VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ KATEDRA

Universalus pertvarkų įrankis

Universal refactoring tool

Magistro baigiamasis darbas

Atliko:	Stasys Peldžius	(parašas)
Darbo vadovas:	j. m. d. Mindaugas Plukas	(parašas)
Recenzentas:	lekt. Mindaugas Žilinskas	(parašas)

Vilnius – 2009

Santrauka

Vykstant nuolatiniam programų sistemų atnaujinimui, nuolatos reikia prižiūrėti, kad jos būtų kokybiškai projektuojamos ir programuojamos. Tačiau neišvengiamai atsiranda nekokybiško išeities teksto, arba atsiranda projektavimo trūkumų. Todėl yra svarbu mokėti ieškoti tokias problemas, ir jas ištaisyti.

Šio darbo tikslas yra sukurti automatinio – universalaus įrankio modelį, kuris savarankiškai aptiktų pertvarkas, bei būtų nepriklausomas nuo konkrečios programavimo kalbos. Šiam tikslui pasiekti yra nagrinėjami mokslininkų siūlomi automatiškai aptinkantys pertvarkas metodai. Taip pat yra nagrinėjamos tokio įrankio realizavimo galimybės, pateikti realizaciniai sprendimai ir pavyzdžiai. Taip pat tikslas yra sukurti praktiškai naudingas automatines pertvarkas, kurios būtų realizuotos pasiūlytu įrankiu, ir pademonstruotas jų veikimas.

Šis įrankis naudoja loginį programavimą, kurio faktais yra aprašomos pertvarkomos programos, o taisyklėmis – pačios pertvarkų programos. Sėkmingai sukurti automatiniai pertvarkų radimo pavyzdžiai, leidžia daryti išvadą, kad šiame darbe rastas būdas automatiškai aptikti nekokybišką išeities tekstą, bei realizuoti tokias pertvarkas nepriklausomai nuo programavimo kalbos.

Summary

In the continual evolution of software systems, should be continuous to ensure that they are high quality designed and programmed. But inevitably the defective code, that call “bad small” or the design deficiencies. It is therefore important to be able to find such problems, and to correct them.

The aim of this thesis is to create automatic - universal refactoring tool, which is detected in self-refactoring, and is independent of specific programming languages. To achieve this objective are scientists considered the proposed automatic detection of refactoring methods. It is also considered the possibility of the realization of such a tool, to provide examples and realizable solutions. It also aims to create a practical benefit of the automatic adjustments to the proposed tool is to be realized, and a demonstration of their operation.

This tool uses a logic programming, which is a factual description of the conversion, and the rules - the refactoring of the program. The successful creation of automatic detection for refactoring, it can be concluded that this work is found way to automatically detect poor quality of source code, and the realization of the restructuring, regardless of programming language.

Turinys

Įvadas ... 5

1. Nekokybiškas kodas ir pertvarkos ... 6

1.1. Nekokybiškas kodas ... 7

1.2. Pertvarkos ... 8

1.3. Pertvarkymo eiga ... 10

1.4. Pertvarkų radimo būdai ... 10

1.4.1. Pertvarkų radimas naudojant matus ... 11

1.4.2. Pertvarkos naudojant grafus ... 13

1.4.3. Loginis programavimas pertvarkoms aptikti ... 16

2. Pertvarkų kalba ... 22

2.1. Pertvarkų duomenys ... 23

2.1.1. MSIL ... 23

2.1.2. XML panaudojimas ... 25

2.1.3. Loginė kalba ... 26

2.2. Pertvarkų programos ... 27

2.3. Pertvarkų rezultatai ... 31

3. Universalus įrankis ... 32

3.1. Duomenų generatorius ... 32

3.2. Rezultatų interpretatorius ... 35

4. Pertvarkų pavyzdžiai ... 37

4.1. Paketų ciklai ... 37

4.2. Metodų matomumas ... 43

4.2.1. Nekokybiško programavimo atsiradimas ... 44

4.2.2. Metodų matomumo pertvarkos programa ... 45

4.3. Rezultatai ... 53

Išvados ir pasiūlymai ... 55

Literatūros sąrašas ... 56

1 priedas. SOUL LiCoR predikatai ... 58

2 priedas. Paketų ciklų pertvarkos predikatai ... 59

3 priedas. Metodų matomumo pertvarkų predikatai ... 61

Įvadas

Programų sistemos po sukūrimo, palaikymo stadijoje turi būti nuolat plečiamos, tobulinamos pagal nuolat besikeičiančius reikalavimus. Šis palaikymas gali nepalankiai paveikti jų kokybę. Pertvarkos (angl. refactoring) yra viena iš svarbiausių ir dažniausiai naudojamų technologijų, kad pagerintų programinės įrangos kokybę, kuri gali būti išmatuota, naudojant įvairius matus [SS07]. Pertvarkų radimui yra naudojami įvairūs būdai, vykdomi moksliniai tyrimai, o komercinių įrankių, kurie atitiktų visus keliamus reikalavimus – nėra.

Dauguma komercinių ir nemokamų pertvarkų įrankių patys neieško galimai nekokybiško kodo, o jei ieško, tai tik labai paprastų, elementarių (pavyzdžiui, deklaruotas, bet nenaudojamas kintamasis). Jie tik padeda programuotojui įvykdyti pertvarką, kurią pats programuotojas pasirenka, taip pat pats programuotojas turi surast, kurioje vietoje ją reikia taikyti. Tokie įrankiai negarantuoja, kad įvykdžius pertvarką sistemos funkcionalumas liks nepakitęs, atsakomybė lieka programuotojui [CMM06]. Tai gi jie ne tik nesuranda automatiškai pertvarkų vietų, bet ir automatiškai negali jų įgyvendinti. Todėl jų taikymas yra labai ribotas, jie tik palengvina programuotojui perrašymo darbą, bet sudėtingiausią – pertvarkų aptikimo darbą turi atlikti pats programuotojas.

Mokslininkai kuria tokius metodus, kurie patys aptinka nekokybišką kodą. Šie metodai pasiūlo atitinkamą pertvarką, ir taip pat kokybiškai ją pritaiko, kad nebūtų pažeistas sistemos funkcionalumas. Tačiau jie yra skirti arba tik tam tikrom pertvarkom aptikti, arba tik tam tikroje konkrečioje programavimo kalboje. Vienas toks sprendimas yra naudoti loginį programavimą, tam yra kuriamas įrankis SOUL, kuris aptinka pertvarkas Smalltalk programose. Šis įrankis tenkina svarbiausią idėja – automatiškai aptikti pertvarkas, tačiau jo trūkumas yra pririšimas prie šios programavimo kalbos parašytų programų.

Šio darbo tikslas būtų sukurti automatinio – universalaus įrankio modelį, kuris savarankiškai aptiktų pertvarkas, bei būtų nepriklausomas nuo konkrečios programavimo kalbos. Kuriant šį modelį bus remiamasi mokslininkų siūlomais automatiškai aptinkančiais pertvarkas metodais. Šiame darbe yra nagrinėjamos tokio įrankio realizavimo galimybės, pateikti realizaciniai sprendimai ir pavyzdžiai. Taip pat bus sukurtos praktiškai naudingos pertvarkos, kurios būtų realizuotos pasiūlytu įrankiu, ir pademonstruotas jų veikimas.

