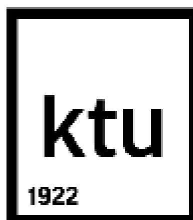


KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

VYTENIS SODAITIS

JAVA PROGRAMOS KODO ANALIZĖS NAUDOJANT SCRO ONTOLOGIJĄ GALIMYBIŲ TYRIMAS

Baigiamasis magistro projektas

Vadovas
doc. dr. R. Butkienė

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

JAVA PROGRAMOS KODO ANALIZĖS NAUDOJANT SCRO ONTOLOGIJĄ GALIMYBIŲ TYRIMAS

Baigiamasis magistro projektas

Informacinių sistemų inžinerijos studijų programa (kodas 621E15001)

Vadovas
doc. dr. R. Butkienė
2015-05-22

Recenzentas
doc. dr. A. Riškus
2015-05-22

Projektą atliko
Vytenis Sodaitis
2015-05-22

KAUNAS, 2015

KAUNO TECHNOLOGIJOS UNIVERSITETAS
INFORMATIKOS FAKULTETAS

(Fakultetas)

Vytenis Sodaitis

(Studento vardas, pavardė)

Informacinių sistemų inžinerijos studijų programa, 621E15001

(Studijų programos pavadinimas, kodas)

Baigiamojo projekto „Pavadinimas“
AKADEMINIO SĄŽININGUMO DEKLARACIJA

20 15 m. gegužės 25 d.
Kaunas

Patvirtinu, kad mano, **Vytenio Sodaičio**, baigiamasis projektas tema „Java programos kodo analizės naudojant SCRO ontologiją galimybių tyrimas“ yra parašytas visiškai savarankiškai ir visi pateikti duomenys ar tyrimų rezultatai yra teisingi ir gauti sąžiningai. Šiame darbe nei viena dalis nėra plagijuota nuo jokių spausdintinių ar internetinių šaltinių, visos kitų šaltinių tiesioginės ir netiesioginės citatos nurodytos literatūros nuorodose. Įstatymų nenumatytų piniginių sumų už šį darbą niekam nesu mokėjęs.

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

(vardą ir pavardę įrašyti ranka) *(parašas)*

Sodaitis, V. Research on Java program code analysis using SCRO ontology. *Final Degree Project of Master of Information Systems Engineering* / Supervisor Assoc. Prof. Dr. Rita Butkienė; Kaunas University of Technology, Faculty of Informatics.

Kaunas, 2015. 47 p.

SUMMARY

Program code analysis is especially important for developers who maintain and support software developed by other programmers, these days. Software code created by another author could be difficult to understand. Understanding it takes a lot of time, but the code must be used, edited, extender or reprogramed in another programming language.

Existing static code analysis tools were reviewed and as a result, it is found that while they are great at revealing bugs and making sure that code is covered by unit tests, they lack of capabilities to visualize code and make code comprehension easier.

That is why it was decided, that we have to develop a static Java code analysis tool, using ontologies. The most fitting ontology for the job is SCRO (Source Code Representation Ontology), as it is programming language agnostic, but supports most features that modern object oriented programming languages provide.

The resulting tool has no problems processing huge amounts of data, while doing it with reasonable speed. Analysis results then could be reviewed using various ontology tools, for example – Protégé. Using it, it is possible to query code and find out if specific design patterns are used, what packages, classes and methods depend on each other. Code elements can be represented either textually or graphically

TURINYS

Lentelių sąrašas 6
Paveikslų sąrašas 7
Terminų ir santrumpų žodynas 8
Įvadas 9
1. Java programos kodo analizės naudojant SCRO ontologiją galimybių tyrimo analizė 10
1.1. Analizės tikslas 10
1.2. Tyrimo objektas, sritis, problema 10
1.3. Programinio kodo statinės analizės proceso analizė 10
1.4. Vartotojų analizė 15
1.5. Esamų problemos sprendimo metodų analizė 15
1.6. Architektūros ir galimų įgyvendinimo priemonių variantų analizė 18
1.7. Panašių sprendimų ir galimų įgyvendinimo priemonių analizės išvados 20
1.8. Darbo tikslas ir uždaviniai 21
1.9. Siekiamo sprendimo apibrėžimas 22
1.10. Analizės išvados 22
2. Java programos kodo analizės eksperimentinio įrankio reikalavimų specifikacija ir analizė 23
2.1. Reikalavimų specifikacija 23
2.2. Dalykinės srities modelis 29
2.3. Vartotojo sąsajos reikalavimai 29
3. Java programos kodo analizės įrankio prototipo projektas 30
3.1. Sistemos architektūra 30
3.1.1. Reikalavimų analizė 31
3.1.2. Loginė visos sistemos architektūra 32
3.1.3. Vartotojo sąsajos klasių modelis 32
3.1.4. Veiklos logikos (valdymo ir esybių klasių) modelis 33
3.2. Sistemos elgsenos modelis 34
3.3. Realizacijos modelis 35
3.3.1. Programinių komponentų architektūra 35
4. Java programos kodo analizės įrankio eksperimentinio prototipo realizacija ir testavimas 37
4.1. Sprendimo realizacijos ir veikimo aprašas 37
4.2. Testavimo modelis, duomenys, rezultatai 39
5. Eksperimentinis Java programos kodo analizės įrankio, naudojančio SCRO ontologiją, galimybių tyrimas 41
6. Išvados 46
7. Literatūra 47

LENTELIŲ SĄRAŠAS

1.1 lentelė. Sprendimų palyginimas 21
2.1 lentelė. Panaudojimo atvejis 1 23
2.2 lentelė. Panaudojimo atvejis 2 24
2.3 lentelė. Panaudojimo atvejis 3 25
2.4 lentelė. Panaudojimo atvejis 4 26
2.5 lentelė. Panaudojimo atvejis 5 27
2.6 lentelė. Nefunkcinis reikalavimas 1 28
2.7 lentelė. Nefunkcinis reikalavimas 2 28
5.1 lentelė. Projektų apimtis 41
5.2 lentelė. Analizės trukmė 41
5.3 lentelė. Rezultatų failų dydžiai 42
5.4 lentelė. Rezultatų failų atidarymo trukmė 42
5.5 lentelė. TransOWL palyginimas su esamais sprendimais 44

PAVEIKSLŲ SĄRAŠAS

1.1 pav. Programos gyvavimo ciklo diagrama [9] 10
1.2 pav. Programinės įrangos gyvavimo ciklas 11
1.3 pav. Kodo analizės proceso pavyzdys 12
1.4 pav. Organizacinė struktūra 13
1.5 pav. Kodo analizavimo veiklos konceptai 13
1.6 pav. Programinės įrangos gyvavimo ciklo panaudojimo atvejai 14
1.7 pav. OBPCM struktūra [3] 16
1.8 pav. CQLinq užklausos pavyzdys [5] 17
1.9 pav. Sugeneruotas priklausomybių tarp projektų grafas [4] 18
1.10 pav. SCRO ontologijos ištrauka [1] 19
1.11 pav. AST struktūra [8] 20
1.12 pav. Tikslų modelis 22
2.1 pav. Kodo analizės įrankio panaudojimo atvejai 23
2.2 pav. PA „Atlikti kodo analizę“ veiklos diagrama 24
2.3 pav. PA „Pasirinkti analizės failų katalogą“ veiklos diagrama 26
2.4 pav. PA „Pasirinkti analizės failų vardų šabloną“ veiklos diagrama 27
2.5 pav. PA „Pasirinkti rezultatų saugojimo failą“ veiklos diagrama 28
2.6 pav. Dalykinės srities modelis 29
3.1 pav. Analizės diagrama 31
3.2 pav. Loginė architektūra 32
3.3 pav. Vartotojo sąsajos klasių modelis 32
3.4 pav. Veiklos logikos modelis 33
3.5 pav. Kodo analizės sekų diagrama 34
3.6 pav. Kodo analizės būsenų diagrama 35
3.7 pav. Komponentų diagrama 35
3.8 pav. Komponentų realizacija artefaktais 36
4.1 pav. Programinio įrankio veikimo algoritmas 37
4.2 pav. Grafinė vartotojo sąsaja 38
4.3 pav. Realizacijos klasių diagrama 39
4.4 pav. Testavimo rezultatai 40
5.1 pav. Grafinis informacijos apie kodo elementą atvaizdavimas 44

TERMINŲ IR SANTRUMPŲ ŽODYNAS

- **SCRO** (angl. *Source Code Representation Ontology*) – programinio kodo atvaizdavimo ontologija.
- **Ontologija** - tam tikros srities sąvokų visumos specifikavimas išreikštu pavidalu.
- **Java** - objektiškai orientuota programavimo kalba.
- **API** - aplikacijų programavimo sąsaja (angl. Application Programming Interface). Ją suteikia kompiuterinė sistema, biblioteka ar programa tam, kad programuotojas per kitą programą galėtų pasiekti jos funkcionalumą ar apsikeistų su ja duomenimis.
- **JavaDoc** – dokumentacijos generatorius, skirtas generuoti API dokumentaciją iš Java išeities tekstų HTML formatu.

ĮVADAS

Šiame darbe aprašomas tyrimas yra skirtas Informatikos fakulteto informacinių sistemų inžinerijos studijų programai. Programinio kodo analizė šiomis dienomis ypač svarbi programuotojams, kurie palaiko kitų programuotojų rašytą programinę įrangą. Kito autoriaus sukurtą programinį kodą suprasti gali būti sudėtinga. Tai atima nemažai laiko, o jį reikia naudoti, taisyti, praplėsti, perprogramuoti kita kalba.

Darbo tikslas ir uždaviniai

Darbo tikslas yra palengvinti ir pagreitinti programinio kodo supratimą, atvaizduojant jį programų kodo ontologijos elementais.

Suformuoti uždaviniai:

1. išanalizuoti programos kodo nuskaitymo procesą,
2. išanalizuoti SCRO ontologijos sudėtį ir panaudojimo galimybes,
3. ištirti darbo su ontologijomis įrankius ir bibliotekas,
4. sukurti automatinio kodo analizavimo priemonės prototipą, gebantį išsaugoti programos kodo analizės rezultatus SCRO ontologijos pavidalu,
5. eksperimentiškai ištirti sukurtą įrankį,
6. apibendrinti tyrimo rezultatus.

Darbo rezultatai ir jų svarba

Sukurtas įrankis gali būti panaudotas Java programinio kodo elementų atvaizdavimui SCRO ontologijos elementais. Rezultatas gali būti panaudotas kodo vizualizacijai, analizei, transliavimui į kitas objektinio programavimo kalbas, UML klasių diagramų generavimui. Įrankio principas gali būti panaudotas ir kitų objektinio programavimo kalbų programinio kodo elementų atvaizdavimui SCRO ontologijos elementais.

Darbo struktūra

Darbe yra 7 skyriai. Pirmame skyriuje pateikta tyrimo objekto, srities ir problemos, vartotojų, esamų statinės kodo analizės įrankių analizė.

Antrame skyriuje apibrėžti reikalavimai Java programinio kodo atvaizdavimo SCRO ontologijos elementais įrankiui.

Trečiame skyriuje pateikiamas Java programinio kodo atvaizdavimo SCRO ontologijos elementais įrankio projektas.

Ketvirtame skyriuje aprašyta įrankio realizacija ir testavimas.

Penktame skyriuje pateikiamas eksperimentas, kurio metu tiriama įrankio greitaveika ir panaudojimo galimybės.

Šeštame skyriuje pateikiamos galutinės darbo išvados.

Septintame skyriuje pateikiamas naudotos literatūros sąrašas.

1. JAVA PROGRAMOS KODO ANALIZĖS NAUDOJANT SCRO ONTOLOGIJĄ GALIMYBIŲ TYRIMO ANALIZĖ

1.1. Analizės tikslas

Analizės tikslas – ištirti esamą automatizuotos statinės kodo analizės sistemų problematiką, apžvelgti esamus darbus, susijusius su šia tema ir ištirti galimus sprendimo variantus.

Analizei atlikti naudoti šie metodai:

- literatūros šaltinių studijavimas,
- statinės kodo analizės procesų analizė,
- analogiškų problemos sprendimų paieška ir analizė.

1.2. Tyrimo objektas, sritis, problema

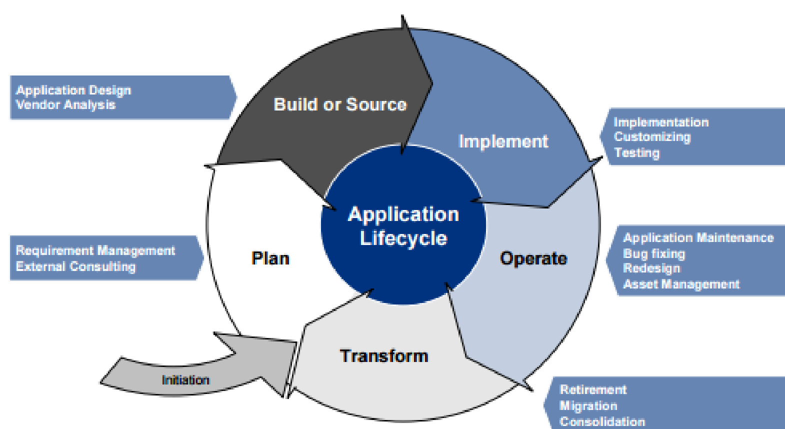
Tyrimo objektu pasirinkta programos kodo analizė. Bus tiriami programos kodo analizės automatizavimo metodai bei įrankiai, programos kodo supratimą palengvinantys įrankiai, SCRO ontologijos panaudojimas kodo analizėje.

Kito autoriaus sukurtą programinį kodą suprasti gali būti sudėtinga, jeigu to autoriaus programavimo patirtis, programavimo stilius skiriasi nuo to asmens, kuris bando tą kodą suprasti. Tai atima nemažai laiko, kurį galima būtų skirti kodo naudojimui, taisymui, praplėtimui, perprogramavimui kita kalba.

Daroma prielaida, kad šią problemą padėtų spręsti naujoviškos priemonės (prototipo), kuri automatiškai analizuotų programos išeities kodą ir jį atvaizduotų SCRO (angl. *Source Code Representation Ontology* [1]) programų kodo ontologijos elementais, sukūrimas.

1.3. Programinio kodo statinės analizės proceso analizė

Kiekviena programa turi savo gyvavimo ciklą. Gyvavimo ciklo apibrėžimai įvairiuose šaltiniuose šiek tiek skiriasi, bet pagrindiniai etapai visur sutampa. Štai konsultacijų kompanija „Detecon“ pateikia iš penkių, cikliškai vienas po kito einančių, etapų sudarytą programos gyvavimo ciklo modelį 1.1 pav.

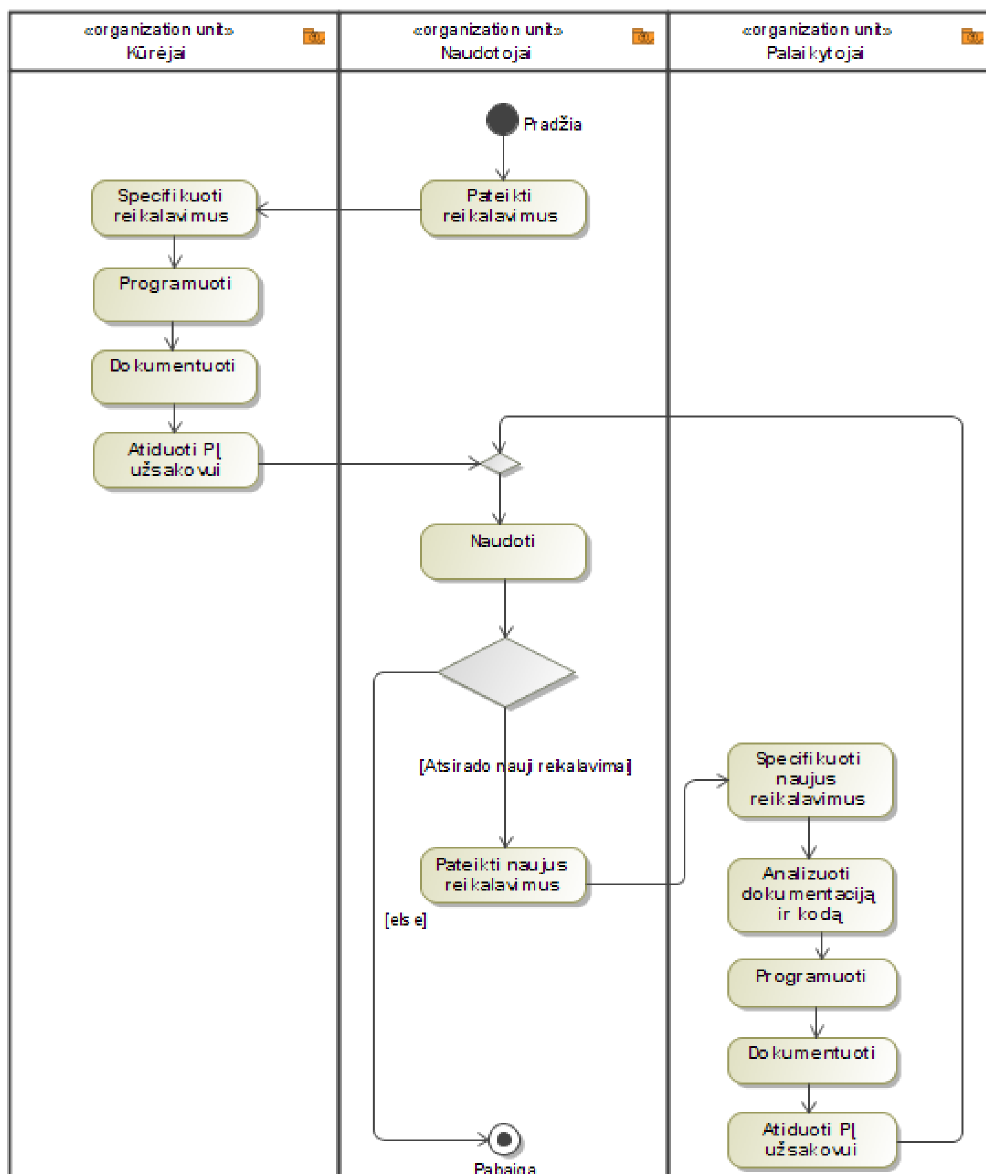


1.1 pav. Programos gyvavimo ciklo diagrama [11]

Šie etapai yra tokie:

1. Planavimas – kur yra derinami reikalavimai programai.
2. Projektavimas – kur yra kuriama programos architektūra pagal reikalavimus.
3. Realizacija – kur yra programuojama ir testuojama programa.
4. Vykdymas – kur programa yra naudojama, prižiūrima, taisomos klaidos.
5. Transformavimas – kur programa yra keičiama kitu produktu, atnaujinama, perdaroma.

