus *foreach* ciklas su sąrašo dydžio inicializavimu	100000	734.525 ms	1.15	6	593.44 MB	1.4
Paprastas *while* ciklas su sąrašo dydžio inicializavimu	100000	742.547 ms	1.2	6	593.37 MB	1.4
Lygiagretus asinchroninis *foreach* ciklas su sąrašo dydžio inicializavimu	100000	809.178 ms	1.31	7	629.22 MB	1.48
Lygiagretus *for* ciklas	1000000	5,806.580 ms	0.94	1	4253.2 MB	1
Paprastas *foreach* ciklas	1000000	6,207.728 ms	1	2	4245.42 MB	1
Paprastas *while* ciklas	1000000	6,295.026 ms	1.01	3	4245.42 MB	1
Lygiagrečios užklausos	1000000	6,461.999 ms	1.04	4	4253.07 MB	1
Lygiagretus *foreach* ciklas	1000000	6,616.635 ms	1.07	5	4245.67 MB	1
Lygiagretus asinchroninis *foreach* ciklas	1000000	6,819.205 ms	1.1	6	4603.97 MB	1.08
Nesaugus *while* ciklas naudojant *MemoryMarshal*	1000000	7,203.269 ms	1.16	7	4252.68 MB	1
Paprastas *foreach* ciklas su sąrašo dydžio inicializavimu	1000000	7,238.691 ms	1.17	7	5933.51 MB	1.4
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	1000000	7,241.203 ms	1.17	7	4253.05 MB	1

Nesaugus *for* ciklas naudojant *Span* iš masyvo	1000000	7,255.631 ms	1.17	7	4252.68 MB	1
Paprastas *while* ciklas su sąrašo dydžio inicializavimu	1000000	7,299.027 ms	1.18	7	5933.51 MB	1.4
Paprastas *foreach* ciklas naudojant LINQ	1000000	7,332.917 ms	1.18	7	4253.05 MB	1
Nesaugus *for* ciklas naudojant *MemoryMarshal*	1000000	7,359.726 ms	1.19	7	4252.68 MB	1
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	1000000	7,382.555 ms	1.19	7	4252.68 MB	1
Nesaugus *for* ciklas naudojant *Span* iš *List*	1000000	7,513.893 ms	1.21	8	4253.05 MB	1
Paprastas *for* ciklas	1000000	7,535.345 ms	1.21	8	4253.05 MB	1
Lygiagretus asinchroninis *foreach* ciklas su sąrašo dydžio inicializavimu	1000000	7,893.338 ms	1.27	9	6292.01 MB	1.48
Lygiagretus *foreach* ciklas su sąrašo dydžio inicializavimu	1000000	8,447.781 ms	1.36	10	5934.03 MB	1.4

3 priedas. Darbo išsaugojimo patobulinimų kodo metrikos

15 lentelė. Darbo išsaugojimo patobulinimų kodo metrikos

Patobulinimas	**Palaikomumo indeksas**	**Ciklomatinis sudėtingumas**	**Paveldėjimo medžio gylis**	**Klasių susietumas**	**Kodo eilučių skaičius**
Paprastas *for* ciklas	67	5	1	9	38
Nesaugus *for* ciklas naudojant *Span* iš *List*	67	5	1	10	42
Nesaugus *for* ciklas naudojant *Span* iš masyvo	67	5	1	10	42
Nesaugus *for* ciklas naudojant *MemoryMarshal*	65	5	1	10	44
Paprastas *foreach* ciklas	72	11	1	10	64
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	68	5	1	11	38
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	68	5	1	10	38
Paprastas *while* ciklas	67	5	1	8	42
Nesaugus *while* ciklas naudojant *MemoryMarshal*	64	5	1	10	48

4 priedas. Darbo išsaugojimo patobulinimų greitaveikos tyrimo rezultatai

16 lentelė. Darbo išsaugojimo patobulinimų greitaveikos tyrimo rezultatai

Patobulinimas	**Elementų skaičius**	**Vidurkis**	**Santykis**	**Vieta**	**Paskirta atmintis**	**Santykis**
Nesaugus *for* ciklas naudojant *Span* iš *List*	1000	1.333 ms	0.76	1	828.37 KB	0.99
Nesaugus while ciklas naudojant MemoryMarshal	1000	1.341 ms	0.77	1	830.05 KB	0.99