Šio magistrinio darbo rezultatas būtų suprojektuotas universalus – automatinis pertvarkų įrankis, kuris būtų nepriklausomas nuo pertvarkomos programos programavimo kalbos, bei pasiūlytos praktiškos pertvarkymo programos šiam įrankiui su konkrečiais pavyzdžiais.

1. Nekokybiškas kodas ir pertvarkos

Kadangi dabar daugelis sistemų yra nuolatos atnaujinamos, vyksta nuolatinė jų evoliucija, kyla naujų problemų, kurių anksčiau nebūdavo. Dabar sistemų versijos išeina labai dažnai, net ir nedideli projektai yra nuolatos atnaujinami, nes nuolatos kinta reikalavimai. Būtent tokioje aplinkoje ir atsiranda galimybė kristi projekto kokybei. [Fow00] knygoje pertvarkos apibrėžiama kaip vidinės programinės įrangos struktūros pakeitimas, kad būtų lengviau suprantama, ir pigiau modifikuojama, nekeičiant esminių programų sistemos savybių. Toje pačioje knygoje pertvarkymo procesas vadinamas pertvarkų taikymas nekeičiant programinės įrangos funkcionalumo. Straipsnyje [SS07] nagrinėjama, ar pertvarkų taikymas išties pagerina programinės įrangos kokybę, ir išvadose autoriai teigia, kad pertvarkos yra vienas svarbiausių metodų programinės įrangos kokybės gerinimui, kurį galima išmatuoti.

Remiantis [EM02] straipsniu nekokybišku kodu galima vadinti programų dalis, kurios yra netinkamai suprojektuotos, arba taikytos netinkamos programavimo praktikos. Pertvarkos yra naudojamos norint pašalinti nekokybišką kodą. Nekokybiškas kodas gali atsirasti visur, todėl ir jų sprendimų yra įvairių. [Opd92] darbe apibrėžtos pagrindinės keturios pertvarkų grupės nekokybiškam kodui panaikinti:

1) klasių pertvarkos: pridėti, pervadinti, ištrinti klasę, konvertuoti į vaikinę – tėvinę klasę;
2) klasių kintamųjų pertvarkos: pridėti, pervadinti, ištrinti kintamąjį, perkelti į kitą hierarchijos lygį kintamąjį;
3) metodų pertvarkos: perkelti į kitą klasę, pervadinti, ištrinti metodą, pridėti, ištrinti parametrą, iškelti į tėvinę, ar vaikinę klasę;
4) kodo pertvarkos: iškirpti metodo dalį ir sukurti naują metodą, sulieti kelis metodus, panaikinti kodo dubliavimą ir t.t.

Anot [Fow00] pertvarkos turėtų būti atliekamos, kai yra pridedamas naujas funkcionalumas, arba, kai reikia rasti klaidą, nepertvarkytas programas gali būti sudėtinga testuoti, pavyzdžiui, jei tarp paketų yra ciklai, arba tiesiog reikia atlikti pertvarkas per programos išeities tekstų peržiūras. Daug pertvarkų ir nekokybiško kodo yra pristatyta [Fow00] knygoje, taip pat [Fow08] internetiniame puslapyje yra nuolatos atnaujinama informacija apie naujausias sugalvotas pertvarkas. Norint sėkmingai aptikti ir pašalinti nekokybišką kodą, reikia mokėti jo ieškoti, ir žinoti, kaip jį reikia pataisyti. Kituose skyreliuose tai pristatoma plačiau.

1.1. Nekokybiškas kodas

Nekokybiškas kodas, kaip apibrėžta [Fow00] knygoje, pasižymi tam tikromis blogomis savybėmis, kurios netrukdo gerai funkcionuoti programai, tačiau blogina skaitomumą, apsunkina klaidų paiešką, ar kodo atnaujinimą. Dažnai nekokybiškas kodas atsiranda laikinai, suvokiant, kad jis atsiranda, ir iškart jis būna panaikinamas. Pavyzdžiui diegiant tam tikrą naują funkcionalumą, dažnai jis būna realizuojamas keliuose dideliuose metode, o tik vėliau pertvarkoma į keletą metodų, ar net klasių. Toliau pateikiama keletą nekokybiško kodo pavyzdžių iš [Fow00] knygos.

Kodo dubliavimas (angl. Duplicated Code) turbūt yra pats kritiškiausias nekokybiškas kodas, kuris sukelia daugiausia vargo ateityje modifikuojant programą. Šios klaidos reikėtų vengti praktiškai be išimties. Paprasčiausiai išsprendžiama ši problema, kai kodas dubliuojasi dviejuose metoduose toje pačioje klasėje. Tada tiesiog užtenka įvykdyti metodų išplėtimą, ir perkelti kodą iš abiejų metodų.

Kita dubliavimo problema būna, kai kodas kartojasi vaikinėse to paties tėvo klasėse. Tokiu atveju reikia apjungiant tuos metodus atlikti išplėtimą ir, tada jį pakelti į tėvinę klasę. Jie metodai nėra visiškai vienodi, bet atlieka tą patį loginį algoritmą, reikia suvienodinti algoritmą, ir atlikti prieš tai aprašytus veiksmus.

Kita galimai probleminė vieta gali būti per ilgi parametrų sąrašai. Struktūriniame programavime parametrais būna perdavinėjama viskas, ko reikia tai procedūrai, ar funkcijai. Tai buvo įprasta, nes alternatyvūs globalūs kintamieji yra dažniausiai bloga praktika ir dažnai sukelia daug nepatogumų, ypač nepatyrusiems programuotojams. Objektinės kalbos iš esmės pakeitė šią situaciją, nes visko, ko reikia metodui nėra būtina perdavinėti per parametrus, nes jis tiesiog gali atitinkamų metodų paprašyti iš kitų objektų. Dažniausiai visko, ko reikia metodui, jis gali rasti savo klasėje. Objektiniame programavime parametrų sąrašai patartina, kad būtų kuo trumpesni lyginant su struktūrinių programavimu. Tai yra gerai, nes ilgus parametrus sunku suprasti, nes jie būna surašomi nenuosekliai, ir sunku įsiminti ir išskirti juos logiškai, ir nuolat reikės juos keisti, jei bus norima perduoti daugiau duomenų. Geriausia daugumą parametrų panaikinti, jei įmanoma juos gauti metodo viduje keliais iškvietimais naujų metodų, ar objektų sukūrimu.

Vienas sprendimų būdų, norint sumažinti parametrų skaičių, yra naudoti parametrų keitimo metodais pertvarkymą. Taip pat galima perkelti parametrus į naują klasę, jei tie parametrai dažnai kartojasi įvairiuose metoduose. Jei vis tiek parametrai būna ilgi, ir labai dažnai tenka naudoti pertvarkas, reikia bandyti pergalvoti klasių priklausomybių ryšius, ir bandyti patobulinti klasių hierarchiją.