Programinio kodo analizė gali būti atliekama bet kuriame iš šių etapų, jei susiduriame su programa, kuri yra jau naudojama ir ją reikia atnaujinti ar pertvarkyti. Naujai programai analizė gali praversti realizacijos (pavyzdžiui programuotojų komandos nariams bandant suprasti vieniems kitų kodą, arba rasti klaidas, kol dar programa nepradėta naudoti), vykdymo (norint rasti klaidas, arba suprasti kaip veikia programos komponentai), bet transformavimo etapuose.

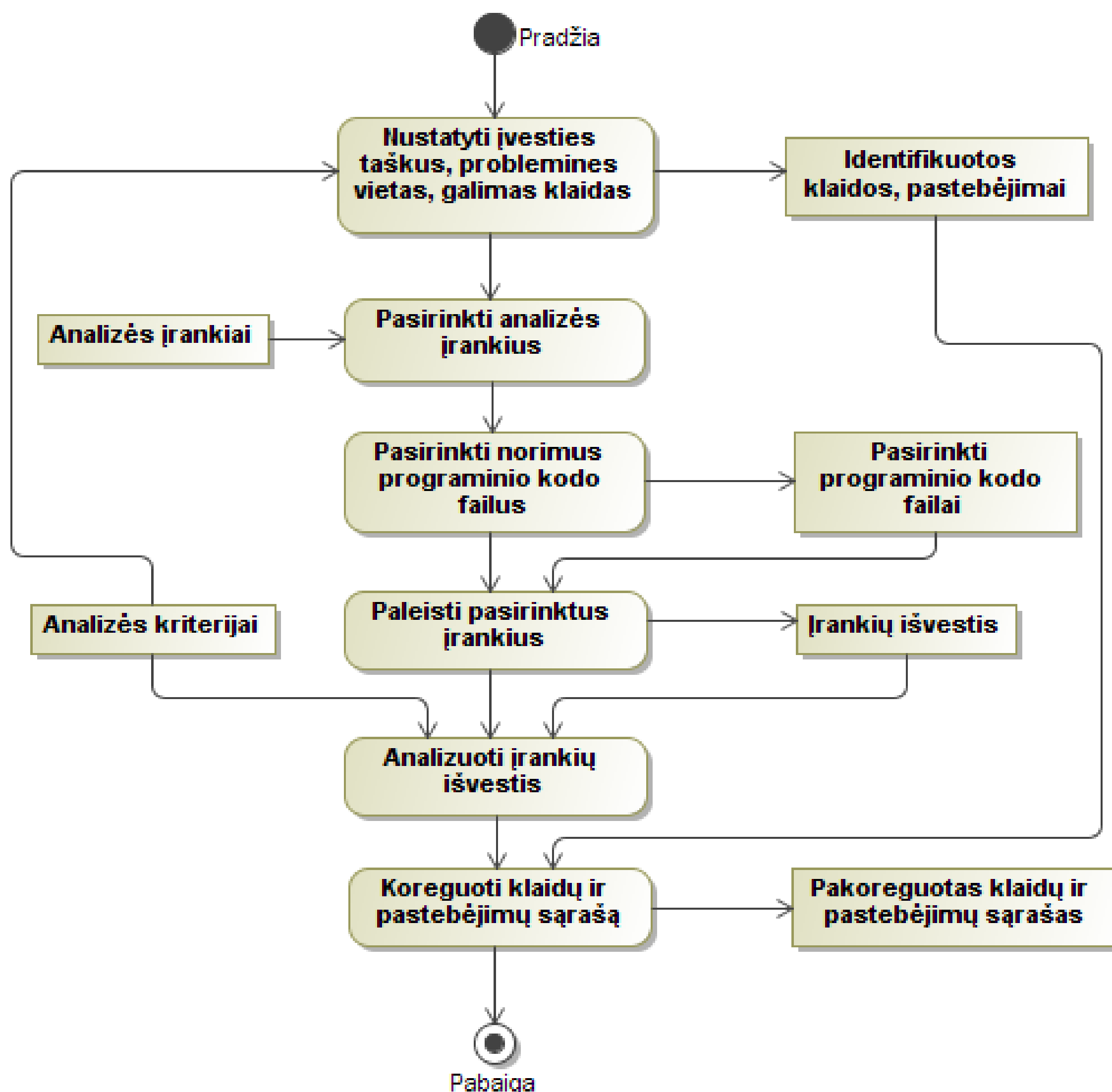


1.2 pav. Programinės įrangos gyvavimo ciklas

1.2 pav. matome programinio kodo analizės kontekstą – PĮ gyvavimo ciklą. Iš modelio matome kad programą kuriantis ir naujus reikalavimus realizuojantis personalas gali skirtis, dėl to programinio kodo analizė čia yra būtina tam, kad tobulinantis personalas galėtų geriau suprasti

Programinio kodo analizė atliekama norint patobulinti, optimizuoti, pertvarkyti ar surasti klaidas esamame programiniame sprendime. Taip pat tai gali būti naudojama palengvinti programinio kodo supratimą naujai į komandą atėjusiam ar projektą perėmusiam programuotojui. Vienas iš kodo analizės būdų – statinė kodo analizė. Statinei kodo analizei atlikti naudojami statinės kodo analizės įrankiai. Dauguma šių įrankių gali padėti aptikti įvairias klaidas programiniame kode, tačiau nepalengvina kodo supratimo. 1.3 pav. matome statinės kodo analizės proceso pavyzdį, naudojant statinės kodo analizės įrankį: suinteresuotas žmogus peržiūri kodą, nustato galimai silpnas

ir pažeidžiamas vietas. Po to pasirinktiems programinio kodo failams automatiškai išanalizuoti gali būti panaudojami automatiniai įrankiai. Peržiūrimas galimai silpnų ir pažeidžiamų vietų sąrašas ir pakoreguojamas.



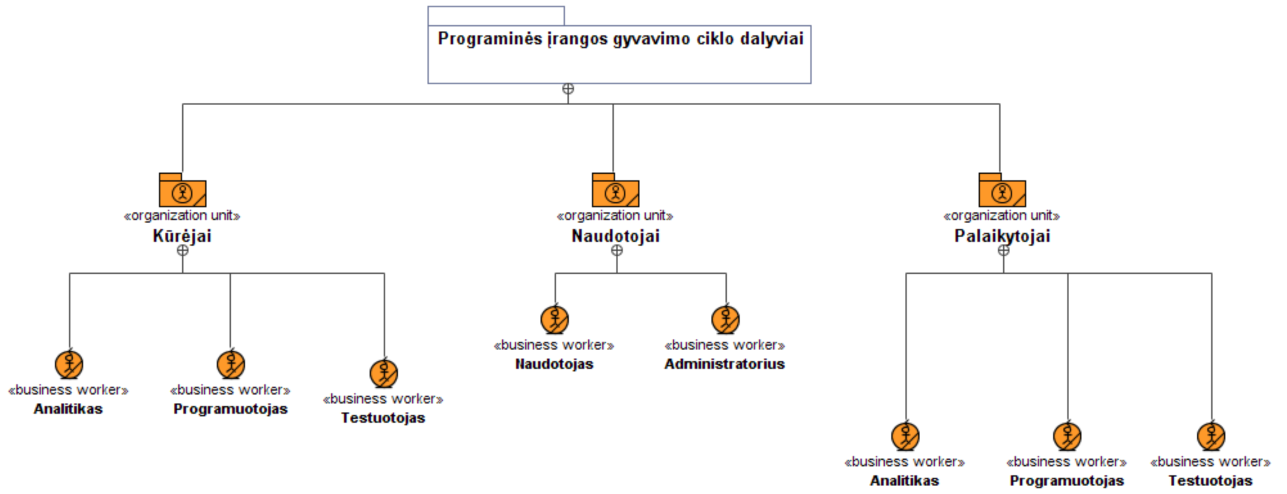
1.3 pav. Kodo analizės proceso pavyzdys

Taip pat prie statinės kodo analizės galima priskirti ir programinio kodo supratimo procesą. Jo metu kodą skaito suinteresuotas programuotojas ir bando suprasti kaip programa iš tikrųjų veikia. Nedidelėms programoms suprasti pilnai užtenka vieno žmogaus, tačiau augant programos apimčiai didėja ir reikiamos įsiminti informacijos kiekis. Todėl be pagalbinių įrankių suprasti didelės apimties programos veikimą gali būti itin sunku.

Tyrimo metu norima palengvinti programos kodo supratimą panaudojant SCRO ontologiją – atvaizduojant programos kodo elementus ir ryšius tarp jų ontologijos elementais. Ontologijos – tai tam tikros srities sąvokų aprašymas iš anksto apibrėžtu griežtu pavidalu. Ontologijose apibrėžiamos sąvokos ir jų hierarchijos, esybių tipai ir jų tarpusavio ryšiai. Tokiu pavidalu pateikiami duomenys yra aiškūs, lengvai skaitomi, galima naudoti programinius įrankius (pvz., *Protege*) duomenų tarpusavio sąveikoms tirti.

Šiame darbe bus apsiribojama Java programavimo kalba, kuri yra populiari objektinio programavimo kalba, dažnai naudojama verslo informacinėms sistemoms kurti. Kadangi verslui yra svarbus informacinių sistemų palaikymas ir klaidų taisymas – kodo analizė taip pat yra svarbi.

1.4 pav. vaizduojami aktoriai, dalyvaujantys programinės įrangos gyvavimo procese.



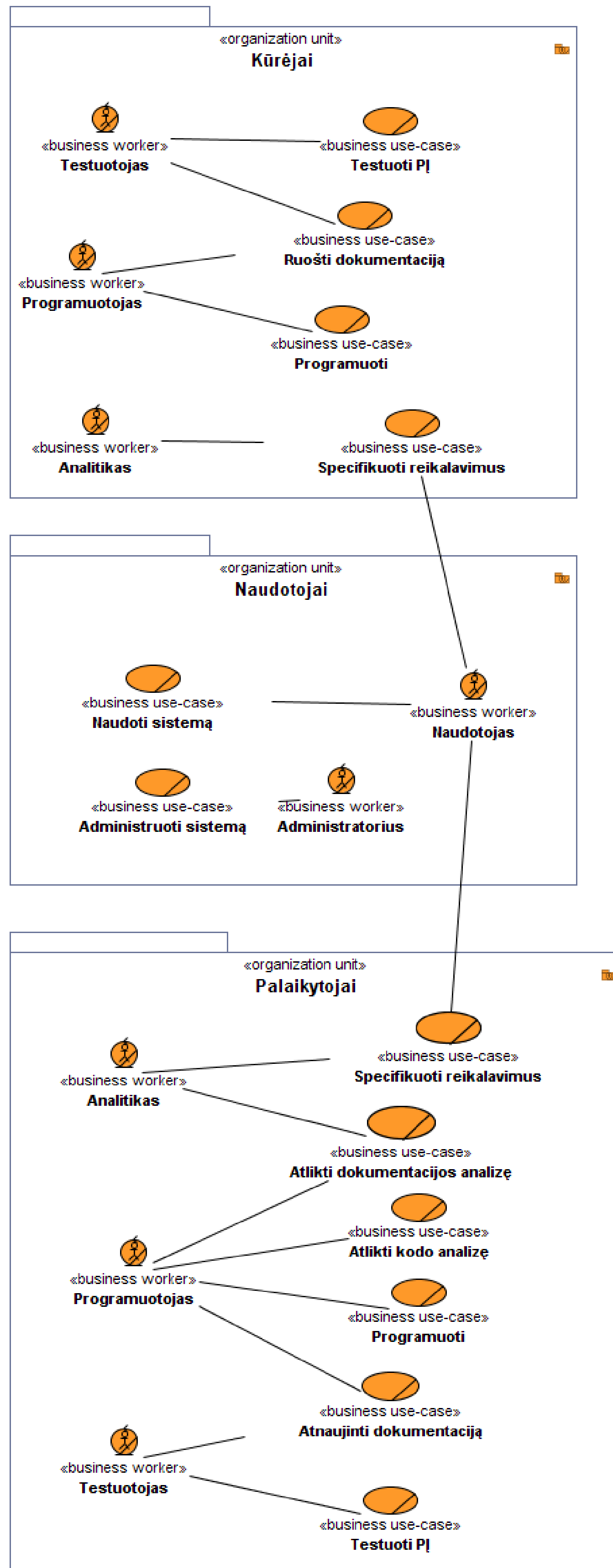
1.4 pav. Organizacinė struktūra

1.5 pav. vaizduojami kodo analizavimo veiklos konceptai.

«business entity»
Reikalavimai
«business worker»
Naudotojas
«business worker»
Analitikas
«business entity»
Naudojimo dokumentacija
«business entity»
Programinė įranga
«business worker»
Administratorius
«business worker»
Testuotojas
«business entity»
Dokumentacija
«business entity»
Administravimo dokumentacija
«business entity»
Komentaras
«business entity»
Kodo dokumentacija
«business entity»
Klasė
«business entity»
Interfeisas
«business entity»
Operatorius
«business worker»
Programuotojas
«business entity»
Programinės įrangos kodas
«business entity»
Metodas
«business entity»
Kintamasis
«business entity»
Reikšmė

1.5 pav. Kodo analizavimo veiklos konceptai

1.6 pav. pavaizduoti programinės įrangos gyvavimo ciklo veiklos panaudojimo atvejai.



1.6 pav. Programinės įrangos gyvavimo ciklo panaudojimo atvejai

1.4. Vartotojų analizė

Statinės kodo analizės sistemos aktualios programuotojams, testuotojams, palaikančiam programas ir klaidų ieškančiam personalui.

Naudodami statinės kodo analizės įrankius, jie siekia programinio kodo supratimui sugaišti mažiau laiko ir gauti išsamesnius rezultatus.

1.5. Esamų problemos sprendimo metodų analizė

Panašių sprendimų paieška buvo vykdoma pagal šiuos kriterijus:

- Ar naudojamos ontologijos? Šis kriterijus svarbus, nes keliama hipotezė kad ontologijos palengvina kodo supratimą.
- Ar palaikoma Java programavimo kalba? Kadangi magistro baigiamajame darbe nuspręsta apsiriboti Java programavimo kalba, todėl svarbu, kad ieškomi sprendimai taip pat ją palaikytų.
- Ar naudojamas automatinis kodo nuskaitymas? Rankinis kodo perrašymas ontologijos elementais užimtų labai daug laiko, todėl sutaupytas laikas analizuojant kodą netektų prasmės.
- Ar palaikomi visi svarbiausi Java programinio kodo elementai? Jeigu nepalaikomi visi baziniai Java programinio kodo elementai, tai kodo supratimą gali tik apsunkinti, nes bus prarandama svarbi informacijos, esančios kode, dalis.
- Ar sprendimas palengvina programinio kodo supratimą? Sprendimas turėtų palengvinti kodo supratimą. Jeigu rezultatai neaiškus, jeigu pateikiami tik kokiu nors vienu pjūviu – kodo suprasti tai nepadės.
- Ar analizės rezultatus galima peržiūrėti trečiųjų šalių įrankiais? Jeigu analizės rezultatų formatas aiškus, dokumentuotas ir plačiai taikomas – galima naudoti įvairius trečiųjų šalių įrankius kodo peržiūrėjimui skirtingais pjūviais, apie kuriuos, rašant analizės įrankį, galbūt net nebuvo pagalvota.

CODEPRO ANALYTIX KODO ANALIZĖS IR TESTAVIMO ĮRANKIS

CodePro Analytix – tai „Google“ kompanijoje sukurtas galingas statinės kodo analizės ir testavimo įrankis [2]. Jis padeda išvengti klaidų programuojant, tikrindamas kodo kokybę pagal iš anksto nustatytus klaidų šablonus ir kodo kokybės parametrus. Tai pat palengvina vienetų testų rašymą, matuoja kodo padengimą. CodePro Analytix palaiko tik Java programavimo kalba ir veikia tik Eclipse programavimo aplinkoje.