Nesaugus *foreach* ciklas naudojant *Span* iš *List*	1000	1.346 ms	0.77	1	828.37 KB	0.99
Nesaugus *for* ciklas naudojant *Span* iš masyvo	1000	1.351 ms	0.77	1	830.05 KB	0.99
Nesaugus *for* ciklas naudojant *MemoryMarshal*	1000	1.367 ms	0.78	1	830.05 KB	0.99
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	1000	1.417 ms	0.8	1	830.05 KB	0.99
Paprastas *foreach* ciklas	1000	1.747 ms	1	2	840.75 KB	1
Paprastas while ciklas	1000	1.773 ms	1.02	2	840.53 KB	1
Paprastas *for* ciklas	1000	1.794 ms	1.03	2	838.72 KB	1
Nesaugus *for* ciklas naudojant *Span* iš *List*	10000	10.323 ms	0.68	1	8197.47 KB	0.99
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	10000	10.804 ms	0.72	2	8213.22 KB	0.99
Nesaugus *for* ciklas naudojant *MemoryMarshal*	10000	10.904 ms	0.73	2	8213.22 KB	0.99
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	10000	10.921 ms	0.72	2	8197.47 KB	0.99
Nesaugus while ciklas naudojant MemoryMarshal	10000	10.935 ms	0.72	2	8213.22 KB	0.99
Nesaugus *for* ciklas naudojant *Span* iš masyvo	10000	10.991 ms	0.71	2	8213.22 KB	0.99
Paprastas *for* ciklas	10000	14.274 ms	0.94	3	8294.06 KB	1
Paprastas while ciklas	10000	14.643 ms	0.96	4	8309.91 KB	1
Paprastas *foreach* ciklas	10000	15.178 ms	1	5	8310.1 KB	1
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	100000	102.053 ms	0.68	1	82324.56 KB	0.99
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	100000	104.354 ms	0.71	1	82168.3 KB	0.99
Nesaugus *for* ciklas naudojant *MemoryMarshal*	100000	104.752 ms	0.7	1	82324.56 KB	0.99
Nesaugus while ciklas naudojant MemoryMarshal	100000	104.816 ms	0.7	1	82324.56 KB	0.99
Nesaugus *for* ciklas naudojant *Span* iš *List*	100000	107.014 ms	0.72	1	82168.24 KB	0.99
Nesaugus *for* ciklas naudojant *Span* iš masyvo	100000	108.265 ms	0.73	1	82324.56 KB	0.99
Paprastas while ciklas	100000	146.290 ms	0.98	2	83289.09 KB	1
Paprastas *for* ciklas	100000	148.045 ms	1	2	83132.3 KB	1
Paprastas *foreach* ciklas	100000	148.711 ms	1	2	83289.35 KB	1
Nesaugus *foreach* ciklas naudojant *Span* iš *List*	1000000	1,129.461 ms	0.71	1	821874.45 KB	0.99
Nesaugus *for* ciklas naudojant *MemoryMarshal*	1000000	1,170.532 ms	0.73	1	823436.84 KB	0.99

Nesaugus *for* ciklas naudojant *Span* iš *List*	1000000	1,171.990 ms	0.74	1	821874.45 KB	0.99
Nesaugus *foreach* ciklas naudojant *Span* iš masyvo	1000000	1,186.239 ms	0.74	1	823436.84 KB	0.99
Nesaugus *while* ciklas naudojant *MemoryMarshal*	1000000	1,289.748 ms	0.81	2	823436.84 KB	0.99
Nesaugus *for* ciklas naudojant *Span* iš masyvo	1000000	1,358.512 ms	0.84	2	823436.84 KB	0.99
Paprastas *for* ciklas	1000000	1,544.689 ms	0.95	3	831569.38 KB	1
Paprastas *foreach* ciklas	1000000	1,598.121 ms	1	4	833131.51 KB	1
Paprastas *while* ciklas	1000000	1,688.073 ms	1.06	5	833131.34 KB	1

5 priedas. Darbo įkėlimo būdų kodo metrikos

17 lentelė. Darbo įkėlimo būdų kodo metrikos

Patobulinimas	**Palaikomumo indeksas**	**Ciklomatinis sudėtingumas**	**Paveldėjimo medžio gylis**	**Klasių susietumas**	**Kodo eilučių skaičius**
Darbo įkėlimas skaitant failą po vieną eilutę su *StreamReader*	61	31	1	24	322
Darbo įkėlimas naudojant *Factory* projektavimo šabloną	62	22	1	26	181