1.2. Pertvarkos

Praeitame skyrelyje buvo peržvelgtos keletas dažniausiai pasitaikančių nekokybiško kodo vietų. Prie kiekvieno iš jų buvo minimas ir galimas sprendimo būdas. Šiame skyriuje jie bus peržvelgti detaliau. Jie jau būna ir automatizuoti, nes programuotojas pats nusprendžia, kaip jam taisyti nekokybišką kodą, ir jam tiesiog užtenka jį pažymėti ir pasirinkti reikiamą įrankio teikiamą pertvarkymą. Automatiniai įrankiai gali automatiškai pasiūlyti tik elementarius pertvarkymus, kuriems nereikia papildomo programuotojo apsisprendimo, pavyzdžiui, apibrėžti kintamieji, kurie niekada nenaudojami, arba perspėja, kad naudojam neinicijuotą objektą, kurio reikšmė gali būti neapibrėžta. Programuotojas gali pasinaudoti patarimu, ir ištaisyti kodą, arba turėti kitokią nuomonę ir jį ignoruoti.

Metodų išplėtimo (angl. Extract Method) būdas tarp programuotojų yra vienas populiariausių ir dažniausiai naudojamų. Tiesiog peržvelgiant kodą, jei matoma, kad tam tikras metodas yra per ilgas, arba, jei jaučiamas poreikis komentuoti tam tikrą kodą, reikia tas vietas iškelti į atskirą metodą, kurio pavadinimas paaiškins metodo atliekamą darbą. Tuo yra padidinamas našumas, nes yra didelė tikimybė, kad tam tikrą kodą gali tekti panaudoti ir kitose situacijose. Šio metodo taikymo žingsniai:

- pirmiausia sukuriamas naujas metodas, kurio pavadinimas atspindi ne kaip metodas realizuoja užduotį, bet ką realizuoja;
- nukopijuoti reikalingą kodą iš pirminio metodo į naują;
- išskirti lokalius kintamuosius pagal naujus sukurtus metodus;
- sukurti reikiamus parametrus, kurie būtini naujo metodo funkcionalumui;
- buvusioje kodo vietoje parašyti kreipinį į metodą, bei rezultato nuskaitymą.

Pavyzdys:

Kodą, kuris yra savarankiškas nuo likusios kodo dalies, galima iškelti į naują metodą:

```
void printOwing(double amount) {
printBanner();
//print details
System.out.println ("name:" + _name);
System.out.println ("amount" + amount);
}
```

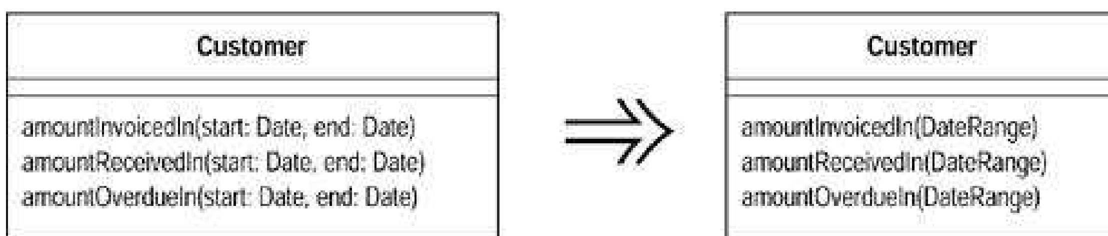
Iškėlus gaunama:

```
void printOwing(double amount) {
printBanner();
printDetails(amount);
}
void printDetails (double amount) {
```

```
System.out.println ("name:" + _name);
System.out.println ("amount" + amount);
}
```

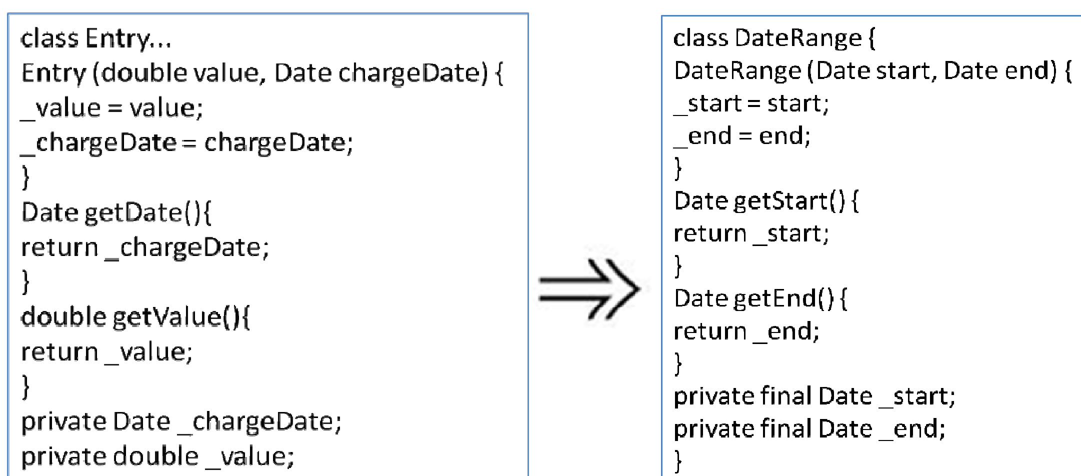
Tokiu atveju naujai gautą metodą galima panaudoti ir daugelyje kitų metodų, taip ne tik sutaupoma eilučių, bet ir supaprastinamas keitimas, jei, pavyzdžiui, reikia atspausdinti papildomą informaciją, tai tektų tik pataisyti metodą.

Taip pat dar vienas pavyzdys yra parametrų objektų kūrimas (angl. Introduce Parameter Object). Jei turima parametrų aibė, kuri dažnai naudojama kartu, reikia pakeisti jas objektu, kuris atspindės visus tuos parametrus.



1 paveikslėlis. Parametrų pakeitimas objektu

Labai dažnai taip būna, kai grupė parametrų turi tendenciją dažnai būti naudojami metodų aprašymuose drauge, tai gali būti keliuose metoduose ir net keliose skirtingose klasėse. Ši pertvarka yra ypač naudinga ir efektyvi, nes sutrumpina parametrų skaičių iki vieno, pakeičiant juos objektu, juk ilga parametrų eilutę yra ganėtinai sunku suprasti, ir valdyti. Dar vienas didelis privalumas yra naujo parametro įvedimas, jei prireikia naujo duomens metodui, tokiu atveju užtenka papildyti klasę, o pačių metodų parametrų perrašinėti nereikia, pavyzdys 2 paveikslėlyje.



2 paveikslėlis. Pertvarkoma klasė

1.3. Pertvarkymo eiga

Norint sėkmingai pritaikyti šiuos pertvarkymus, reikia vadovautis keliais paprastais pasiūlymais, kurie padės padidinti kodo kokybę [Fow00]:

- Aptikimas
  - Identifikuoti dalį kodo, kuriai yra galimybė pritaikyti tam tikrą pertvarkymo metodą, tai dažniausiai atliekama intuityviai, nors šiame darbe siekiama tai automatizuoti.
- Pertvarkymo kaina
  - Įvertinti, ar pakeitimo nauda kompensuos pakeitimo sąnaudas, ar apsimoka kurti naują pertvarkos programą, jei tokios dar nėra.
- Pertvarkos rengimas
  - Paruošti kodą pertvarkymui.
- Įvykdymas
  - Įvykdyti pertvarkymo metodą (pasitelkiant įrankį, arba rankiniu būdu).
- Vystymas
  - Patikrinti, ar sistema išsaugojo visas savo savybes (testuoti).