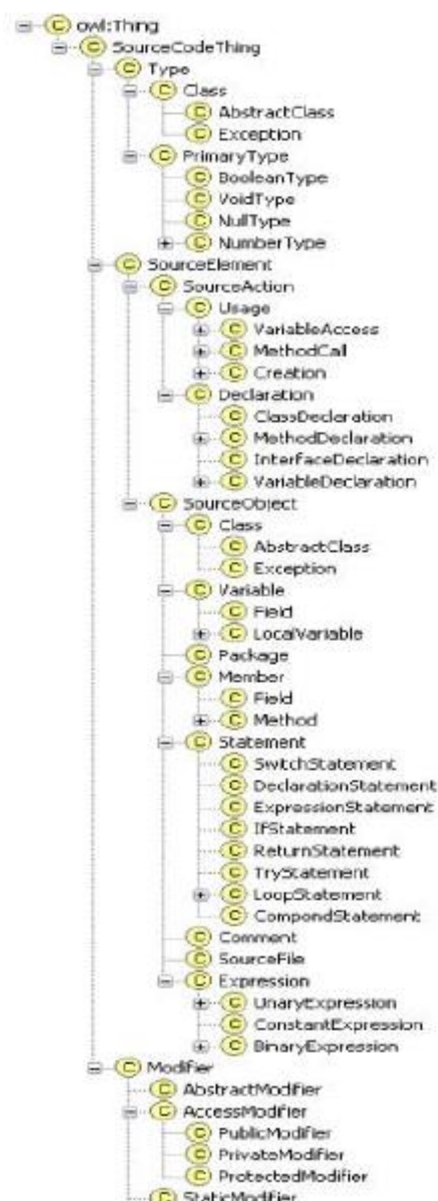
Pagrindinės CodePro Analytix taikymo galimybės:

- statinė kodo analizė – integruoti įrankiai, analizuojantys kodą ir ieškantys nukrypimų nuo priimtų kodo standartų,
- metrikos – įrankiai, analizuojantys kodo kokybės parametrus,
- JUnit testų generavimas – automatinis JUnit testų generavimas, naudojantis programos vykdymo sekos analizę,
- JUnit testų redaktorius – įrankis, leidžiantis greitai modifikuoti JUnit testus, nesigilinant į testo kodą,
- kodo padengimas – įrankis, matuojantis kiek kodo yra padengiama JUnit testais,
- priklausomybių analizė – įrankis, vaizdžiai parodantis priklausomybes tarp projektų, paketų ir tipų,
- panašaus kodo analizė – įrankis, leidžiantis sumažinti kodo dubliavimąsi, automatiškai randant panašius kodo gabalus ir siūlant juos apjungti.

Tačiau šis įrankis nepalengvina kodo supratimo: randamos klaidos, tačiau negalima pamatyti bendro programos veikimo principo, negalima atlikti paieškos pagal kodo dizaino šablonus, negalima įtraukti savo kodo kokybės taisyklių.

AN ONTOLOGY-BASED PROGRAM COMPREHENSION MODEL (ONTOLOGIJOMIS PAGRĮSTAS PROGRAMOS SUVOKIMO MODELIS)

Šį modelį sukūrė Yonggang ZHANG, 2007 metais. Jis išsamiai aprašytas autoriaus disertacijoje [3]. Modelis tinkamas įvairioms programavimo kalboms, tačiau tezėje buvo nagrinėjamas jo panaudojimas Java kalbai. Java kodo transliavimui buvo panaudotas *Eclipse* JDT įrankis. Modelis skirtas efektyviai paieškai kode ir geresniam kodo supratimui. Struktūrą matome 1.7 pav.



1.7 pav. OBPCM struktūra [3]

Iš struktūros matome, kad SCRO modelis analizei tinkamesnis, nes palaikoma daugiau programos kodo atributų, tokių kaip interfeisai (angl. *Interface*), vidinės klasės (angl. *Inner class*), anotacijos (angl. *Annotation*), raktažodis *final*, metodų parametrai.

JARCHITECT KODO VIZUALIZACIJOS ĮRANKIS

JArchitect tai įrankis, padedantis supaprastinti sudėtingą Java kodo bazę. Architektai ir programuotojai gali analizuoti kodo struktūrą, užduoti architektūros taisykles, efektyviai peržiūrėti kodą ir suvaldyti programos evoliuciją, lyginant skirtingas kodo versijas. [4]

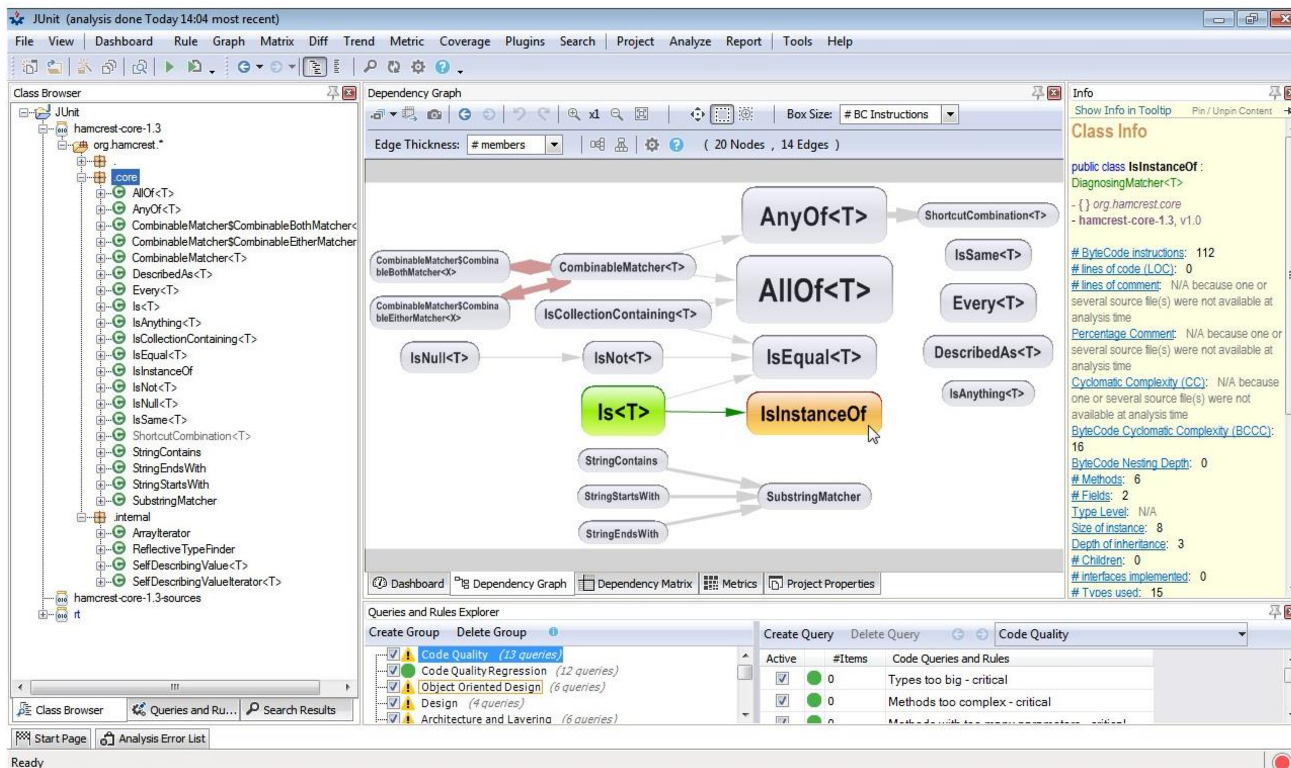
Tai yra komercinis įrankis, palaikantis Java programavimo kalbą, veikiantis visose trijose pagrindinėse operacinėse sistemose (Linux, Mac OS, Windows).

Naudojantis įrankiu, galima rašyti CQLinq užklausas 1.8 pav., gebančias grąžinti vertingą informaciją apie kodą, pvz.: kokios klasės paveldi (angl. *extends*) norimą klasę, koks tam tikro metodo ciklomatinis sudėtingumas ir pan.

Įrankis gali grafiškai atvaizduoti programos struktūra, kurti priklausomybių grafus 1.9 pav.

```
// <Name>Base class should not use derivatives</Name>
warnif count > 0
from baseClass in JustMyCode.Types
where baseClass.IsClass &&
      baseClass.NbChildren > 0
let derivedClassesUsed =
      baseClass.DerivedTypes.UsedBy(baseClass)
where derivedClassesUsed.Count() > 0
select new { baseClass, derivedClassesUsed }
```

1.8 pav. CQLinq užklausos pavyzdys [5]



1.9 pav. Sugeneruotas priklausomybių tarp projektų grafas [4]

JDEPEND ARCHITEKTŪROS KOKYBĖS NUSTATYMO ĮRANKIS

JDepend įrankis palengvina kodo architektūros kokybės nustatymą metrikomis, kurios skaičiuojamos pagal paketų ir klasių tarpusavio priklausomybes [6].

Metrikos skaičiuojamos pagal:

- klasių ir interfeisų kiekį pakete,
- paketų kiekį, kurie priklauso nuo klasių esančių nagrinėjamame pakete,
- klasių iš kitų paketų panaudojimų kiekį nagrinėjamame pakete,
- abstrakčių klasių (ir interfeisų) ir konkrečių klasių santykį nagrinėjamame pakete,
- paketų su ciklinėmis priklausomybėmis kiekį.

JDepend analizės rezultatus galima peržiūrėti tiek programos grafinėje vartotojo sąsajoje, tiek tekstiniu, bei XML formato pavidalais.

Kadangi šis įrankis tiesiog apskaičiuoja kodo architektūros kokybės metrikas pagal paketų ir klasių tarpusavio priklausomybes ir pateikia tai skaitiniu pavidalu – galime susidaryti įspūdį tik apie pačią kokybę, tačiau nematome kodo veikimo principų, naudojamų dizaino šablonų. Todėl kodo supratimo jis nepalengvina.

1.6. Architektūros ir galimų įgyvendinimo priemonių variantų analizė

SCRO – SOURCE CODE REPRESENTATION ONTHOLOGY (PROGRAMINIO KODO ATVAIZDAVIMO ONTOLOGIJA)

SCRO sukurta palengvinti programinio kodo supratimo užduotis, vaizdžiai parodant programinio kodo struktūrą. Ontologija apima pagrindinius objektinio programavimo principus ir padeda suprasti ryšius bei priklausomybes tarp programinio kodo elementų [7]. Palaikomi tokie konceptai kaip:

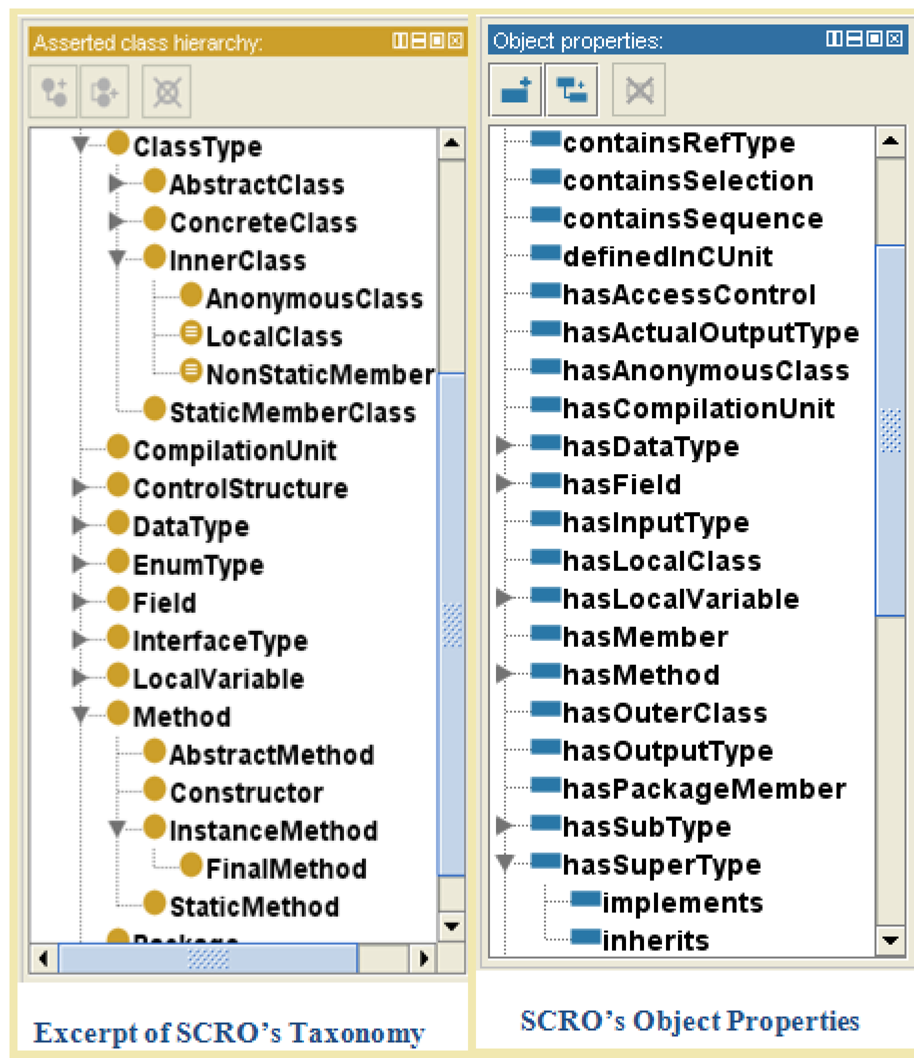
- enkapsuliacija,
- pakartotinis panaudojimas,
- paveldėjimas,

- metodų perkrovimas,
- metodų perrašymas.

SCRO klasės atitinka svarbiausius įvairių objektinių programavimo kalbų elementus:

- įvairūs metodų tipai,
- klasės,
- kelių lygių klasės,
- laukai,
- ir kiti.

Įvairūs šių elementų parametrai taip pat naudojami SCRO ontologijoje, tam kad būtų galima rasti ryšius tarp skirtingų įvairių tipų programos kodo elementų [1]. SCRO ontologijos žodyno dalį ir kai kuriuos galimus elemento parametrus matome 1.10 pav.:

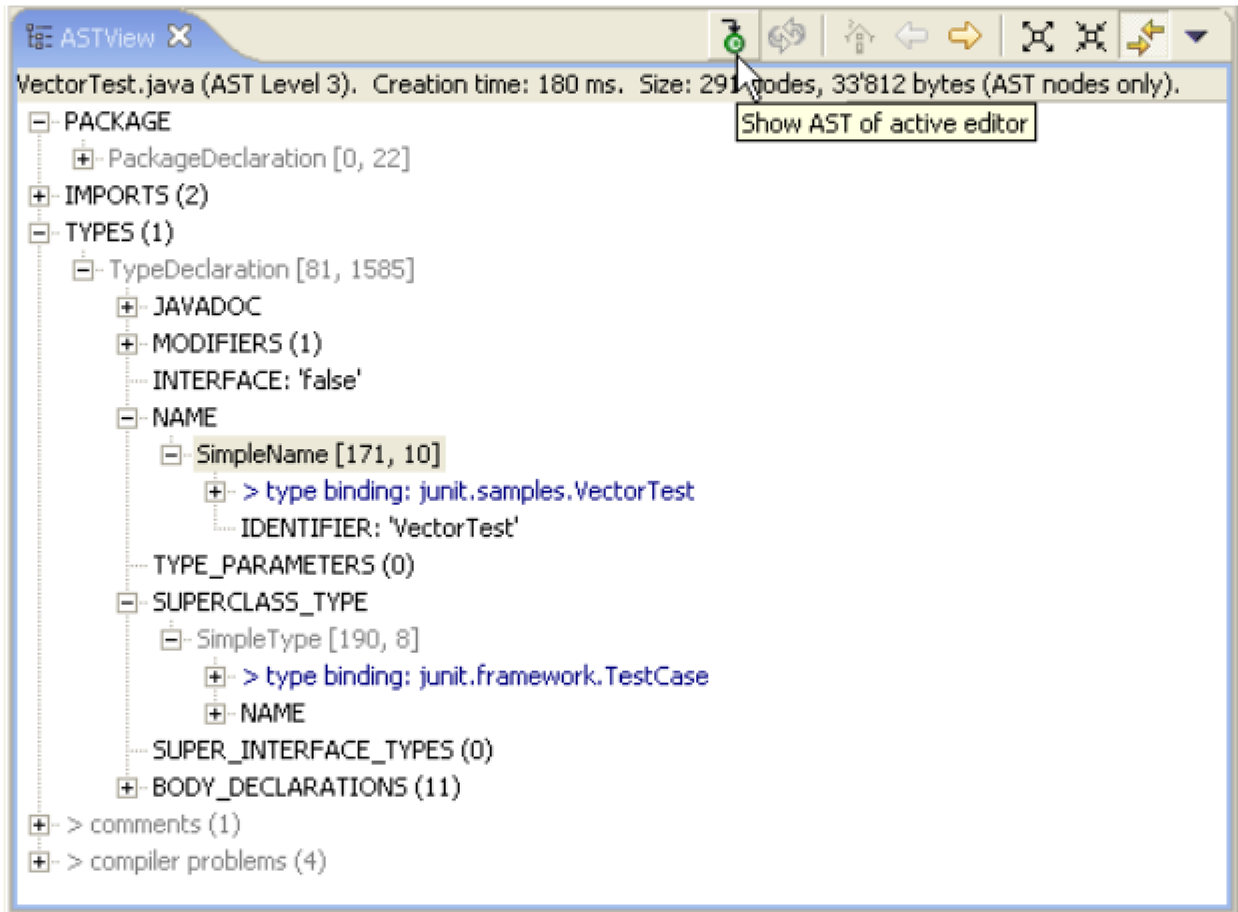


1.10 pav. SCRO ontologijos ištrauka [1]

ECLIPSE JDT IR ABSTRAKTUS SINTAKSĖS MEDIS (ANGL. *ABSTRACT SYNTAX TREE* (AST))

Eclipse JDT įrankį sukūrė ir palaiko organizacija „*The Eclipse Foundation*“. Jis naudojamas *Eclipse* IDE programinėje įrangoje, siekiant palengvinti navigaciją kode, sintaksės spalvinimui, kodo

pertvarkymo (angl. *refactoring*) funkcijai. Tai pasiekiama transliuojant JAVA kodą į abstraktų sintaksės medį [8] [9]. AST struktūra matoma 1.11 pav.:



1.11 pav. AST struktūra [9]

AST struktūra yra pakankamai išsami kodo analizei, tačiau sunki suprasti. Šios problemos galima būtų išvengti transliuojant AST struktūrą į SCRO ontologiją.