6 priedas. Darbo įkėlimo būdų greitaveikos tyrimo rezultatai

18 lentelė. Darbo įkėlimo būdų greitaveikos tyrimo rezultatai

Patobulinimas	**Elementų skaičius**	**Vidurkis**	**Santykis**	**Vieta**	**Paskirta atmintis**	**Santykis**
Darbo įkėlimas naudojant *Factory* projektavimo šabloną	1000	1.510 ms	0.77	1	1.61 MB	0.95
Darbo įkėlimas skaitant failą po vieną eilutę su *StreamReader*	1000	1.976 ms	1	2	1.69 MB	1
Darbo įkėlimas naudojant *Factory* projektavimo šabloną	10000	26.308 ms	0.86	1	15.95 MB	0.95
Darbo įkėlimas skaitant failą po vieną eilutę su *StreamReader*	10000	30.575 ms	1	2	16.71 MB	1
Darbo įkėlimas naudojant *Factory* projektavimo šabloną	100000	263.647 ms	0.85	1	160.29 MB	0.96
Darbo įkėlimas skaitant failą po vieną eilutę su *StreamReader*	100000	312.182 ms	1	2	167.81 MB	1
Darbo įkėlimas naudojant *Factory* projektavimo šabloną	1000000	3,075.339 ms	0.94	1	1598.28 MB	0.96
Darbo įkėlimas skaitant failą po vieną eilutę su *StreamReader*	1000000	3,303.308 ms	1	2	1673.41 MB	1

7 priedas. Aukšto lygio įvykių formavimo būdų kodo metrikos

19 lentelė. Aukšto lygio įvykių formavimo būdų kodo metrikos

Patobulinimas	Palaikomumo indeksas	Ciklomatinis sudėtingumas	Paveldėjimo medžio gylis	Klasių susietumas	Kodo eilučių skaičius
Aukšto lygio įvykių formavimas naudojant *switch* sakinį	66	44	1	10	308
Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	68	4	1	11	37

8 priedas. Aukšto lygio įvykių formavimo būdų greitaveikos tyrimo rezultatai

20 lentelė. Aukšto lygio įvykių formavimo būdų greitaveikos tyrimo rezultatai

Patobulinimas	Elementų skaičius	Vidurkis	Santykis	Vieta	Paskirta atmintis	Santykis
Aukšto lygio įvykių formavimas naudojant *switch* sakinį	1000	262.8 µs	1	1	293.47 KB	1
Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	1000	297.8 µs	1.14	2	293.47 KB	1
Aukšto lygio įvykių formavimas naudojant *switch* sakinį	10000	3,570.3 µs	1	1	2930.19 KB	1
Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	10000	3,860.2 µs	1.08	2	2930.19 KB	1
Aukšto lygio įvykių formavimas naudojant *switch* sakinį	100000	54,824.2 µs	1	1	29299.48 KB	1
Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	100000	56,864.8 µs	1.04	2	29298.47 KB	1
Aukšto lygio įvykių formavimas naudojant *switch* sakinį	1000000	522,544.4 µs	1	1	292976.09 KB	1
Aukšto lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	1000000	563,582.2 µs	1.08	2	292979.43 KB	1

9 priedas. Žemo lygio įvykių formavimo būdų kodo metrikos

21 lentelė. Žemo lygio įvykių formavimo būdų kodo metrikos

Patobulinimas	Palaikomumo indeksas	Ciklomatinis sudėtingumas	Paveldėjimo medžio gylis	Klasių susietumas	Kodo eilučių skaičius
Žemo lygio įvykių formavimas naudojant *switch* sakinį	63	104	1	12	555
Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	59	65	1	18	247

10 priedas. Žemo lygio įvykių formavimo būdų greitaveikos tyrimo rezultatai

22 lentelė. Žemo lygio įvykių formavimo būdų greitaveikos tyrimo rezultatai

Patobulinimas	Elementų skaičius	Vidurkis	Santykis	Vieta	Paskirta atmintis	Santykis
Žemo lygio įvykių formavimas naudojant *switch* sakinį	1000	401.2 µs	1	1	539.1 KB	1
Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	1000	493.3 µs	1.23	2	539.1 KB	1
Žemo lygio įvykių formavimas naudojant *switch* sakinį	10000	6,433.0 µs	1	1	5382.93 KB	1
Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	10000	7,328.4 µs	1.14	2	5382.91 KB	1
Žemo lygio įvykių formavimas naudojant *switch* sakinį	100000	87,520.4 µs	1	1	53821.21 KB	1
Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	100000	96,718.6 µs	1.1	2	53820.69 KB	1
Žemo lygio įvykių formavimas naudojant *switch* sakinį	1000000	918,478.6 µs	1	1	538196.31 KB	1
Žemo lygio įvykių formavimas naudojant *Chain of Responsibility* projektavimo šabloną	1000000	1,006,043.2 µs	1.09	2	538199.02 KB	1