Dabar galima bandyti savarankiškai peržiūrinėti savo kodą, ir ieškoti nekokybiško kodo, ir jį sėkmingai perdaryti.

1.4. Pertvarkų radimo būdai

Blogų kodo variantų yra labai daug, vieni iš jų daro didesnę žalą, kiti mažesnę, tačiau norima sukurti metodus, kaip būtų galima blogą kodą aptikti automatiškai, nes dažnai tai gali būti sudėtingas ir varginantis darbas. Praktiškai automatizuotas pertvarkų palaikymas yra esminės svarbos, kadangi pertvarkų įgyvendinimas yra daug darbo reikalaujantis ir linkęs į klaidas procesas, jei jį atlieka pats programuotojas. Idealus pertvarkų įrankis turėtų:

- Automatiškai identifikuoti vietas, kuriose reikia pritaikyti pertvarkas;
- Vizualizuoti programas, ar jų dalis, kurias norima pertvarkyti;
- Automatiškai įvykdyti pertvarkymą nurodytai išeities teksto daliai;
- Vaizduoti pertvarkos rezultatą.

Šiame poskyryje pristatomi mokslininkų nagrinėjami automatizuoti pertvarkų radimo būdai, kuriuos galima pritaikyti ir praktiškai. Pasinaudojus jų stipriosiomis pusėmis, ir išvengiant jų trūkumų, bus projektuojamas universalus pertvarkų įrankis, kuris yra šio darbo tikslas. Tačiau jokie

įrankiai viso to neatlieka pilnai, tačiau yra keletas metodų, kurie įgyvendina keletą šių savybių. Toliau pateikiami tie metodai, kurie labiausiai atitinka iškeltus reikalavimus.

1.4.1. Pertvarkų radimas naudojant matus

Pirmas nagrinėjamas būdas remiasi matais. Šiame poskyryje analizuojama [DDN00, EL96, SLL99, SSL01] medžiaga. Šiuose straipsniuose nagrinėjama galimybė pritaikyti matus (angl. metrics), kai kurių pertvarkų automatiniam aptikimui, arba pasiūlyti, kuriose vietose, kokias pertvarkas galima panaudoti. Nagrinėjami keturi dažniausiai naudojami pertvarkymai: metodų / laukų perkėlimas, bei klasių suliejimas arba išskyrimas (angl. Move Method, Move Attribute – Field, Extract Class, Inline Class).

[SSL01] darbe parodyta, kad matai gali padėti identifikuoti tam tikras anomalijas, kurioms galima pritaikyti pertvarkas. Tačiau autoriai teigia, kad visgi programuotojai turi būti paskutiniai, kurie nusprendžia, ar pritaikyti pertvarką, ar ne. Vis dėlto yra būdų imituoti žmogaus intuiciją pakankamai efektyviai. Autoriai mano, kad vizualus programų būsenos vaizdavimas padės geriau programuotojams nuspręsti, ar reikia taikyti pertvarkas, nes iš išeities teksto ne visada tai paprastai pastebima. Šis galimas automatizavimo būdas remsis keturiais minėtais pertvarkymais, į kuriuos įeina klasės, metodai ir laukai.

Daugelis pertvarkų, tame tarpe ir nagrinėjamos, yra paremtos principu: sujunk kartu, kas turi būti kartu. Priklausymui kartu laipsniui nustatyti yra naudojama daug matų. Vienas pagrindinių iš jų yra atstumu paremto ryšio matas, kuris remiasi panašumų ir skirtumų teorija. Kitaip sakant, šis matas apibrėžia panašumą tarp dviejų esybių pagal bendrai turimų savybių kiekį. Tariant, kad B yra aibė visų galimų savybių, kurios priklauso esybėms tam tikru požiūriu. Tokiu būdu yra įmanoma apibrėžti atstumo apskaičiavimo formulę:

$$dist_{B}(x,y) := 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|}$$

, kur x ir y yra esybės, o p(x) yra x savybių skaičius aibėje B.

Visos keturios nagrinėjamos pertvarkos yra susijusios su ryšiais. Savybės, kurias naudosime nagrinėjamose pertvarkose turi būti toje pačioje klasėje, kaip metodai, ar kintamieji. Šias pertvarkas galima nagrinėti atstumais tarp klasės atributų. Naudojant šį matą galima sudaryti atstumų matricą tarp klasių metodų ir laukų. Nagrinėjamos bus šios dvi klasės:

class class_A	class class_B

```
{
public:
static void methodA1()
{ attributA1=0;
methodA2();}
static void methodA2()
{ attributA2=0;
attributA1=0;}
static void methodA3()
{ attributA1=0;
attributA2=0;
methodA1();
methodA2();}
static int attributA1;
static int attributA2;
}
```

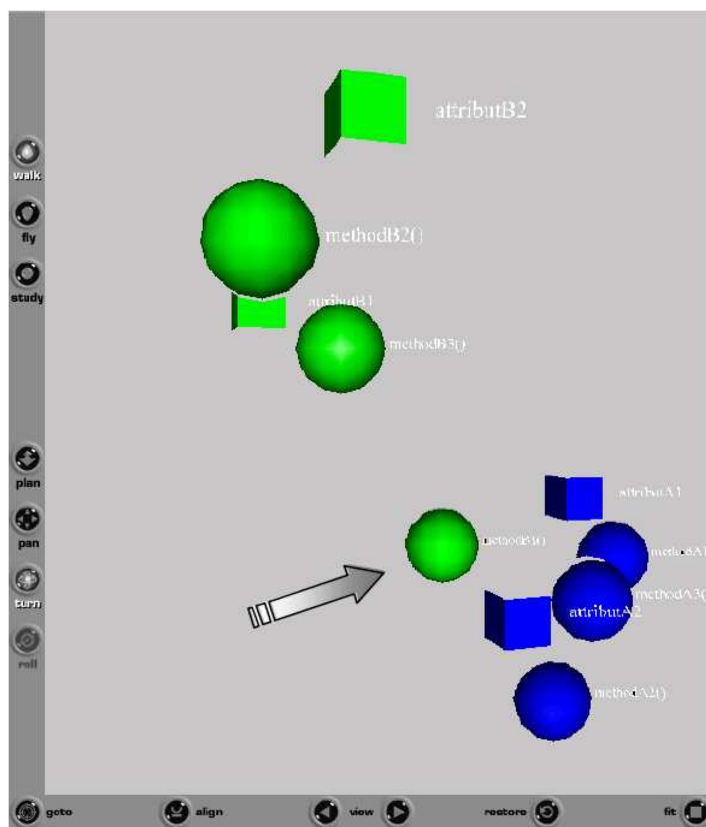
```
{
public:
static void methodB1()
{ class_A::attributA1=0;
class_A::attributA2=0;
class_A::methodA1();}
static void methodB2()
{ attributB1=0;
attributB2=0;}
static void methodB3()
{ attributB1=0;
methodB1();
methodB2();}
static int attributB1;
static int attributB2;
}
```

Vizualiai matosi, kad metodas methodB1() yra „blogo kvapo“, nes jis naudojamas tik klasėje class_A, nors yra apibrėžtas klasėje class_B. Todėl šiam metodui reiktų pritaikyti metodo perkėlimo pertvarką. Tačiau galima tai ir automatizuoti. Tam reikia nustatyti atstumus tarp šešių metodų ir keturių laukų. Rezultatai gali būti atvaizduoti atstumų matrica (1 lentelė).