1.7. Panašių sprendimų ir galimų įgyvendinimo priemonių analizės išvados

1.1 lentelėje matome, kad OBPCM mums tiktų, tačiau buvo prieita prie išvados, kad *ontology-based program comprehension model* ontologijos detalumo neužtenka norint patobulinti statinės kodo analizės procesą – trūkstami kodo atributai programos supratimą gali apsunkinti, pavyzdžiui ar klasė pagal dizainą yra *Singleton* tipo (t.y. ar visos programos ribose klasės objektas gali būti sukuriamas tik vieną kartą), naudodami šį modelį, galime spręsti tik iš to, kad klasės konstruktorius yra privatus (angl. *private*). Tuo tarpu pagal Java EJB specifikaciją, klasė gali būti *Singleton* ir visai neturėdama konstruktoriaus – užtenka *Singleton* anotacijos klasės apraše. Taip pat OBPCM neturi metodų perrašymo atvaizdavimo galimybės (angl. *override*), kas keltų sumaištį nagrinėjant toliau pateikiamą kodą:

```
public class OverrideDemoClass {
        private int x;

        public int getX() {
                return this.x;
        }

        static public int getX(OverrideDemoClass object) {
                return object.x;
        }
}
```

SCRO ontologija šių trūkumų neturi, todėl ją galima naudoti ir kodo dizaino šablonų aptikimui [10].

1.1 lentelė. Sprendimų palyginimas

	OBPCM	CodePro Analytix	JArchitect	JDepend	SCRO	Eclipse JDT
Ar naudojamos ontologijos	+	–	–	–	+	–
Ar palaikoma Java programavimo kalba	+	+	+	+	+	+
Ar naudojamas automatinis kodo nuskaitymas	+	+	+	+	-	+
Ar palaikomi visi svarbiausi Java kodo elementai	-	+	+	-	+	+
Ar sprendimas palengvina programinio kodo supratimą	-	-	+	-	+	-
Ar analizės rezultatus galima peržiūrėti trečiųjų šalių įrankiais	+	-	-	+	+	-

1.8. Darbo tikslas ir uždaviniai

Šio darbo tikslas yra palengvinti programinio kodo analizę, sukuriant įrankį atvaizduojantį Java programinį kodą SCRO ontologijos elementais.

Tikslui pasiekti turi būti išspręsti tokie uždaviniai:

1. Išanalizuoti programos kodo nuskaitymo procesą.
2. Išanalizuoti SCRO ontologijos sudėtį ir panaudojimo galimybes.
3. Ištirti darbo su ontologijomis įrankius ir bibliotekas.
4. Sukurti automatinio kodo analizavimo priemonės prototipą, gebantį išsaugoti programos kodo analizės rezultatus SCRO ontologijos pavidalu.
5. Eksperimentiškai ištirti sukurtą įrankį.
6. Apibendrinti tyrimo rezultatus.

Sukurtam programiniam įrankiui keliami tokie kokybės kriterijai:

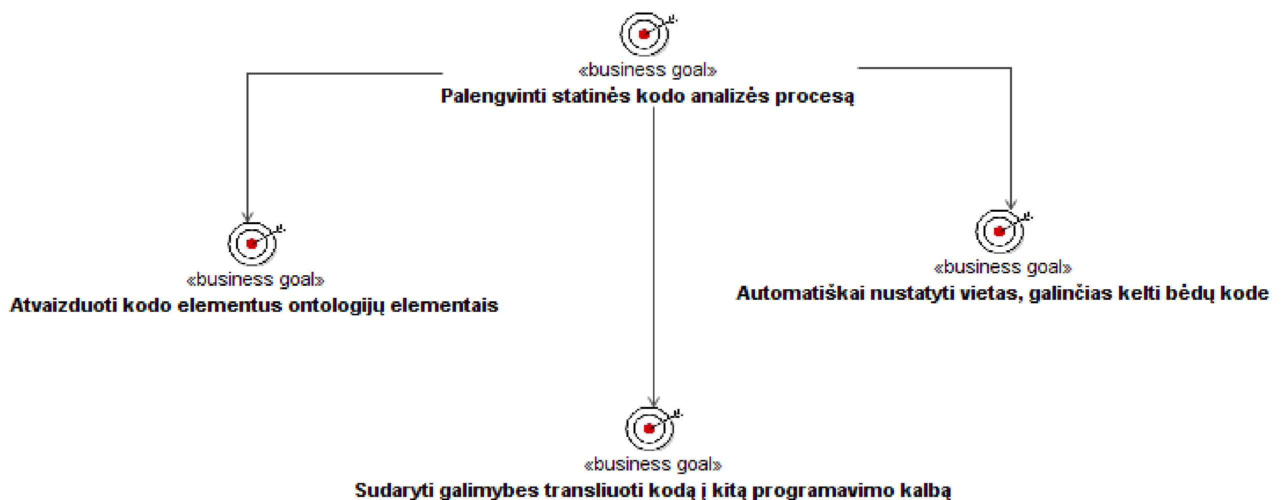
1. Atpažintų Java elementų kiekis.
2. Panaudotų SCRO ontologijos elementų kiekis.
3. Naudojamų SCRO ontologijos elementų atributų kiekis.
4. Maksimali leidžiama analizuojamo kodo failų apimtis – įrankis neturi lūžti apdorodamas didelius projektus. Kaip atskaitos taškas bus naudojamas atviro kodo portalo „Liferay“ projektas, susidarantis iš 10118 failų, 3613967 eilučių.
5. Analizės trukmė – įrankis apdorodamas „Liferay“ projektą, atsižvelgus į jo apimtį, neturėtų užtrukti ilgiau kaip 5 minutes.

1.9. Siekiamo sprendimo apibrėžimas

Atlikus analizę, nuspręsta įgyvendinti programinio įrankio, Java programavimo kalbos kodo analizei atlikti, prototipą. Tai būtų programa, atvaizduojanti Java programavimo kalba parašytą kodą SCRO ontologijos elementais. Java programavimo kalba parašytas kodas turėtų interpretuojamas naudojant *Eclipse* JDT bibliotekas, o tada, naudojant OWL-API biblioteką, transliuojamas į SCRO ontologiją. Rezultatus būtų galima analizuoti naudojant bet kurį įrankį skirtą darbui su ontologijomis, palaikantį OWL-DL ontologijų kalbą.

Kuriamas programinis įrankis dalyvautų 1.3 pav. pateiktame kodo analizės procese – analizuoti įrankio išvestis bus patogiau naudojantis darbui su ontologijomis skirta programine įranga.

1.12 pav. vaizduojami tikslai, kuriuos stengiamasi išpildyti įgyvendinus programinio įrankio prototipą.



1.12 pav. Tikslų modelis

1.10. Analizės išvados

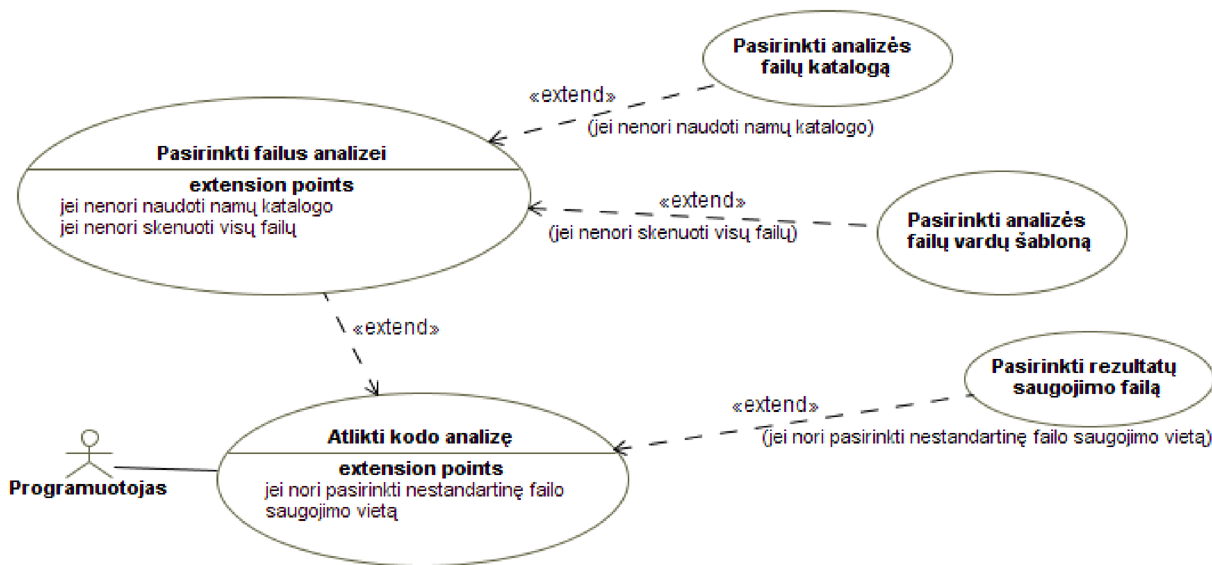
1. Atlikus analizę, matome kad kuriant šiuolaikinę, sudėtingą programinę įrangą, labai svarbu yra naudotis statinės kodo analizės įrankiais. Jie leidžia aptikti klaidas dar nesukompiliavus ir neišbandžius programos, palengvina kitų programuotojų parašyto kodo supratimą, kai programos kūrime dalyvauja didelis programuotojų kiekis.
2. Programinių įrankių (tiek komercinių, tiek nemokamų), skirtų klaidų paieškai statiškai analizuojant kodą yra daug, jie veikia gerai, todėl nuspręsta dirbti kodo vizualizavimo ir supratimo palengvinimo kryptimi.
3. Vizualizuojant kodą patogu remtis ontologijomis, nes jos nėra priklausomos nuo konkrečios programavimo kalbos ir gali tarnauti kaip meta-modelis, kuriame nėra prarandamos svarbios kodo ypatybės.
4. Remiantis panašiomis išvadomis yra sukurtas „Ontologijomis paremtas programinio kodo supratimo modelis“ (angl. *Ontology Based Program Comprehension Model*). Tačiau atlikus detalesnę analizę, paaiškėjo, kad šiame meta-modelyje yra prarandamos kai kurios svarbios kodo ypatybės, dėl kurių kodo supratimas gali ne palengvėti, o pasunkėti.
5. Nutarta kad didžiąją OBPCM modelio trūkumų dalį pašalina SCRO ontologijos panaudojimas, todėl reikia sukurti įrankį, leidžiantį atvaizduoti programinį kodą SCRO ontologijos elementais.

2. JAVA PROGRAMOS KODO ANALIZĖS EKSPERIMENTINIO ĮRANKIO REIKALAVIMŲ SPECIFIKACIJA IR ANALIZĖ

Šiame skyriuje formuluojami reikalavimai kodo analizės įrankio prototipui. Tai turi būti programa, turinti grafinę sąsają, su galimybe nurodyti failų, kuriuos norima analizuoti, saugojimo vietą ir jų paieškos parametrus.

2.1. Reikalavimų specifikacija

2.1 pav. vaizduojami kodo analizės įrankio panaudojimo atvejai. Esminis panaudojimo atvejis yra „Atlikti kodo analizę“.



2.1 pav. Kodo analizės įrankio panaudojimo atvejai

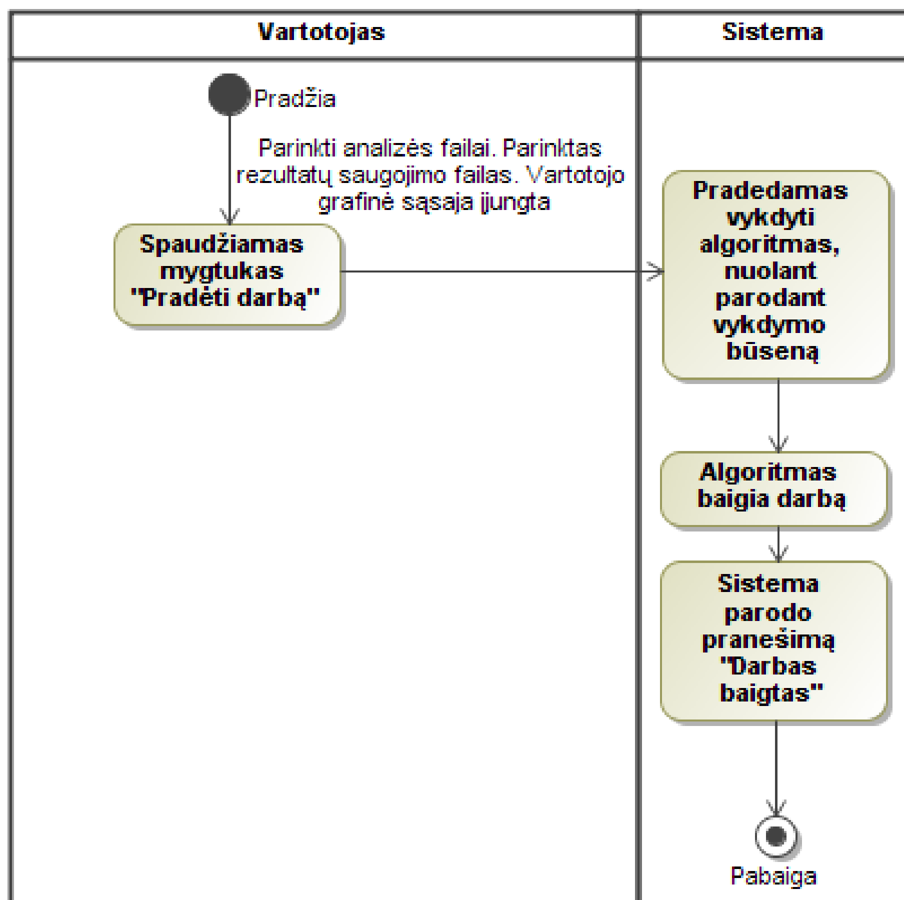
PANAUDOJIMO ATVEJAI

Panaudojimo atvejis „Atlikti kodo analizę“ aprašytas 2.1 lentelėje ir pavaizduotas 2.2 pav.

2.1 lentelė. Panaudojimo atvejis 1

PA „Atlikti kodo analizę“		
Tikslas		Išanalizuoti turimą programinį kodą
Aprašymas		Programuotojas paleidžia įrankį, norėdamas išanalizuoti turimą programą
Prieš sąlyga		1. Pasirinkti failai analizei 2. Pasirinkta rezultatų failo saugojimo vieta
Aktorius		Programuotojas (sistemos naudotojas)
Sužadinimo sąlyga		Vartotojas nustatė įrankio parametrus ir nori pradėti kodo analizę
Susiję panaudojimo atvejai	**Išplečia PA**	
	Apima PA	
	Specializuoja PA	
Pagrindinis įvykių srautas		Sistemos reakcija ir sprendimai
1. Spaudžiamas mygtukas „Pradėti analizę“		Rodoma progreso juosta, kurioje atsispindi kiek darbo jau yra atlikta (procentais).

	Parodomas pranešimas „Darbas baigtas“.
	Sistema baigia PA
Po sąlyga	Sugeneruotas .owl ontologijos failas
Alternatyvūs scenarijai	
Jei pagal nurodytus parametrus nerasti programinio kodo failai	Sistema parodo klaidos pranešimą



2.2 pav. PA „Atlikti kodo analizę“ veiklos diagrama

Panaudojimo atvejis „Pasirinkti failus analizei“ aprašytas 2.2 lentelėje.