11 priedas. Aukšto lygio artifaktų formavimo būdų kodo metrikos

23 lentelė. Aukšto lygio artifaktų formavimo būdų kodo metrikos

Patobulinimas	Palaikomumo indeksas	Ciklomatinis sudėtingumas	Paveldėjimo medžio gylis	Klasių susietumas	Kodo eilučių skaičius
Aukšto lygio artifaktų formavimas naudojant *switch* sakinį	63	94	1	14	610
Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną	63	15	1	20	96

12 priedas. Aukšto lygio artifaktų formavimo būdų greitaveikos tyrimo rezultatai

24 lentelė. Aukšto lygio artifaktų formavimo būdų greitaveikos tyrimo rezultatai

Patobulinimas	Elementų skaičius	Vidurkis	Santykis	Vieta	Paskirta atmintis	Santykis
Aukšto lygio artifaktų formavimas naudojant *switch* sakinį	1000	4.215 ms	1	1	647.59 KB	1
Aukšto lygio artifaktų formavimas naudojant *Chain of*	1000	4.218 ms	1	1	647.52 KB	1

Responsibility projektavimo šabloną						
Aukšto lygio artifaktų formavimas naudojant *switch* sakinį	10000	61.514 ms	1	1	6476.67 KB	1
Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną	10000	65.390 ms	1.06	2	6475.73 KB	1
Aukšto lygio artifaktų formavimas naudojant *switch* sakinį	100000	5,262.051 ms	1	1	64749.48 KB	1
Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną	100000	7,075.850 ms	1.34	2	64747 KB	1
Aukšto lygio artifaktų formavimas naudojant *switch* sakinį	1000000	140,130.315 ms	1	1	647635.51 KB	1
Aukšto lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną	1000000	161,202.489 ms	1.15	2	647634.57 KB	1

13 priedas. Žemo lygio artifaktų formavimo būdų kodo metrikos

25 lentelė. Žemo lygio artifaktų formavimo būdų kodo metrikos

Patobulinimas	**Palaikomumo indeksas**	**Ciklomatinis sudėtingumas**	**Paveldėjimo medžio gylis**	**Klasių susietumas**	**Kodo eilučių skaičius**
Žemo lygio artifaktų formavimas naudojant *switch* sakinį	62	40	1	11	208
Žemo lygio artifaktų formavimas naudojant Chain of Responsibility projektavimo šabloną	68	17	1	10	73

14 priedas. Žemo lygio artifaktų formavimo būdų greitaveikos tyrimo rezultatai

26 lentelė. Žemo lygio artifaktų formavimo būdų greitaveikos tyrimo rezultatai

Patobulinimas	**Elementų skaičius**	**Vidurkis**	**Santykis**	**Vieta**	**Paskirta atmintis**	**Santykis**
Žemo lygio artifaktų formavimas naudojant *switch* sakinį	1000	636.7 µs	1	1	1.41 MB	1
Žemo lygio artifaktų formavimas naudojant *Chain of Responsibility* projektavimo šabloną	1000	642.1 µs	1.01	1	1.41 MB	1
Žemo lygio artifaktų formavimas naudojant *switch* sakinį	10000	15,192.1 µs	1	1	14.04 MB	1
Žemo lygio artifaktų formavimas naudojant Chain of Responsibility projektavimo šabloną	10000	15,396.8 µs	1.01	1	14.04 MB	1

Žemo lygio artifaktų formavimas naudojant *switch* sakinį	100000	160,596.7 µs	1	1	140.38 MB	1
Žemo lygio artifaktų formavimas naudojant Chain of Responsibility projektavimo šabloną	100000	160,990.7 µs	1.01	1	140.38 MB	1
Žemo lygio artifaktų formavimas naudojant Chain of Responsibility projektavimo šabloną	1000000	1,548,926.6 µs	1	1	1403.81 MB	1
Žemo lygio artifaktų formavimas naudojant *switch* sakinį	1000000	1,549,939.1 µs	1	1	1403.81 MB	1