1 lentelė. Atstumų matrica

	mA1()	mA2()	mA3()	mB1()	mB2()	mB3()	aA1	aA2	aB1	aB2
mA1()	0									
mA2()	0.5	0								
mA3()	0.4	0.	0							
mB1()	0.6	0.6	0.5	0						
mB2()	1	1	1	1	0					
mB3()	1	1	1	0.86	0.6	0				
aA1	0.5	0.67	0.33	0.5	1	0.88	0			
aA2	0.83	0.6	0.5	0.67	1	0.86	0.5	0		
aB1	1	1	1	1	0.5	0.25	1	1	0	
aB2	1	1	1	1	0.33	0.8	1	1	0.75	0

Šiuos atstumus atvaizdavus vizualiai naudojant VRML-word įrankį. Žalios figūros vaizduoja class_A, mėlynos – class_B. Metodai vaizduojami sferomis, laukai – kubais, gauname tokį vaizdą:



3 paveikslėlis. VRML-word

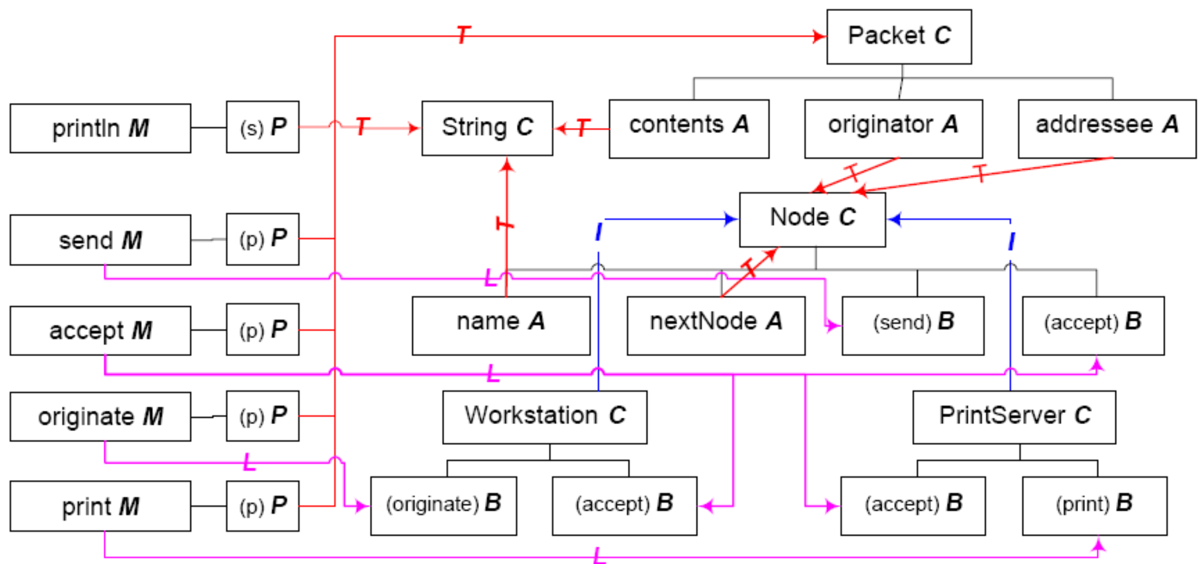
Iš jo akivaizdžiai matoma, kas yra negerai. Rodykle pažymėtas method_B1(), kurio ryšys su klase class_A yra daug stipresnis, nei su klase class_B, todėl akivaizdžiai matosi, kad jį reikia perkelti iš klasės class_B į klasę class_A.

Lygiai tokiu pačiu principu galima sudaryti ir kitus atstumus, pagal kitus kriterijus, ir gauti naujus vaizdus, pagal kurias galima spręsti apie kitus pertvarkymus. Šiuos atstumus apskaičiuoti galima automatizuotai, taip pat įrankis gali pasiūlyti tam tikrą pertvarką, tačiau iš vizualaus atvaizdavimo galima ir pačiam programuotojui gana efektyviai ir kokybiškai nustatyti galimas pertvarkas. Pavyzdžiui, iš 3 paveikslėlio vaizdo aiškiai matosi, kad šią klasę reiktų išskirti į dvi savarankiškas klases, nes pagal atstumus labai konkrečiai išsiskiria dvi klasės funkcionalumo kryptys.

1.4.2. Pertvarkos naudojant grafus

Kitas variantas yra panaudoti pertvarkom grafus. Šiame poskyryje analizuojama [MDJ01, MED05, Men05, MTR07] medžiaga. Programas galima vaizduoti grafais, pertvarka gali būti vaizduojama kaip perrašymo žingsnis, kuris transformuoja sudarytą programos grafą. Nurodytuose straipsniuose nagrinėjama grafų perrašymo formalizmas, kur kiekvienas grafo perrašymas gali būti

apibrėžtas pradinių ir galutinių sąlygų išraiškomis, taip pat grafų perrašymo formalizmo naudą tam tikriems objektiškai orientuotiems pertvarkymams. Tokio grafo pavyzdys pateiktas 4 paveikslėlyje.



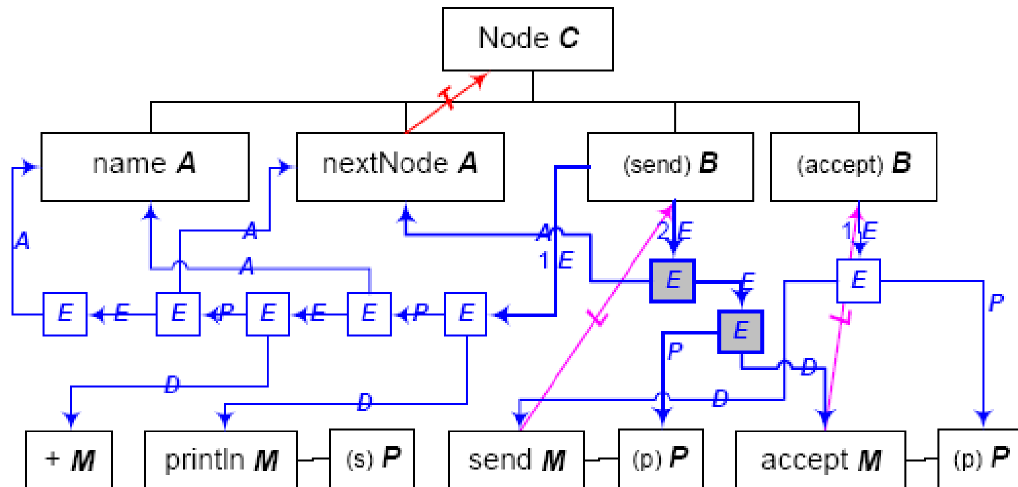
4 paveikslėlis. Programos grafas

Visi galimi viršūnių tipai pateikti 2 lentelėje.