2.2 lentelė. Panaudojimo atvejis 2

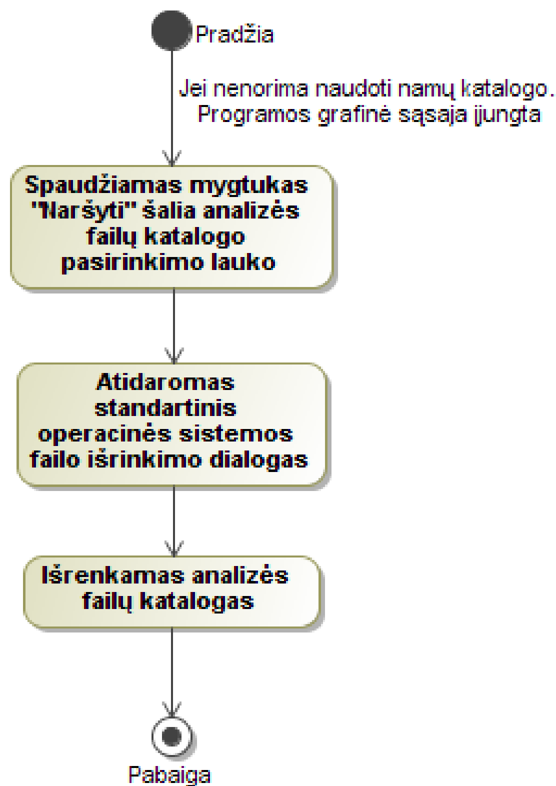
PA „Pasirinkti failus analizei“		
Tikslas		Išanalizuoti turimą programinį kodą
Aprašymas		Parenkami failai, kurie bus analizuojami
Prieš sąlyga		1. Pasirinktas katalogas, kuriame bus vykdoma programos išeities tekstų paieška 2. Pasirinktas failų vardų filtras
Aktorius		Programuotojas (sistemos naudotojas)
Sužadinimo sąlyga		Vartotojas nori pasirinkti failus analizei
Susiję panaudojimo atvejai	**Išplečia PA**	„Atlikti kodo analizę“
	Apima PA	
	Specializuoja PA	
Pagrindinis įvykių srautas		Sistemos reakcija ir sprendimai

1. Pasirenkamas rekursijos gylis (keliuose katalogų medžio lygmenyse ieškoti išeities tekstų failų)	
2. Vartotojas baigia PA	
Po sąlyga	Nustatytas filtras, pagal kurį bus ieškomi failai analizei

Panaudojimo atvejis „Pasirinkti analizės failų katalogą“ aprašytas 2.3 lentelėje ir pavaizduotas 2.3 pav.

2.3 lentelė. Panaudojimo atvejis 3

PA „Pasirinkti analizės failų katalogą“		
Tikslas		Nurodyti katalogą, kuriame bus ieškoma programos išeities tekstų
Aprašymas		Parenkamas katalogas, kuriame bus ieškoma programos išeities tekstų
Prieš sąlyga		Paleista programos vartotojo sąsaja
Aktorius		Programuotojas (sistemos naudotojas)
Sužadinimo sąlyga		Vartotojas nori pasirinkti failus analizei
Susiję panaudojimo atvejai	**Išplečia PA**	„Pasirinkti failus analizei“
	Apima PA	
	Specializuoja PA	
Pagrindinis įvykių srautas		Sistemos reakcija ir sprendimai
1. Parenkamas katalogas, naudojant grafinę sąsają		
2. Vartotojas baigia PA		
Po sąlyga		Nustatytas analizės failų katalogas

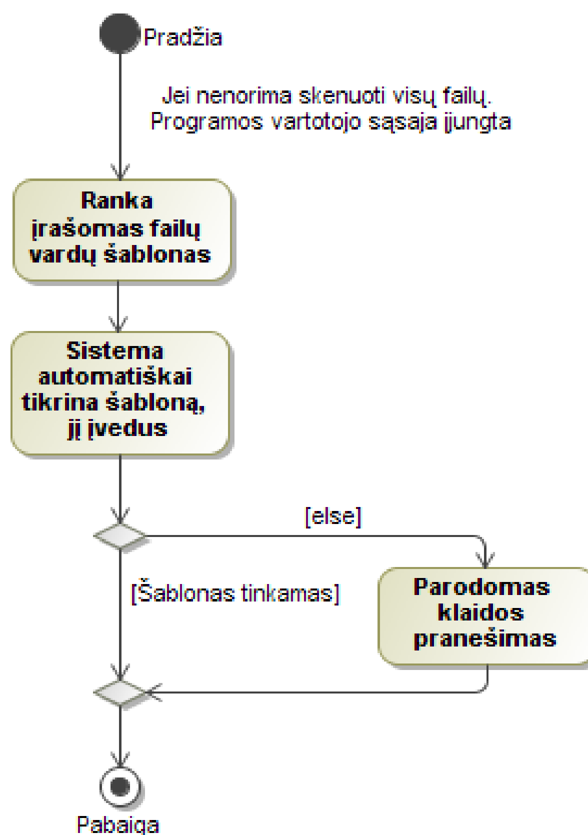


2.3 pav. PA „Pasirinkti analizės failų katalogą“ veiklos diagrama

Panaudojimo atvejis „Pasirinkti analizės failų vardų šabloną“ aprašytas 2.4 lentelėje ir pavaizduotas 2.4 pav.

2.4 lentelė. Panaudojimo atvejis 4

PA „Pasirinkti analizės failų vardų šabloną“		
Tikslas		Nurodyti šabloną, pagal kurį bus atrenkami programos išeities tekstai
Aprašymas		Įrašomas šablonas, pagal kurį bus atrenkami programos išeities tekstai
Prieš sąlyga		Paleista programos vartotojo sąsaja
Aktorius		Programuotojas (sistemos naudotojas)
Sužadinimo sąlyga		Vartotojas nori pasirinkti failus analizei
Susiję panaudojimo atvejai	**Išplečia PA**	„Pasirinkti failus analizei“
	Apima PA	
	Specializuoja PA	
Pagrindinis įvykių srautas		Sistemos reakcija ir sprendimai
1. Ranka įrašoma reguliarioji išraiška (arba išraiška su pakaitos simboliais (angl. *wildcard*)), kuri bus naudojama išeities tekstų failams atpažinti		
2. Vartotojas baigia PA		
Po sąlyga		Nustatytas analizės failų vardų šablonas
Alternatyvūs scenarijai		
Vartotojo įvestas šablonas neteisingas		Sistema parodo klaidos pranešimą

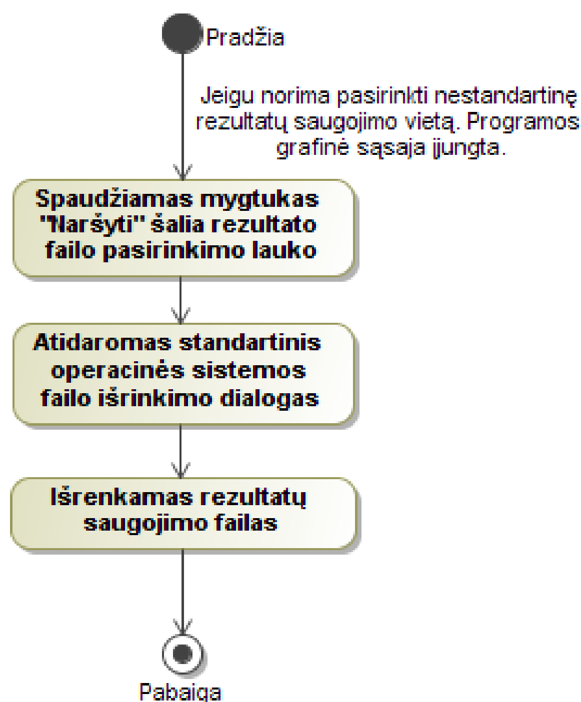


2.4 pav. PA „Pasirinkti analizės failų vardų šabloną“ veiklos diagrama

Panaudojimo atvejis „Pasirinkti rezultatų saugojimo failą“ aprašytas 2.5 lentelėje ir pavaizduotas 2.5 pav.

2.5 lentelė. Panaudojimo atvejis 5

PA „Pasirinkti rezultatų saugojimo failą“		
Tikslas		Nurodyti failą, kuriami bus saugomi programos darbo rezultatai
Aprašymas		Parenkamas failas, į kurį bus įrašomi analizės rezultatai
Prieš sąlyga		Paleista programos vartotojo sąsaja
Aktorius		Programuotojas (sistemos naudotojas)
Sužadinimo sąlyga		Vartotojas nori pasirinkti rezultatų saugojimo vietą
Susiję panaudojimo atvejai	**Išplečia PA**	„Atlikti kodo analizę“
	Apima PA	
	Specializuoja PA	
Pagrindinis įvykių srautas		Sistemos reakcija ir sprendimai
1. Parenkamas rezultatų saugojimo failas, bei katalogas, naudojant grafinę sąsają		
2. Vartotojas baigia PA		
Po sąlyga		Parinktas rezultatų saugojimo failas



2.5 pav. PA „Pasirinkti rezultatų saugojimo failą“ veiklos diagrama

NEFUNKCINIAI REIKALAVIMAI

2.6 lentelė. Nefunkcinis reikalavimas 1

Reikalavimo nr.:	1	Reikalavimo tipas:	12	Įvykis/BUP/PUC nr.:	-
Aprašymas:	Generuojant .owl failą, turi būti galimybė panaudoti visus įmanomus SCRO ontologijos elementus				
Pagrindimas:	Jei nebus panaudoti visi SCRO ontologijos elementai, nebus pasiektas maksimalus galimas analizės tikslumas				
Šaltinis:	Užsakovas				
Tikimo kriterijus:	Parašytas testavimo scenarijus parodantis, kad sistema palaiko visus įmanomus SCRO ontologijos elementus.				
Vartotojo patenkinimas:	5			Vartotojo nepatenkinimas:	2
Priklausomybės:	-			Konfliktai:	-
Papildoma medžiaga:	-				
Istorija:	Sukurta 2013-12-10				

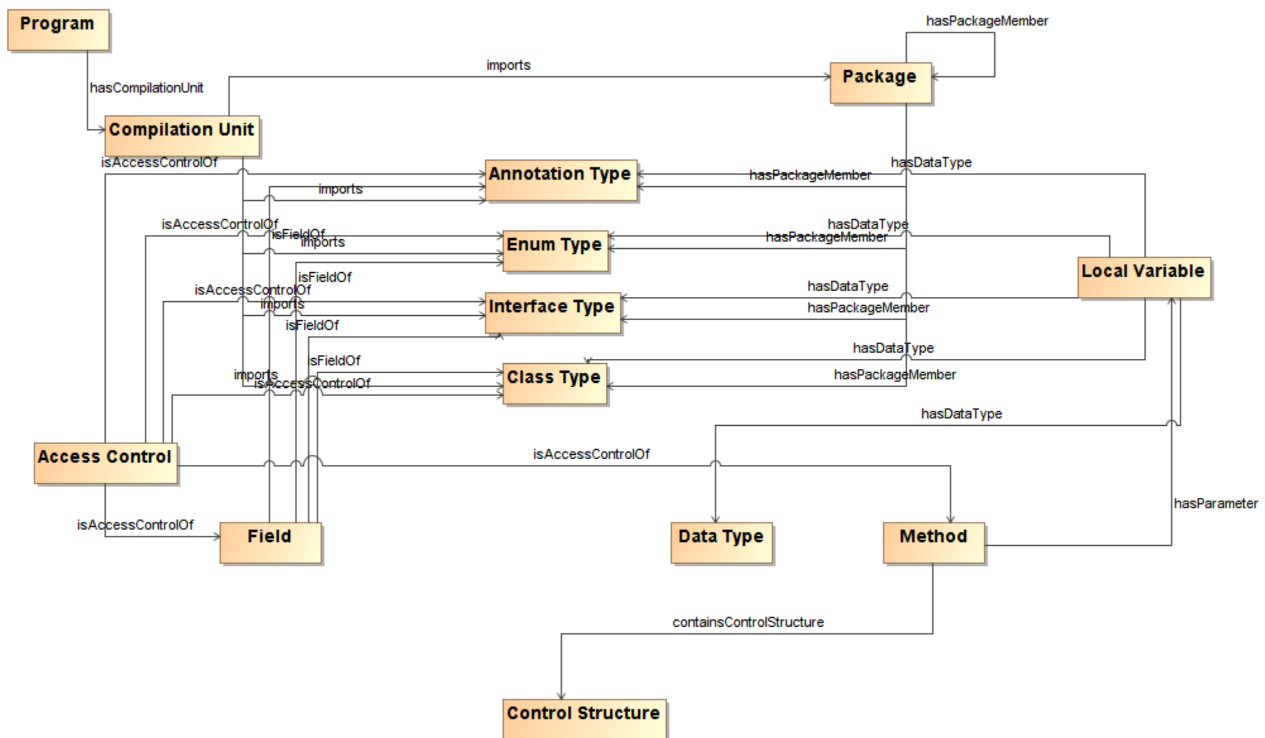
2.7 lentelė. Nefunkcinis reikalavimas 2

Reikalavimo nr.:	2	Reikalavimo tipas:	17	Įvykis/BUP/PUC nr.:	-
Aprašymas:	Sugeneruotas .owl failas turi atitikti SCRO specifikaciją				
Pagrindimas:	-				
Šaltinis:	Užsakovas				
Tikimo kriterijus:	Sugeneruotas .owl failas atitinka SCRO specifikaciją				
Vartotojo patenkinimas:	5			Vartotojo nepatenkinimas:	5

Priklausomybės:	-	Konfliktai:	-
Papildoma medžiaga:	-		
Istorija:	Sukurta 2013-12-10		

2.2. Dalykinės srities modelis

2.6 pav. pavaizduotas esminių dalykinės srities esybių modelis. Jis vaizduoja SCRO ontologijos duomenų struktūrą. Čia matome kad pagrindinės esybės yra *Program, Compilation Unit* ir *Package*. *Compilation Unit* atitinka vieną realų programos kodo failą. *Package* yra paketas, galintis savyje turėti kitus paketus, anotacijas, interfeisus, klases ir enumeracijas. Anotacijos, enumeracijos, interfeisai ir klasės gali turėti laukus, prieinamumo kontrolės raktažodžius, metodus. Metodai gali turėti prieinamumo raktažodžius, lokalius kintamuosius, kontrolės struktūras.



2.6 pav. Dalykinės srities modelis

2.3. Vartotojo sąsajos reikalavimai

Vartotojo sąsaja turi leisti įgyvendinti šiuos panaudojimo atvejus: „Pasirinkti failus analizei“, „Pasirinkti analizės failų katalogą“, „Pasirinkti analizės failų vardų šabloną“, „Pasirinkti rezultatų failo saugojimo vietą“.

Analizės failų, bei rezultato saugojimo vietos katalogai turi būti išrenkami standartinio, naudojamo operacinėje sistemoje arba grafinės vartotojo sąsajos karkase, failų išrinkimo dialogo pagalba. Failų vardų šablonas, bei paieškos rekursijos gylis įvedami ranka.

3. JAVA PROGRAMOS KODO ANALIZĖS ĮRANKIO PROTOTIPO PROJEKTAS

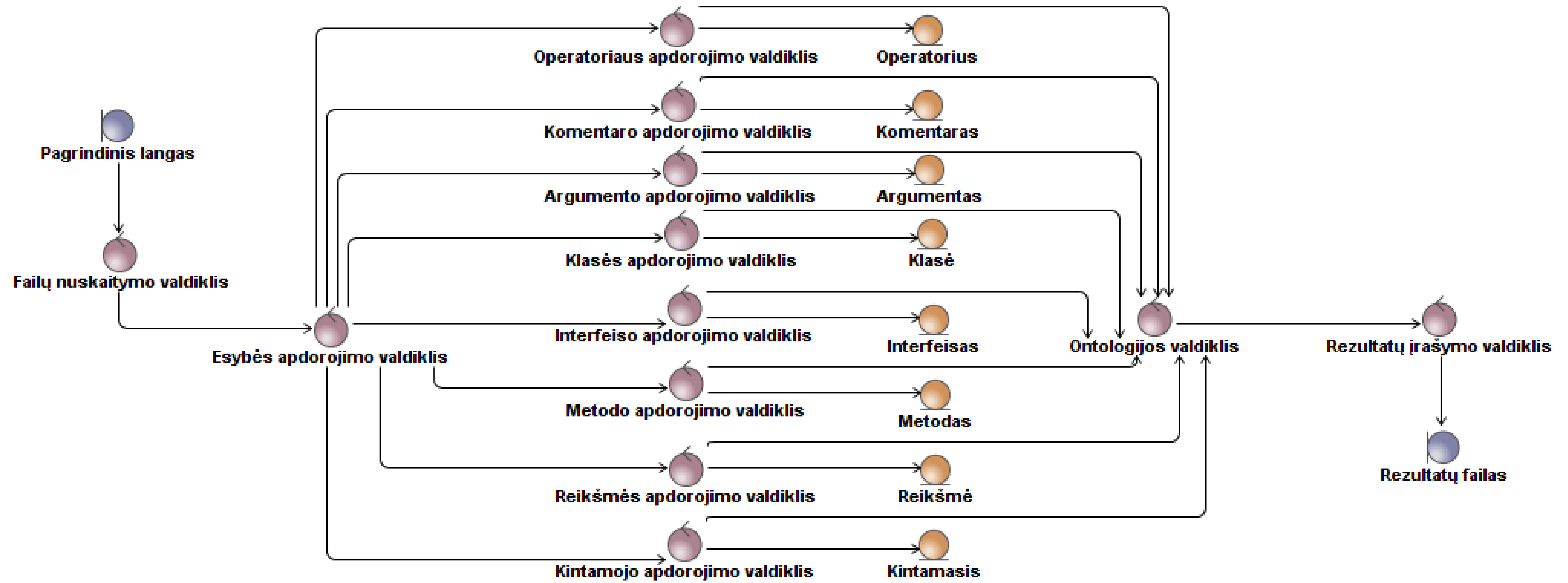
3.1. Sistemos architektūra

Šiame skyriuje pateikiamos kodo analizės įrankio prototipo projekto diagramos: reikalavimų analizės, įrankio prototipo architektūros, naudotojo sąsajos klasių, veiklos logikos klasių modeliai.

Kadangi reikalavimai suformuluoti tik kodo analizės paleidimo panaudojimo atvejui, tai projektuojamas įrankis galės daryti tik tai, be pašalinių funkcijų. Todėl projektuojamos klasės tik šiam procesui.