2 lentelė. Grafo viršūnių tipai

Viršūnės tipas	Aprašymas	Pavyzdys
C	Klasė	`Node, Workstation, PrintServer, Packet`
B	Metodo kūnas	`System.out.println(p.contents)`
A	Atributas (kintamasis, ar laukas)	`Name, nextNode, contents, originator, t.t.`
M	Metodo aprašymas	`Accept, send, print`
P	Parametrai	`P`
E	Išraiškos metodo kūne	`p.contents`

Taip pat yra ir visų briaunų aprašymai. Pavyzdžiui: I:C→C reiškia paveldėjimą, ir jungia subklases su superklase. Dar vienas pavyzdys:



5 paveikslėlis. Pertvarkomas grafas

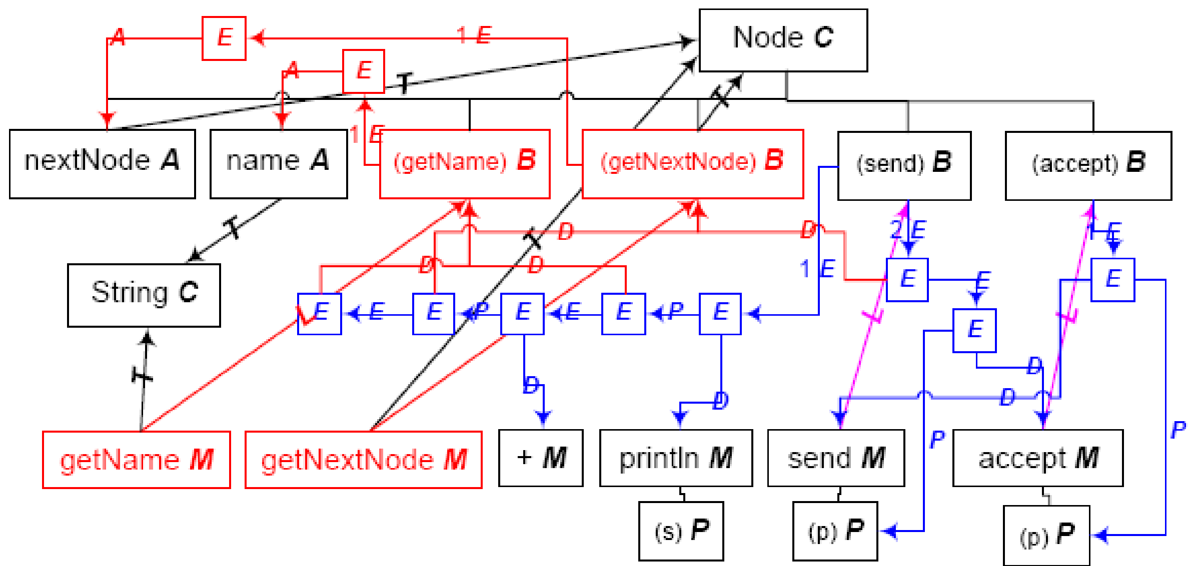
Grafo perrašymas yra ne daugiau kaip transformacija, kuri paima pradinį grafą ir jį transformuoja į rezultatus atitinkantį grafą. Ši transformacija įvyksta pagal kai kurias iš anksto nulemtas taisykles, kurios yra apibrėžtos vadinamojoje grafo gamyboje.

Dabar reikia išanalizuoti tipines pertvarkas, kurias galima įžvelgti duotame LAN pavyzdyje, ir bus parodyta, kaip reikia atlikti pertvarką naudojant grafų perrašymo formalizmą. Pirmiausia pateikiama motyvacija, JAVA programos pavyzdys, tada atitinkamas grafas, grafo perrašymas, ir funkcionalumo išsaugojimas, bei jo patikrinimas. Šiame grafe galima pritaikyti metodų apgaubimo pertvarką. Pertvarka atrodytų taip:

```
    //  ***  PRIEŠ  pertvarkant
laukus juos apgaubiant ***
    public class Node {
    public String name;
    public Node nextNode;
    public  void  accept(Packet  p)
{
    this.send(p); }
    protected  void  send(Packet  p)
{
    System.out.println(
    name +
    "sends to" +
    nextNode.name);
    nextNode.accept(p); }
    }
```

```
    //  ***  PO  laukų  apgaubimo
pertvarkos ***
    public class Node {
    private String name;
    private Node nextNode;
    public String getName() {
    return this.name; }
    public void setName(String s) {
    this.name = s; }
    public Node getNextNode() {
    return this.nextNode; }
    public  void  setNextNode(Node  n)
{
    this.nextNode = n; }
    public void accept(Packet p) {
    this.send(p); }
    protected void send(Packet p) {
    System.out.println(
    this.getName() +
    "sends to" +
    this.getNextNode().getName());
    this.getNextNode().accept(p); }
    }
```

Grafas po pertvarkos pateiktas 6 paveikslėlyje.



6 paveikslėlis. Pertvarkytas grafas

Pertvarka formaliai galėtų būti išreikšta kaip du įvykiai gaminant naują grafą:

```
EncapsulateField(Node,name,String,getName,setName);
EncapsulateField(Node,nextNode,Node,getNextNode,setNextNode)
```

Nesunkiai galima patikrinti, kad programos funkcionalumas nebuvo pažeistas, ir programa veiks taip pat. Analogiškai galima atlikti ir kitas pertvarkas, kaip antai metodų iškėlimas, arba duomenų pakeitimas objektu ir kiti. Visgi šie pertvarkymai neįvyksta automatiškai. Tačiau grafiškas programos atvaizdavimas gerokai palengvina nekokybiško kodo aptikimą, taip pat garantuoja efektyvų programos pertvarkymą, garantuojant, kad nebus pažeistas programos funkcionalumas.

Šis straipsnis pateikė vieną iš būdų formaliai įvykdyti pertvarką naudojant grafus. Tai yra tik pradiniai šios problemos tyrinėjimo rezultatai, dar yra daug tolimesnio darbo vystant šią sritį. Šis būdas yra labai perspektyvus, bet kadangi jis neaptinka pats automatiškai pertvarkų, jis nėra tinkamas projektuojamam įrankiui. Tačiau šis metodas geras tuo, kad yra nepriklausomas nuo konkrečios programavimo kalbos, grafas gali atvaizduoti bet kurių objektinių programavimo kalbų programas.

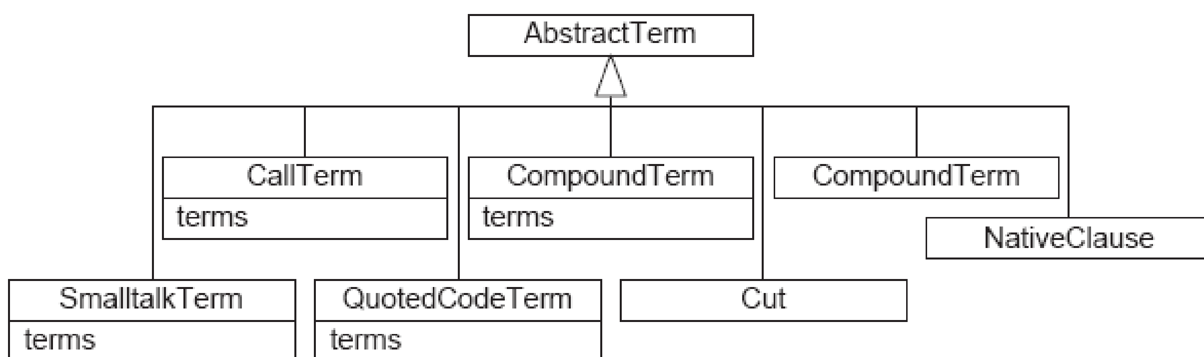
1.4.3. Loginis programavimas pertvarkoms aptikti

Šiame skyrelyje analizuojama [TBM02, TM03, WD01] medžiaga. Šiuose darbuose analizuojama loginio programavimo taikymo galimybės nekokybiškų išeities teksto vietų aptikimui, taip pat nurodomos geriausios praktikos. Tai ypač naudinga, kai norima aptikti projekte būtinas pertvarkų vietas, ir kokios būtent pertvarkos turi būti pritaikytos. Loginis programavimas yra

deklaratyvi aplinka, sujungta su meta programavimu, todėl tai yra tinkama tam, kad aptiktų tokius išeities teksto pažeidimus ir suteiktų informaciją apie galimą pertvarkymą.

Šioje analizėje, bus parodyta, kaip naudoti loginį programavimą pertvarkoms aptikti. Loginė kalba leidžia tiksliai išreikšti projekto nuostatus paprastu ir intuityviu būdu. Be to, aiškus loginis programavimas įgalina tikrinti programos tekstą, kad jis iš tikrųjų nepažeidžia nieko iš tų projektinių nurodymų (tarkime tam tikro šablono prisilaikymas). Bus pateiktas ir analizuojamas pavyzdys, kaip aptikti galimai nekokybišką programos išeities tekstą. Šiam tikslui pasiekti bus naudojama SOUL loginė programavimo aplinka. Ji yra integruota Smalltalk aplinkoje. Iš SOUL galima iškviest Smalltalk klases ir atvirkščiai, iš Smalltalk galima iškviesti SOUL taisykles. Loginės kalbos natūraliai leidžia išreikšti taisykles, pagal tam tikrą apibrėžimą. Be to, glaudi integracija leidžia SOUL logiškai naudoti ir valdyti programas, parašytas Smalltalk, ir tai padaro SOUL tinkama meta programavimo tikslams. Be to, tai taip pat reiškia, kad SOUL aplinka yra visada sinchronizuojama su vystymo aplinka (Smalltalk). Kitas pranašumas naudoti SOUL yra deklaratyvi logiška paradigma. Buvo jau parodyta, kad loginės programavimo kalbos ypač gerai tinka meta programavimui, todėl, kad jos leidžia meta programoms būti apibrėžtoms intuityviai. SOUL programavimo kalba yra iš tikrųjų Prologo kalbos šaka su tam tikru patobulinimu, kad leistų Smalltalk reiškiniams būti įvertintais kaip logiškomis programos dalimis, perrašomomis SOUL. Šitas patobulinimas leidžia materializuoti visą informaciją Smalltalk programų kaip loginius faktus. SOUL turi didelę biblioteką loginių programų, kurios nagrinėja Smalltalk programų informaciją, kad aptiktų aukšto lygio informaciją, tokią kaip projekto struktūros egzistavimas.

Gerai sukurti interfeisai yra svarbūs, projektuojant lanksčias ir daugkartinio naudojimo objektiškai orientuotas sistemas. Bet kokios situacijos, kurioje klasės interfeisas yra neatitinkantis, neužbaigtas ar neaiškus turi būti išvengiama. Kaip konkretus pavyzdys bus naudojamasi AbstractTerm hierarchija, kuri pavaizduota 7 paveikslėlyje.



7 paveikslėlis. AbstractTerm hierarchija

Ši hierarchija yra SOUL aplinkos dalis. Kaip galima pastebėti, CallTerm, CompoundTerm, SmalltalkTerm ir QuotedCodeTerm klasės kiekviena realizuoja terms metodą, tačiau visos kitos klasės to nedaro (įskaitant ir AbstractTerm). Ši situacija sukelia problemą, kai mes norime išplėsti AbstractTerm hierarchiją įterpiant naują klasę. Čia nėra tiesiogiai aišku iš projekto, kuri AbstractTerm subklasė turi realizuoti terms metodą, ir kuri neturi. Programuotojas turintis tokią klasių diagramą, turi vienareikšmiškai suprasti, kas ką daro.

Šio projekto pataisymui yra galimi du skirtingi sprendimai. Kiekviena tarpinė superklasė yra įterpiama tarp originalios superklasės ir visų subklasių, kurios turi interfeisą. Ši naujai sukurta superklasė turi paveldėti subklasėms reikalingą interfeisą. Kitas pasirinkimas yra išplėsti interfeisą originalios superklasės su interfeisu, pasidalintu subklasių. Tai taip pat implementuoja interfeisą subklasėms, kurios iš pradžių neturėjo jo, tačiau, kuris negali būti pageidaujamas.

Naudojant SOUL galima nesunkiai aptikti tokias situacijas, tiesiog reikia įdiegti tokią loginę taisyklę:

```
implementingSubclasses(?superclass,?selector,?subclasses) if
subclassImplements(?superclass,?selector, ?),
not(classImplements(?superclass,?selector)),
findall(?subclass,
subclassImplements(?superclass,?selector,?subclass),
?subclasses).
```

ImplementingSubclasses predikatas apskaičiuoja visas duotos superklasės subklases, kurios įgyvendina tam tikrą selektorių, kuris nėra savarankiškai įgyvendintas superklasėje. Taisyklė yra įgyvendinta išreiškiant dviem pagalbiniais predikatais: classImplements ir subclassImplements. Vėliau ir šie predikatai bus realizuoti šitaip:

```
subclassImplements(?superclass,?selector,?subclass) if
subclass(?subclass,?superclass),
classImplements(?subclass,?selector)
```

ClassImplements ir subclass yra SOUL taisyklių bibliotekos dalis, ir jų apibrėžti nereikia. Subclass predikatas tikrina, ar egzistuoja tiesioginis paveldėjimo ryšys tarp duotų dviejų klasių. Kai tuo tarpu classImplements tikrina, ar klasėje yra realizuotas atitinkamas selektorius. Belieka patikrinti, ar subklasių komplektas, kuris yra apskaičiuotas implementingSubclasses predikato neturi savyje visų duotos klasės subklasių. Tai tiesiog reikia palyginti aibes:

```
inappropriateInterface(?superclass,?selector,?subclasses) if
implementingSubclasses(?superclass,?selector,?subclasses),
```

```
not(allSubclasses(?superclass,?subclasses))
```

Dabar galima tiesiog panaudoti SOUL tam, kad aptiktume netinkamą interfeisą, ar jo realizaciją. Tam tereikia paleisti užklausą, kuri grąžins pažeidimą, kuris buvo anksčiau aptiktas:

```
if inappropriateInterface([AbstractTerm],?selector, ?subclasses)
```

Ši užklausa grąžins visas subklases ir selektorius, kurie netenkina sąlygos. Toks būdas padeda aptikti projektavimo trūkumus, kuriuos rankiniu būdu sunku aptikti. Tačiau tikima, kad vėliau bus galima aptikti ir išeities teksto programavimo stiliaus klaidas.

Taip pat toliau analizuojama [MT01] medžiaga apie projektavimo šablonus. Pertvarkos yra svarbios ne tik išeities tekstų lygyje, bet ir aukštesniame – projektavimo šablonų lygyje, tai ypač svarbu norint vystyti dideles sistemas, kurios nuolat keičiasi, ir norima palaikyti tam tikrą šabloną.