3.1.1. Reikalavimų analizė

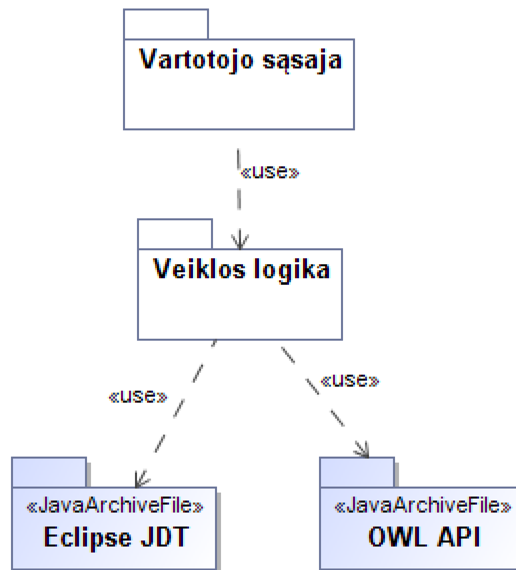
3.1 pav. pavaizduota analizės klasių diagrama, kurioje matomos esybės, valdikliai ir sistemos ribos.



3.1 pav. Analizės diagrama

3.1.2. Loginė visos sistemos architektūra

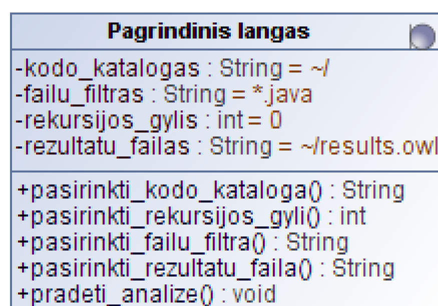
3.2 pav. vaizduojama loginė sistemos architektūra susidedanti iš vartotojo sąsajos, veiklos logikos ir trečiųjų šalių įrankių: Eclipse JDT ir OWL API.



3.2 pav. Loginė architektūra

3.1.3. Vartotojo sąsajos klasių modelis

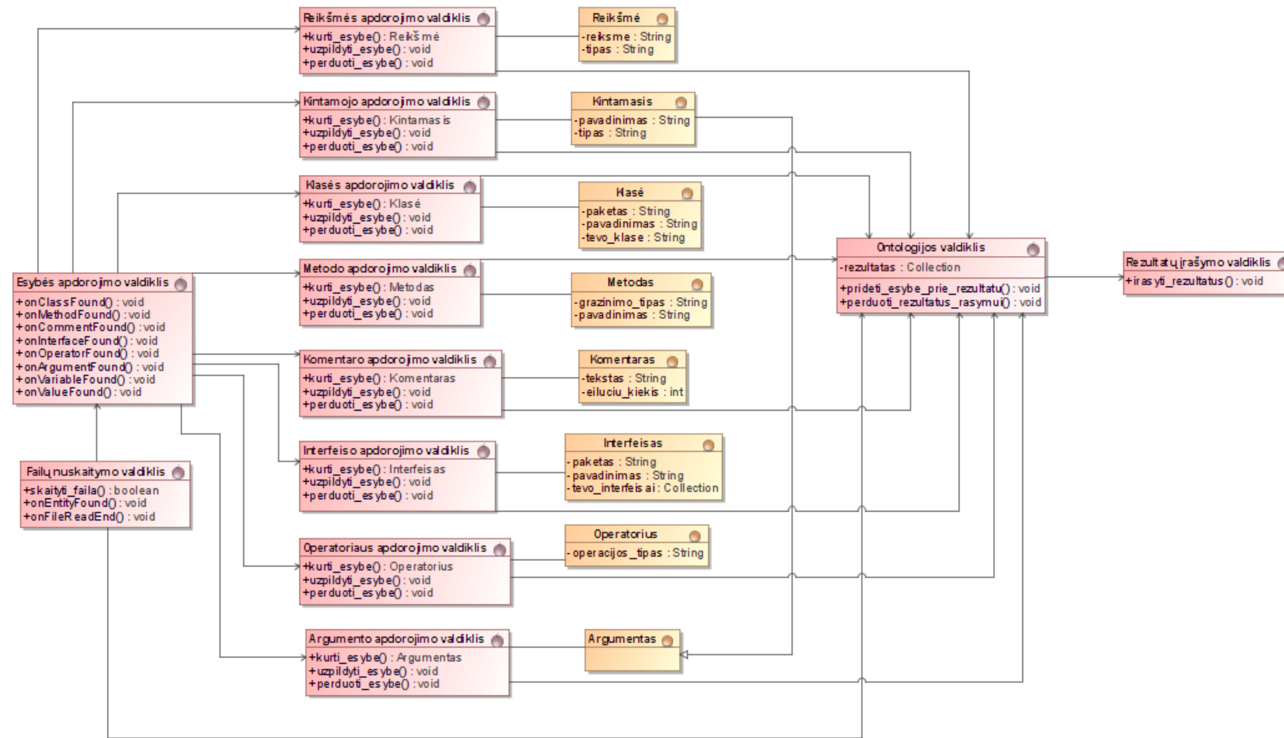
3.3 pav. matome vienintelę klasę realizuojančią vartotojo sąsają.



3.3 pav. Vartotojo sąsajos klasių modelis

3.1.4. Veiklos logikos (valdymo ir esybių klasių) modelis

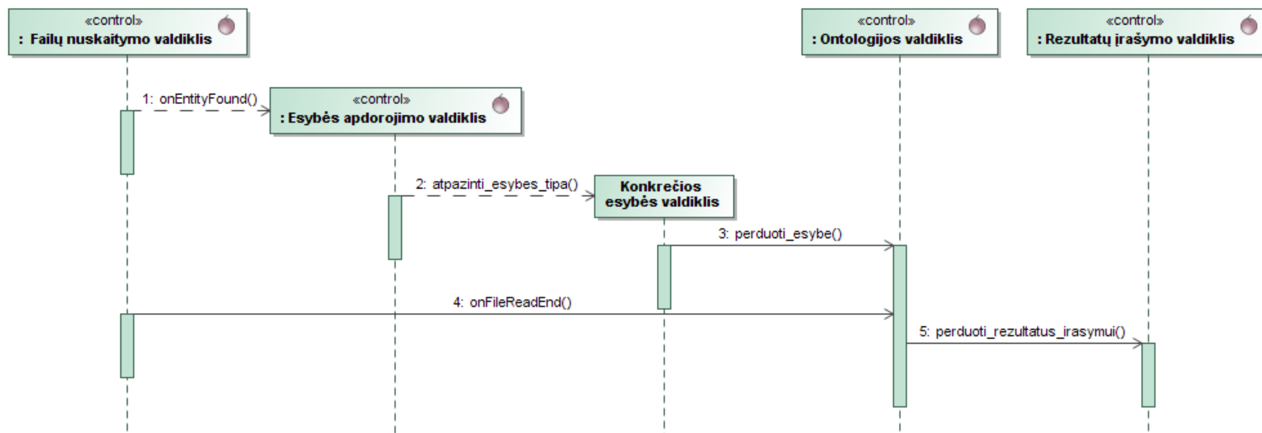
3.4 pav. vaizduojamas valdymo ir esybių klasių su atributais ir metodais modelis.



3.4 pav. Veiklos logikos modelis

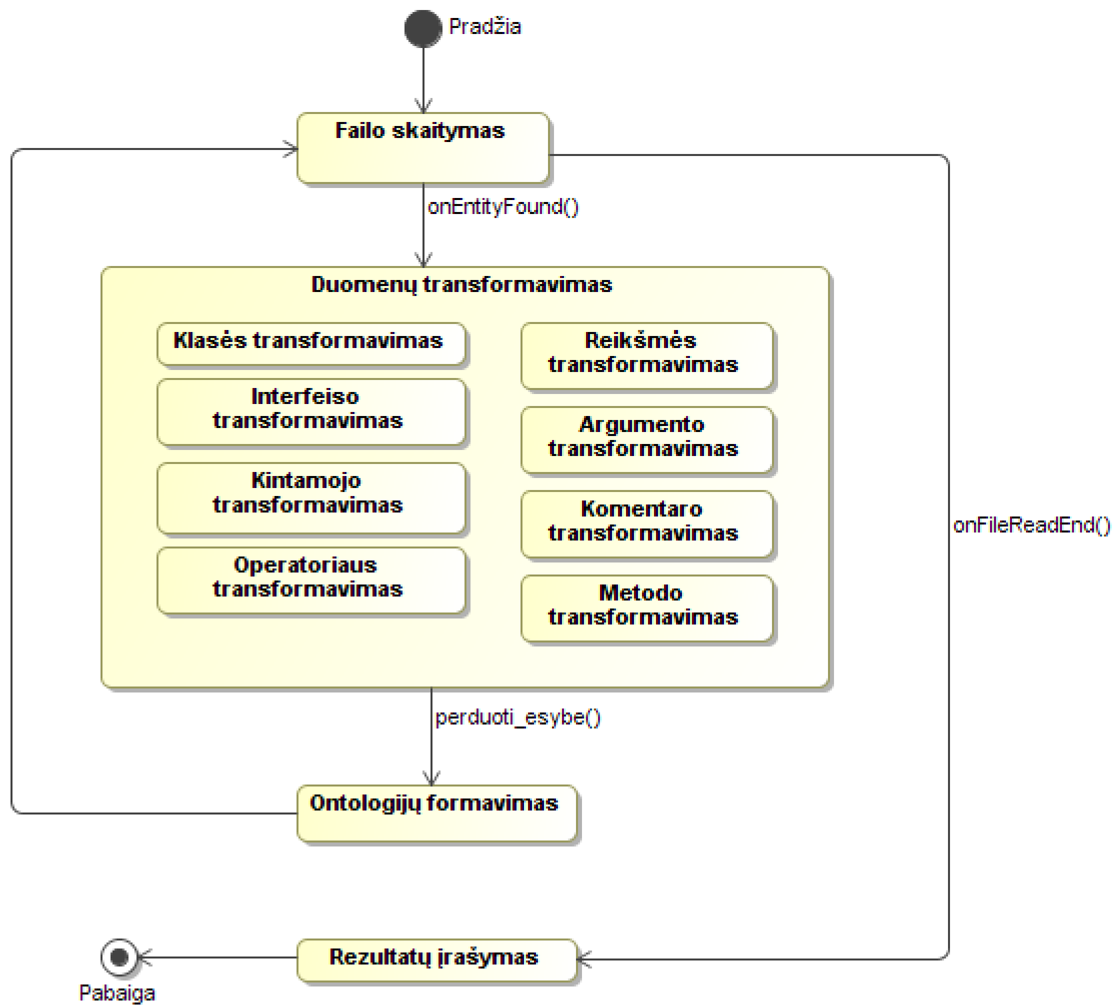
3.2. Sistemos elgsenos modelis

3.5 pav. vaizduojama supaprastinta kodo analizės sekų diagrama. „atpazinti_esybes_tipa()“ įvykio metu yra instancijuojamas naujas tam tikram esybės tipui apdoroti skirtas valdiklis. Pavyzdžiui, esybės apdorojimo valdikliui aptikus, kad esybė yra kintamasis – instancijuojamas kintamojo apdorojimo valdiklis. Jis apdoroja kintamąjį ir perduoda esybę ontologijos valdikliui. Taigi „Konkrečios esybės valdiklis“, priklausomai nuo atpažintos esybės, gali simbolizuoti argumento, interfeiso, kintamojo, klasės, komentaro, metodo, operatoriaus ar reikšmės apdorojimo valdiklius.



3.5 pav. Kodo analizės sekų diagrama

3.6 pav. vaizduojama kodo analizės būsenų diagrama – klasių perėjimai vykdant analizę.

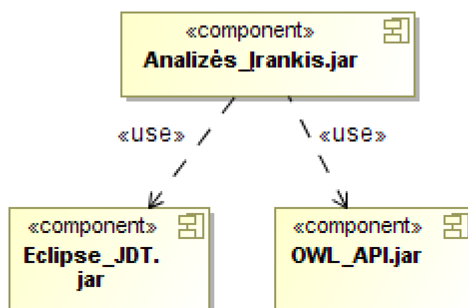


3.6 pav. Kodo analizės būsenų diagrama

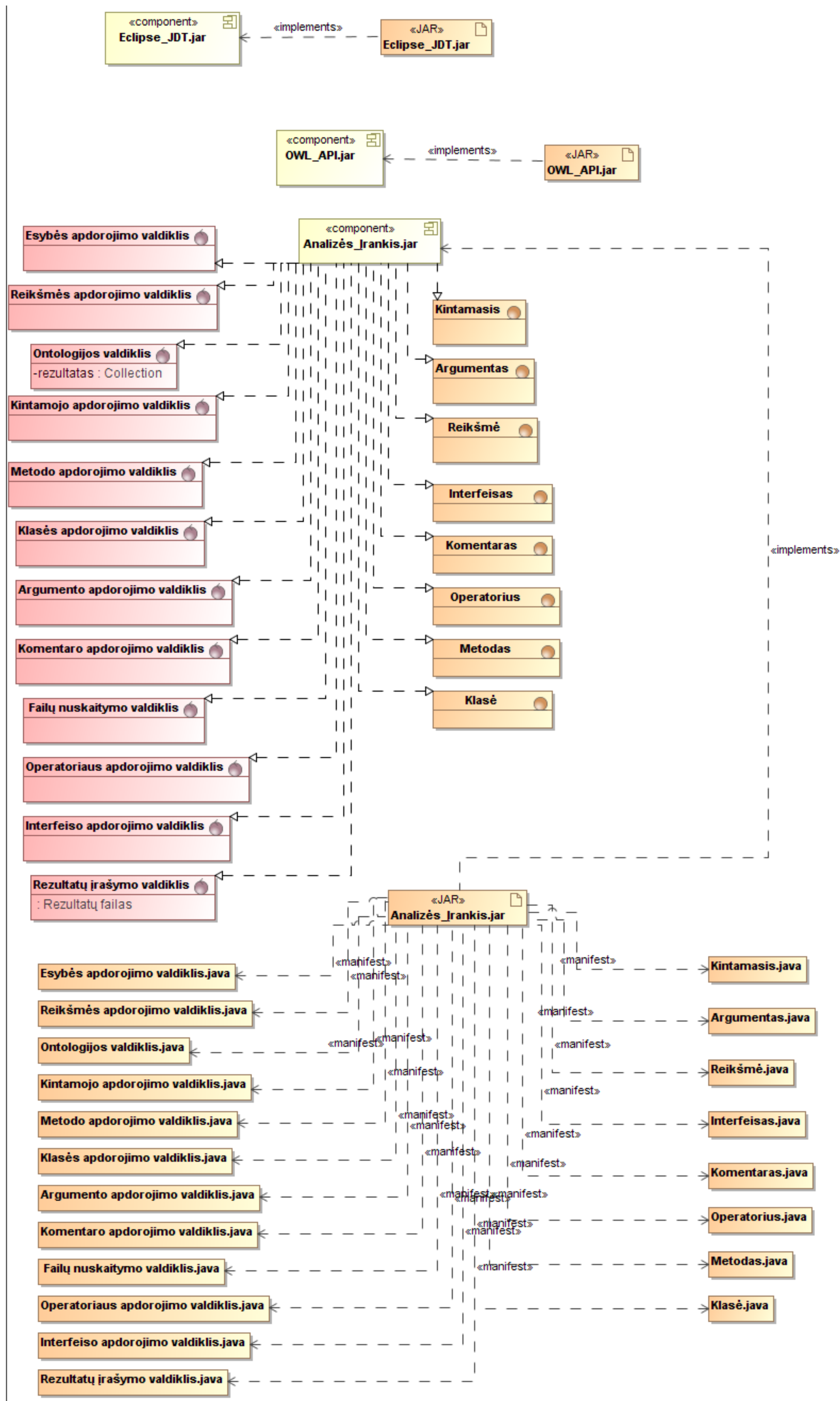
3.3. Realizacijos modelis

3.3.1. Programinių komponentų architektūra

3.7 pav. pavaizduoti komponentai, iš kurių bus sudarytas įrankis.



3.7 pav. Komponentų diagrama



3.8 pav. Komponentų realizacija artefaktais

3.8 pav. matome kokiais artefaktais bus realizuoti komponentai.

4. JAVA PROGRAMOS KODO ANALIZĖS ĮRANKIO EKSPERIMENTINIO PROTOTIPO REALIZACIJA IR TESTAVIMAS

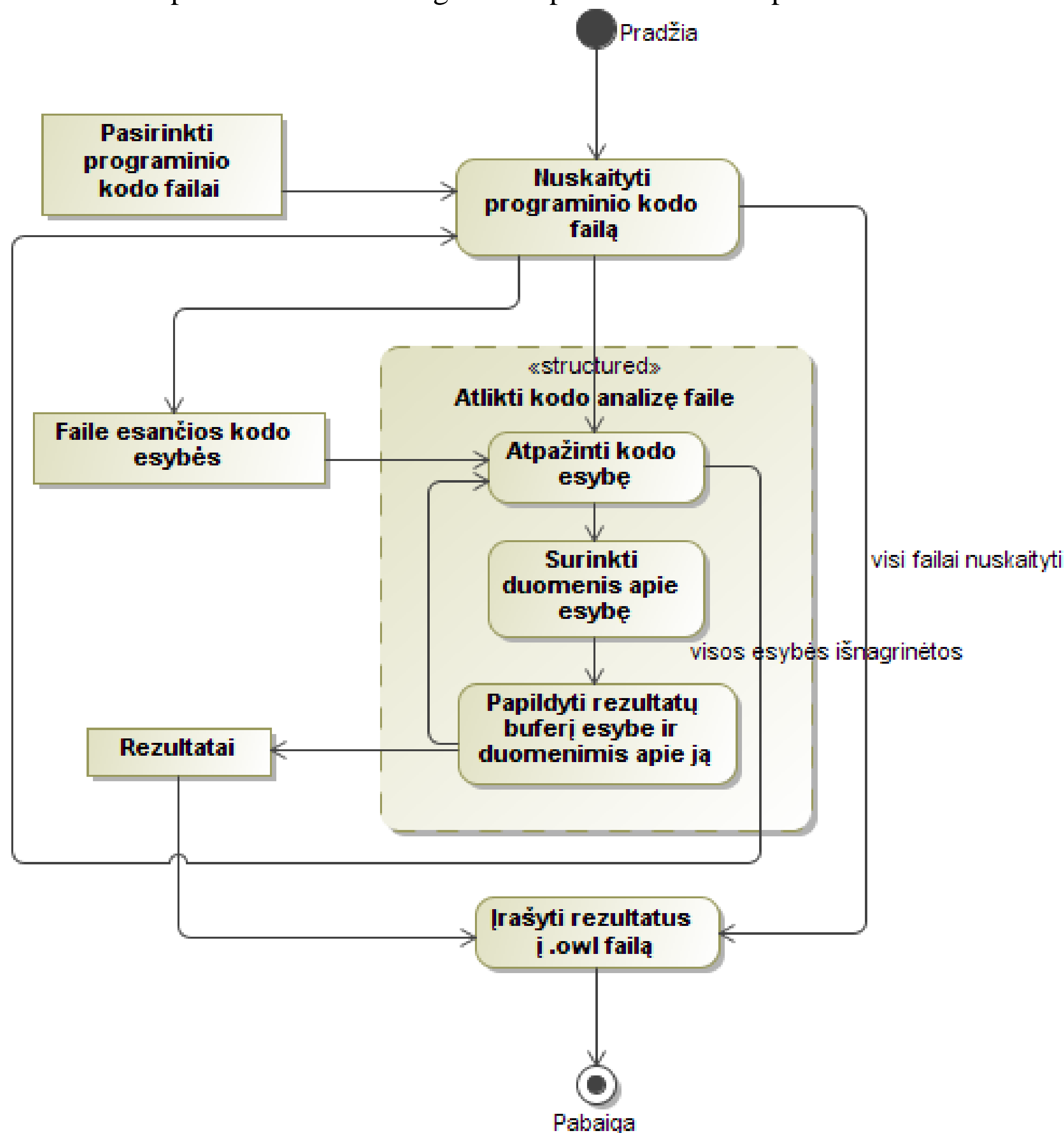
4.1. Sprendimo realizacijos ir veikimo aprašas

Sprendimas turėtų palengvinti kodo analizę, sugeneruodamas ontologijų failą, kurį vėliau galima būtų peržiūrėti ontologijų įrankiais, turinčiais daug analizės funkcijų.

Sprendimui įgyvendinti buvo panaudota Eclipse JDT biblioteka, palengvinanti Java failų skaitymą, ir OWL API biblioteka, palengvinanti darbą su ontologijomis.

SCRO ontologija leidžia išsaugoti visas Java kodo struktūras, todėl jokia informacija iš kodo neturėtų dingti transliavus failą.

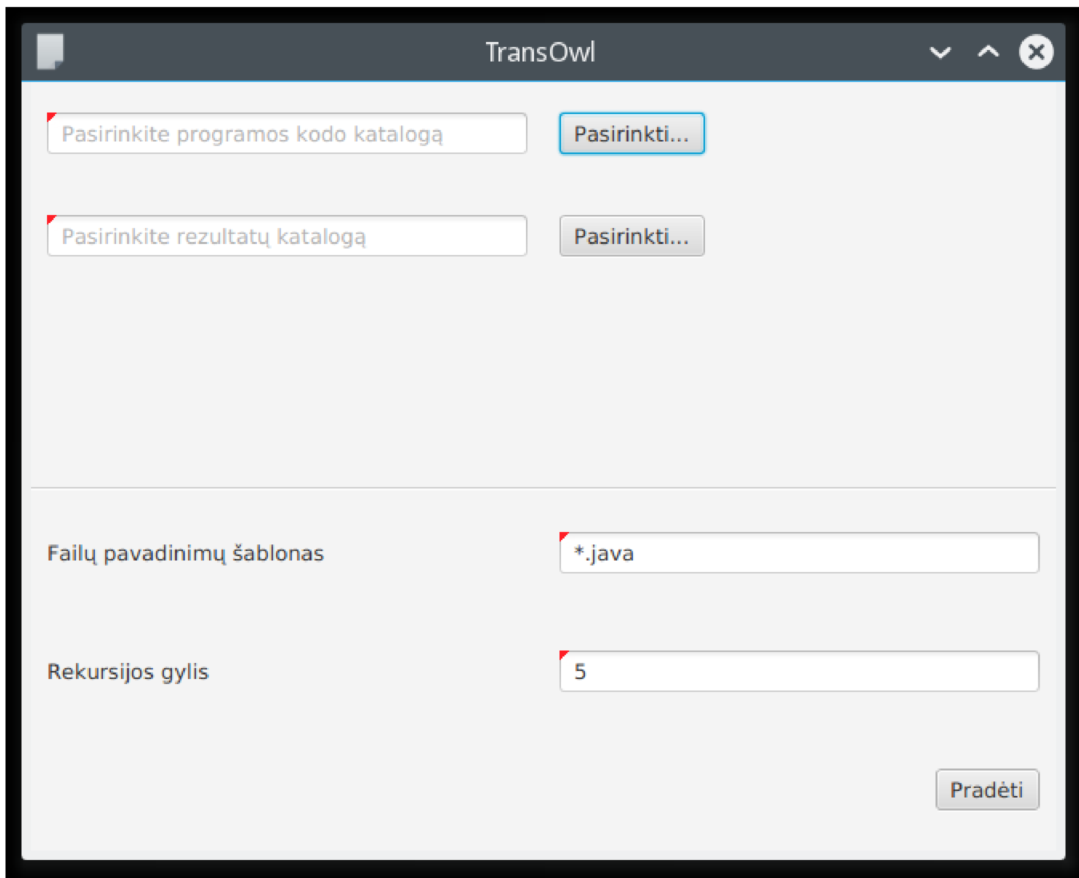
Preliminarus sprendimo veikimo algoritmas pavaizduotas 4.1 pav.



4.1 pav. Programinio įrankio veikimo algoritmas

Naudotojo patogumui sukurta grafinė aplinka (4.2 pav.), leidžianti pasirinkti:

- katalogą, kuriame saugomas analizuotinas kodas,
- katalogą, kuriame norėsime matyti programos darbo rezultatą,
- failų vardų šabloną, pagal kurį bus filtruojami failai prieš atliekant analizę,
- failų paieškos rekursijos gylį (kelių lygių pokatalogiuose bus atliekama paieška).



4.2 pav. Grafinė vartotojo sąsaja

Realizuota validacija, neleidžianti pradėti analizės be kurio nors iš privalomų laukų.

Kuriant sprendimą, panaudotos šios technologijos:

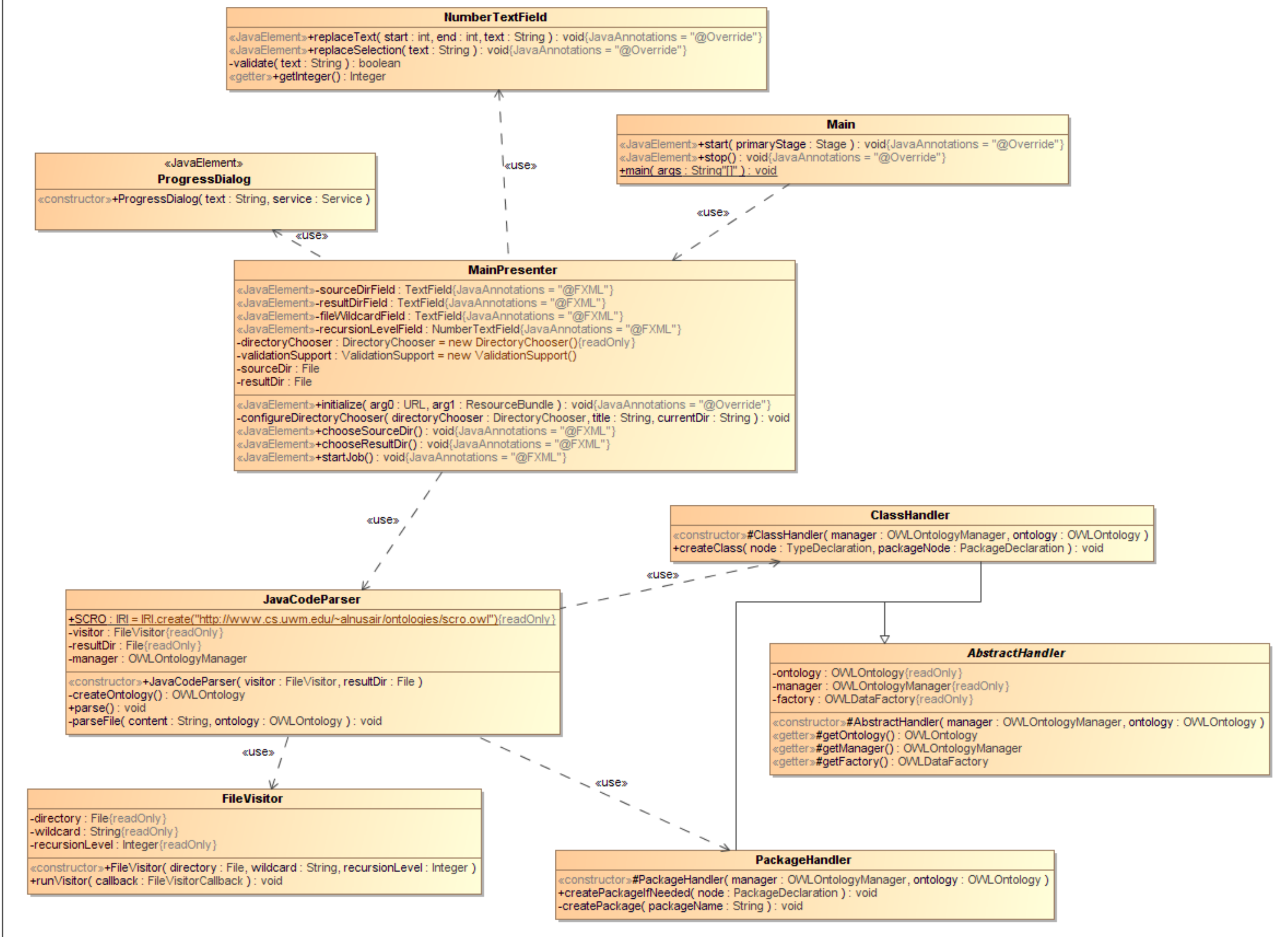
- Java 8 programavimo kalba,
- JavaFX karkasas, grafinei vartotojo sąsajai kurti,
- ControlsFX plėtiniai JavaFX karkasui, skirti supaprastinti dialogų, validacijos ir pranešimų panaudojimui,
- Maven surinkimo automatizavimo įrankis, palengvinantis darbą su projekto priklausomybėmis (paketais, reikalingais programos funkcijų realizavimui),
- JUnit vienetų testavimo karkasas,
- Eclipse JDT bibliotekos, kodo failų nuskaitymui,
- OWL API biblioteka darbui su ontologijomis.

Programos darbo rezultatas yra .owl ontologijos failas, naudojantis SCRO schemą esybių tipams aprašyti.

Pagrindinės programos sudėtinės dalys yra:

- Java kodo apdoroklis (JavaCodeParser), kuris nuskaito Java kodo failus ir naudodamas Eclipse JDT aplanko (angl. visits) kodo elementus,
- valdikliai (Handlers), kiekvienas kurių moka apdoroti kodo apdoroklio perduotus jiems atitinkamo tipo kodo elementus ir įrašyti juos į ontologiją.

Visos realizuotos klasės matomos klasių diagramoje (4.3 pav.).



4.3 pav. Realizacijos klasių diagrama

4.2. Testavimo modelis, duomenys, rezultatai

Testavimui naudojami duomenys yra Liferay projektas. Testavimui parašytas testas, apimantis visą programos veikimą, išskyrus grafinę vartotojo sąsają.

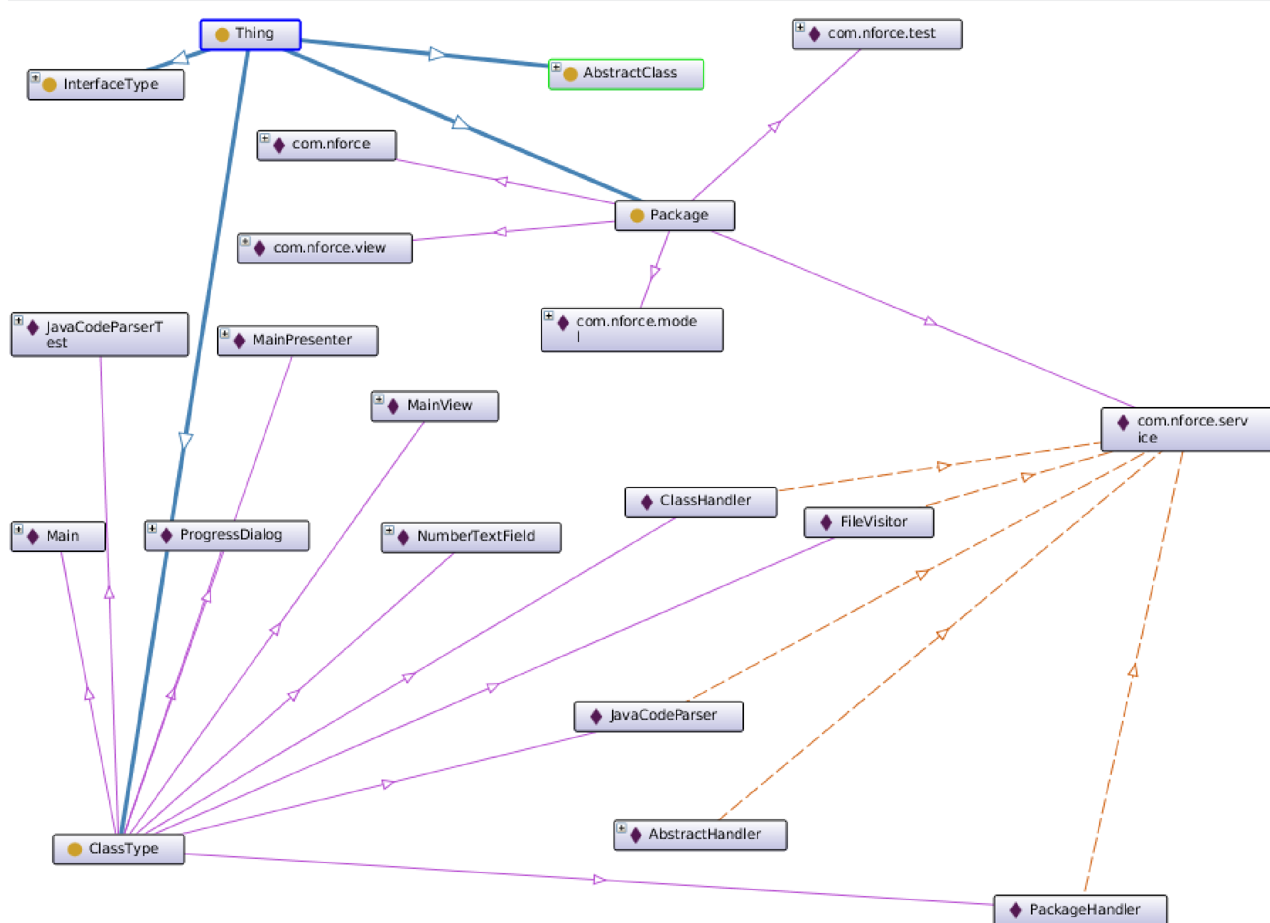
Testo kodas:

```
@Test
public void testParser() {
        FileVisitor visitor = new FileVisitor(new
File("/mnt/Storage/Work/portal-master/"), "*.java", 20);
        JavaCodeParser parser = new JavaCodeParser(visitor, new File(""));
        parser.parse();
}
```

Testavimo rezultatai pagal kokybės kriterijus:

1. atpažintų Java elementų kiekis – 15.
2. panaudotų SCRO ontologijos elementų kiekis – 16.
3. naudojamų SCRO ontologijos elementų atributų kiekis – 8.
4. analizuojamo kodo failų apimtis – 10118 failų, 3613967 eilutės. Įrankis apdorojo „Liferay“ projektą be jokių problemų.
5. analizės trukmė – 20,8 sekundžių. Įrankis „Liferay“ projektą apdorojo 15 kartų greičiau nei tikėtasi.

Testui pasibaigus, gauname *result.owl* failą, kurį atsidarę *Protege* programa, skirta darbui su ontologijomis, matome rezultatus pavaizduotus 4.4 pav.



4.4 pav. Testavimo rezultatai

5. EKSPERIMENTINIS JAVA PROGRAMOS KODO ANALIZĖS ĮRANKIO, NAUDOJANČIO SCRO ONTOLOGIJĄ, GALIMYBIŲ TYRIMAS

Su sukurtu Java programos kodo analizės įrankio prototipu (TransOWL) atliekami eksperimentai, siekiant išsiaiškinti jo darbo spartą, praktines panaudojimo galimybes.

Darbo našumo eksperimente svarbūs kintamieji yra analizuojamos programos Java programinio kodo failų kiekis, eilučių skaičius ir laikas, kurio prireikė analizei atlikti.

Panaudojimo galimybių eksperimente išbandomos SPARQL užklausos.

Svarbu paminėti, kad eksperimentas buvo atliktas naudojant kompiuterį, kurio techniniai duomenys yra:

- Intel Core i7 4710HQ procesorius (maksimalus taktinis dažnis 3.5GHz, 4 fiziniai branduoliai, HyperThreading technologijos palaikymas),
- 16 gigabaitų DDR3 1600MHz operatyviosios atminties (RAM)

Naudojant kompiuterį su kitokiais parametrais, rezultatai gali skirtis.

DARBO NAŠUMO EKSPERIMENTAS

Darbo našumo eksperimentas atliekamas su trimis Java projektais:

1. Java programinio kodo analizės įrankis, naudojantis SCRO ontologiją TransOWL, kuris ir yra šio magistro baigiamojo projekto rezultatas,
2. Mokslo, Inovacijų ir Technologijų Agentūros informacinė sistema MTEPIS,
3. atviro kodo portalas Liferay.

5.1 lentelėje matome projektų apimtį: mažiausias iš bandytų projektų yra TransOWL, kuris sudarytas iš 17 Java kodo failų ir 874 eilučių. Tuo tarpu didžiausias yra Liferay portalo projektas, sudarytas iš maždaug 10 tūkstančių Java kodo failų ir 3,6 milijono kodo eilučių.

5.1 lentelė. Projektų apimtis

	TransOWL	MTEPIS	Liferay
Failų kiekis	17	2574	10118
Eilučių kiekis	874	221846	3613967

5.2 lentelėje matome išmatuotą Java programinio kodo analizės trukmę. Kadangi išmatuota labai skirtingos apimties projektų trukmė, galime susidaryti aiškią nuomonę apie įrankio veikimo greitaveiką. Didžiausio projekto analizė užtruko vos 22 sekundes, taigi įrankis dirba tikrai greitai.

5.2 lentelė. Analizės trukmė

	TransOWL	MTEPIS	Liferay
Bandymas 1	0,168 s	6,064 s	20,528 s
Bandymas 2	0,140 s	5,968 s	22,918 s
Bandymas 3	0,070 s	6,064 s	23,313 s
Bandymas 4	0,058 s	6,359 s	21,922 s
Bandymas 5	0,300 s	5,988 s	22,943 s
Bandymas 6	0,067 s	6,410 s	23,267 s
Bandymas 7	0,091 s	6,373 s	22,656 s
Bandymas 8	0,156 s	6,011 s	22,136 s
Bandymas 9	0,053 s	6,182 s	22,220 s
Bandymas 10	0,097 s	6,270 s	22,062 s
Vidurkis	0,120 s	6,169 s	22,434 s

5.3 lentelėje matome analizės rezultatų failų dydžius.

5.3 lentelė. Rezultatų failų dydžiai

	TransOWL	MTEPIS	Liferay
Failo dydis	60,6 KB	15,2 MB	149,7 MB

5.4 lentelėje matome, kiek laiko užtrunka atidaryti gautus rezultatų failus su Protege programine įranga, skirta darbui su ontologijomis. Atidarymo trukmė tiesiogiai priklauso nuo rezultatų failo dydžio, tačiau net ir su pačiu didžiausiu, Liferay rezultatų failų, truko vos 12 su puse sekundės.

5.4 lentelė. Rezultatų failų atidarymo trukmė

	TransOWL	MTEPIS	Liferay
Laikas	0,046 s	2,52 s	12,64 s

PANAUDOJIMO GALIMYBIŲ EKSPERIMENTAS

Darbui su ontologijomis įrankis *Protege* leidžia kodo analizės įrankio prototipo darbo rezultate atlikti SPARQL, DL-Query užklausas, skirtas išgauti bet kokią ontologijoje esančią informaciją.

Sekantis kodo fragmentas yra SPARQL užklausa, grąžinanti visas *Singleton* šabloną atitinkančias klases, t.y. visas klases, kurių konstruktorius yra privatus.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX scro: <http://www.cs.uwm.edu/~alnusair/ontologies/scro.owl#>

SELECT ?class WHERE {
      ?classType rdfs:subClassOf scro:ClassType .
      ?class a ?classType .
      ?constructor scro:isConstructorOf ?class .
      ?constructor scro:hasAccessControl scro:PrivateModifier .
}
```

Panaudojus šią užklausą TransOWL projekto analizės rezultate gautoje ontologijoje, gauname, kad *Singleton* šabloną atitinka viena klasė: StatisticsBean. Toliau pateikiamas šios klasės kodas:

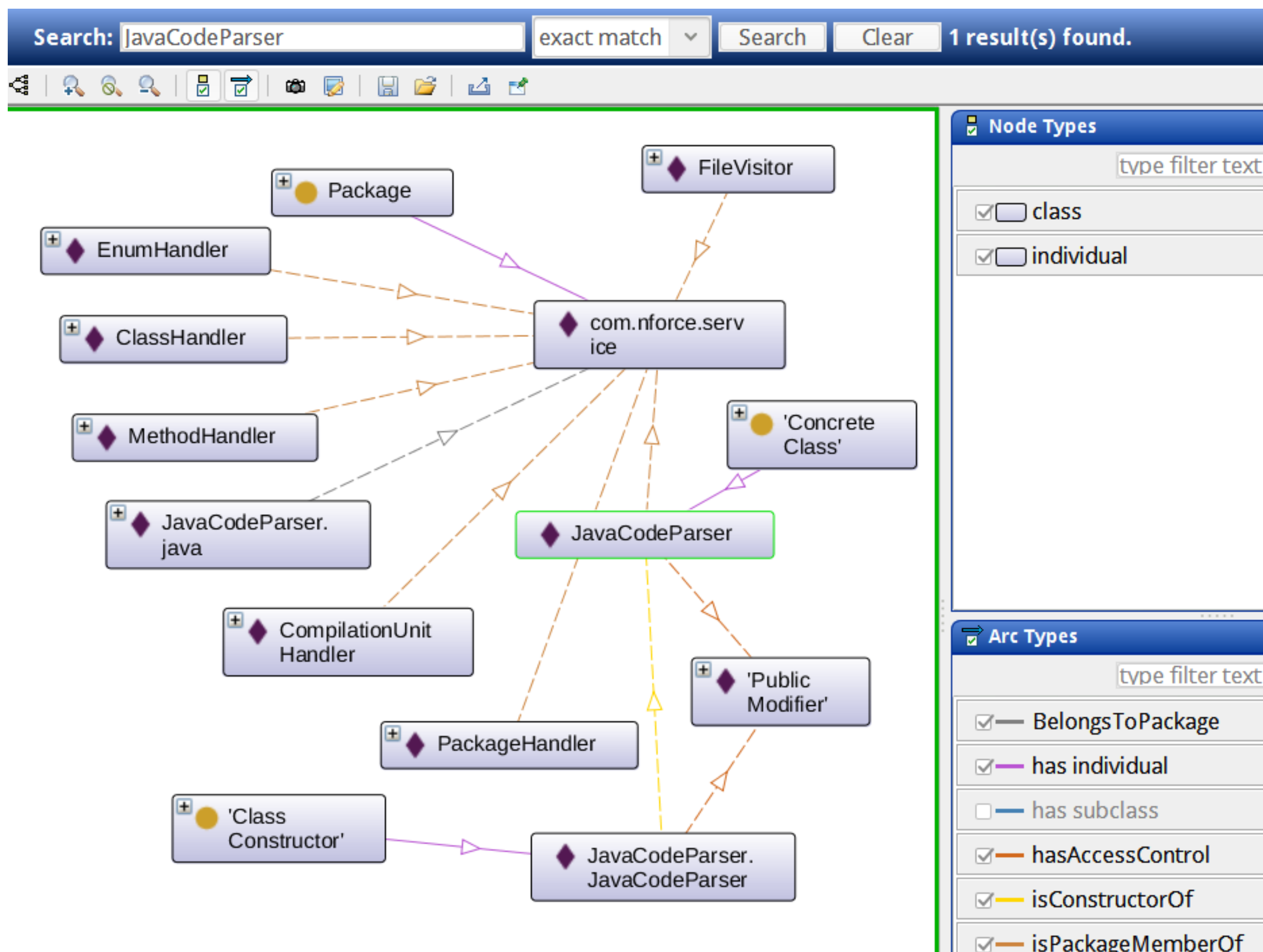
```
public class StatisticsBean {

          private static StatisticsBean instance;

          private Map<String, Integer> files = new HashMap<>();

          private StatisticsBean() {
                    // let's be singleton
          }

          static public StatisticsBean getInstance() {
                    if (instance == null) {
                              instance = new StatisticsBean();
                    }
                    return instance;
          }

          public void addFile(String fileName, String content) {
                    files.put(fileName, getLinesCount(content));
          }

          private Integer getLinesCount(String content) {
```

```
			if (content == null) {
				return null;
			}
			String lines[] = content.split("\n");
			return lines.length;
		}

		public void printSummary() {
			int totalFiles = files.size();
			int totalLines = 0;
			for (String file : files.keySet()) {
				if (files.get(file) != null) {
					totalLines += files.get(file);
				}
			}
			System.out.println(String.format("Total   %d   files   were
analysed", totalFiles));
			System.out.println(String.format("Total   %d   lines   were
found", totalLines));
			System.out.println("---------------------------------------
--------------------------------");
		}

		public void clear() {
			files.clear();
		}
}
```

Nenaudojant ontologijų, tą pačią informaciją galima būtų pasiekti peržiūrėjus visus kodo failus su klasėmis ir patikrinus ar klasė turi konstruktorių, ir ar jis privatus. Esant daug klasių, toks darbas užimtų daug laiko. Peržiūrint daug klasių kyla tikimybė kažką praleisti, kažko nepastebėti, taigi gali būti surastos ne visos *Singleton* šabloną atitinkančios klasės.

Kaip matome, užklausa nėra sudėtinga, tačiau leidžia pasiekti informaciją, kuri nenaudojant ontologijų būtų prieinama tik po žmogaus atliktos analizės.

5.1 pav. matome, kaip Protege leidžia atvaizduoti informaciją apie kokį nors konkretų ontologijos elementą. Šiame pavyzdyje ieškojome klasės JavaCodeParser ir matome tokią informaciją: klasės metodai, prieinamumo raktažodžiai, paketas, kuriame yra klasė, kitos klasės esančios tame pačiame pakete.



5.1 pav. Grafinis informacijos apie kodo elementą atvaizdavimas

EKSPERIMENTO APIBENDRINIMAS

Atlikus eksperimentus matome, kad įrankis palengvina programinio kodo supratimą, jo rezultatus galima peržiūrėti įvairiomis darbui su ontologijomis skirtomis programomis. Dėka naudojamos SCRO ontologijos, palaikomi visi svarbiausi Java programinio kodo elementai. 5.5 lentelėje matome, kad naujai sukurtas įrankis atitinka visus pirmame skyriuje keltus kriterijus: analizės rezultatams saugoti naudojamos ontologijos, dėka Eclipse JDT bibliotekų palaikomas Java programinio kodo automatinis nuskaitymas, dėka SCRO ontologijos palaikomi visi svarbiausi Java programinio kodo elementai. Kadangi naudojamos ontologijos – galima naudoti įvairias užklausas, gauti įvairiai informacijai apie kodą – nuo priklausomybių tarp klasių iki įvairių dizaino šablonų panaudojimų. Taigi programinio kodo supratimą įrankis palengvina.

5.5 lentelė. TransOWL palyginimas su esamais sprendimais

	OBPCM	CodePro Analytix	JArchitect	JDepend	SCRO	Eclipse JDT	TransOWL
Ar naudojamos ontologijos	+	–	–	–	+	–	+
Ar palaikoma Java programavimo kalba	+	+	+	+	+	+	+

	OBPCM	CodePro Analytix	JArchitect	JDepend	SCRO	Eclipse JDT	TransOWL
Ar naudojamas automatinis kodo nuskaitymas	+	+	+	+	-	+	+
Ar palaikomi visi svarbiausi Java kodo elementai	-	+	+	-	+	+	+
Ar sprendimas palengvina programinio kodo supratimą	-	-	+	-	+	-	+
Ar analizės rezultatus galima peržiūrėti trečiųjų šalių įrankiais	+	-	-	+	+	-	+

6. IŠVADOS

1. Atlikus analizę, pamatėme kad kuriant šiuolaikinę, sudėtingą programinę įrangą, labai svarbu yra naudotis statinės kodo analizės įrankiais. Jie leidžia aptikti klaidas dar nesukompiliavus ir neišbandžius programos, palengvina kitų programuotojų parašyto kodo supratimą, kai programos kūrime dalyvauja didelis programuotojų kiekis. Programinių įrankių (tiek komercinių, tiek nemokamų), skirtų klaidų paieškai statiškai analizuojant kodą yra daug, jie veikia gerai, todėl kuriamas analizės įrankis buvo orientuotas kodo vizualizavimo ir supratimo palengvinimo kryptimi. Vizualizuojant kodą patogu remtis ontologijomis, nes jos nėra priklausomos nuo konkrečios programavimo kalbos ir gali tarnauti kaip meta-modelis, kuriame nėra prarandamos svarbios kodo ypatybės.
2. Tyrimas parodė, kad *Ontology-based program comprehension model* ontologijos detalumo neužtenka norint patobulinti statinės kodo analizės procesą, tačiau SCRO ontologija yra pakankamai detali tyrimo tikslui pasiekti. Eclipse JDT panaudojimas padėjo sukurti kodo analizės įrankio prototipą, nes Eclipse JDT yra labai galingas įrankis skaitant ir apdorojant Java kodą, palaikantis visas įmanomas kalbos struktūras, tačiau darbas su juo yra palyginti paprastas ir nereikalaujantis gilaus Eclipse architektūros išmanymo.
3. Buvo sukurtas automatinis statinės kodo analizės įrankis, gebantis atvaizduoti Java programinį kodą SCRO ontologijos elementais.
4. Eksperimentas parodė, kad įrankis geba greitai apdoroti didelius kiekius Java kodo. Gauti rezultatų failai taip pat greitai gali būti atidaromi darbo su ontologijomis programa Protege. Protege aplinkoje galima vykdyti SPARQL užklausas, padedančias išsiaiškinti įvairias programos kodo savybes, galima peržiūrėti kodo elementus ir ryšius tarp jų grafiniu būdu.
5. SCRO ontologija nėra pririšta prie konkrečios programavimo kalbos, dėl to, remiantis šiame darbe sukurtu įrankiu, galima sukurti statinės kodo analizės įrankius ir kitoms objektinio programavimo kalboms. Taip pat panaudojus šio įrankio rezultatus, galima būtų dalinai transliuoti kodą į kitas programavimo kalbas – būtų galima generuoti klases, metodus, kintamuosius. Rezultatų failą galima analizuoti įvairiais pjūviais programomis, skirtomis darbui su ontologijomis. Tai palengvina kodo supratimą, todėl galima teigti kad magistro darbo tikslas buvo pasiektas.

7. LITERATŪRA

[1] A. Alnusair, „Indiana," 7 11 2012. [Tinkle]. Available: http://www.indiana.edu/~awny/index.php/research/ontologies/10-scro. [Kreiptasi 10 2013].

[2] Google Inc., „CodePro Analytix," Google Inc., 2010. [Tinkle]. Available: https://developers.google.com/java-dev-tools/codepro/doc/. [Kreiptasi 10 05 2015].

[3] Y. ZHANG, „An Ontology-Based Program Comprehension Model, a Thesis," 2007. [Tinkle]. Available: http://users.encs.concordia.ca/~haarslev/students/Yonggang_Zhang.pdf. [Kreiptasi 10 2013].

[4] CoderGears, „JArchitect.com," CoderGears, 2010. [Tinkle]. Available: http://jarchitect.com/documentation. [Kreiptasi 12 04 2015].

[5] Wikipedia, „JArchitect," [Tinkle]. Available: http://en.wikipedia.org/wiki/JArchitect. [Kreiptasi 02 04 2015].

[6] Clarkware Consulting, Inc., „JDepend," Clarkware Consulting, Inc., 2009. [Tinkle]. Available: http://clarkware.com/software/JDepend.html. [Kreiptasi 10 05 2015].

[7] A. Alnusair ir T. Zhao, „Retrieving Reusable Software Components Using Enhanced Representation of Domain Knowledge," 2011. [Tinkle]. Available: http://www.indiana.edu/~awny/images/publications/alnusair_lncsbook_2011.pdf. [Kreiptasi 20 05 2015].

[8] L. Vogel, „vogella.com," 08 08 2012. [Tinkle]. Available: http://www.vogella.com/tutorials/EclipseJDT/article.html. [Kreiptasi 10 2013].

[9] O. T. Thomas Kuhn, „eclipse.org," 11 2006. [Tinkle]. Available: http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html. [Kreiptasi 10 2013].

[10] A. Alnusair ir T. Zhao, „Using Ontology Reasoning for Reverse Engineering Design Patterns," įtraukta *Models in Software Engineering*, Springer Berlin Heidelberg, 2010, pp. 344-358.

[11] A. Saffari, „Detecon," 02 2011. [Tinkle]. Available: http://www.detecon.com/ru/files/Opinion-Paper-Next-Generation-Mobile-Application-Management-2011.pdf. [Kreiptasi 12 04 2015].