Loginis programavimas suteikia galimybę nepriklausomai nuo programavimo kalbos kodo specifikuoti projektavimo šabloną, apibrėžti apribojimus, praplėtimus ir kitas šablono transformacijas. Taip pat sukurti galimybę automatiškai generuoti kodą iš projektavimo šablono specifikacijos, kuris atitiks automatizuoto šablono apribojimus. Šioje dalyje taip pat bus nagrinėjama SOUL loginės programavimo kalbos ir Smalltalk kalbos sintezė. SOUL predikatų pavyzdys:

```
% loginių faktų pavyzdžiai:
subclass(Widget,Button).
subclass(Button,MacButton).
% loginių taisyklių pavyzdžiai:
hierarchy(?P, ?C) :- subclass(?P, ?C).
hierarchy(?P, ?C) :- subclass(?P, ?D), hierarchy(?D, ?C).
```

Norint manipuliuoti objektinės kalbos kodu reikia pasiekti jos kodą per SOUL. Visos objektiškai orientuotos kalbos konstrukcijos yra aprašomos kaip faktai, keletas pavyzdžių pateikiama 3 lentelėje. Tokiu būdu SOUL pilnai padengia visas Smalltalk kalbos programas. Naudojantis šiais faktais galima gauti visą reikiamą informaciją apie Smalltalk programas. Lygiai taip pat galima ir iš Smalltalk programos iškviesti šiuos faktus.

3 lentelė. SOUL faktai

Predikatai	Paaiškinimas
`class(?C)`	C privalo būti klasė
`subclass(?P,?C)`	C klasė turi būti tiesioginė subklasė klasei P
`concreteSubclass(?P,?C)`	C klasė turi būti subklasė klasei P
`abstractMethod(?C, ?M)`	M turi būti abstraktus metodas klasėje C
`concreteMethod(?C, ?M, ?B)`	M turi būti konkretus metodas su kūnu B klasėje C
`classMethod(?C, ?M, ?B)`	M turi būti metodas su kūnu B klasėje C

instanceVariable(?C, ?V)	V turi būti kintamasis klasėje C
objectCreationBody(?M, ?B, ?C)	M metodo kūnas B turi sukurti egzempliorių C

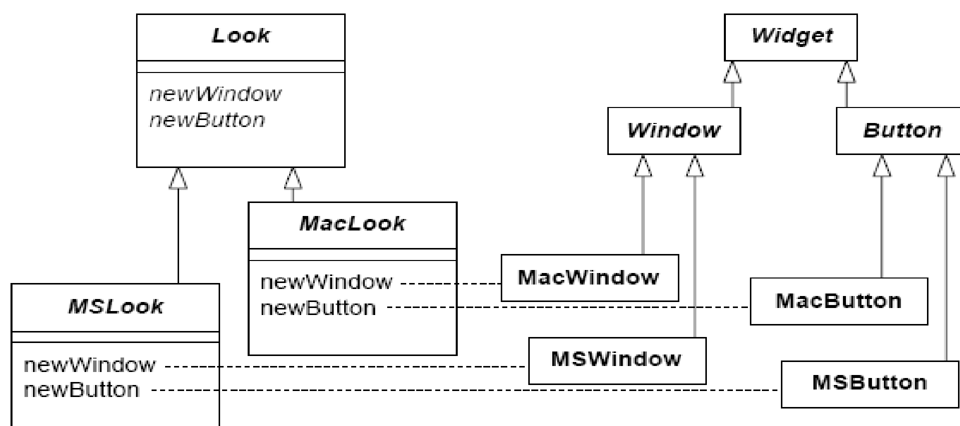
Kaip parodyta lentelėje, faktas apibrėžia Smalltalk objektus. Tokiu būdu rašant loginę taisyklę, galima nagrinėti Smalltalk programas. Apibrėžti kalbos predikatai naudojami tikrinti ir gauti informacijai. Taip pat jie gali būti naudojami generuoti kodui. Kiekvienas loginis predikatas turi savo kodo griaučius: generateCode(subclass(Widget,Button)). Taip pat per juos apibrėžiami nauji predikatai:

```
generateCode(hierarchy(?P, ?C)) :-
nonvar(?P), nonvar(?C),
generateCode(subclass(?P, ?C)).
```

Šablonas yra deklaratyviai specifikuojamas naudojant pattern predikatą, kuris gali specifikuoti šablonus: composite, visitor, abstractFactory ir t.t. Tada apibrėžiamos rolės – komponentai dalyvaujantys šablone:

```
pattern(abstractFactory,
[abstractF actory, concreteFactory, genericP roduct,
abstractP roduct, concreteProduct, abstractRelation,
concreteRelation, abstractFactoryMethod,
concreteF actoryMethod]).
```

Apsibrėžus roles, kiekvienai iš jų yra priskiriami dalyviai. Pavyzdžiuose naudojamas abstrakčios gamyklos pavyzdys pateiktas 8 paveikslėlyje.



8 paveikslėlis. Abstrakti gamykla

Rolės:

```
patternInstance(AF1, abstractFactory).
role(AF1, abstractFactory, Look).
role(AF1, concreteFactory,MSLook).
role(AF1, concreteFactory,MacLook).
role(AF1, genericProduct,Widget).
role(AF1, abstractProduct,Window).
role(AF1, abstractProduct,Button).
```

```
role(AF1, concreteProduct, [MSWindow,Window]).
role(AF1, concreteProduct, [MSButton,Button]).
role(AF1, concreteProduct, [MacWindow,Window]).
role(AF1, concreteProduct, [MacButton,Button]).
role(AF1, abstractRelation, [Look,Window]).
role(AF1, abstractRelation, [Look,Button]).
role(AF1, concreteRelation, [MSLook,MSWindow]).
role(AF1, concreteRelation, [MacLook,MacWindow]).
role(AF1, concreteRelation, [MSLook,MSButton]).
role(AF1, concreteRelation, [MacLook,MacButton]).
role(AF1, abstractFactoryMethod, [newWindow, Look,Window]).
role(AF1, abstractFactoryMethod, [newButton, Look,Button]).
role(AF1, concreteFactoryMethod, [newWindow,MSLook]).
role(AF1, concreteFactoryMethod, [newWindow,MacLook]).
role(AF1, concreteFactoryMethod, [newButton,MSLook]).
role(AF1, concreteFactoryMethod, [newButton,MacLook]).
```

Naudojant tokį principą, yra užtikrinamas šablonų pritaikymo tęstinumas. Specifikuojant šablonus iš anksto, ateityje yra stipriai sumažinama laiko sąnaudos naujų objektų įterpimui ir apskritai modelio modifikavimui, nes galima tiesiog aprašyti logines taisykles, kaip turi kas pasikeisti, ir automatiškai atitinkamai bus pakeistas programos kodas. Kol kas toks funkcionalumas yra naudojamas tik moksliniais tikslais, yra bandoma sukurti panašų įrankį JAVA programavimo kalbai, bei C.

Naudojant SOUL yra generuojamos visos klasės, interfeisai, metodai, ir jie iškart sukuriami reikiamoje vietoje taip, kad nebūtų pažeisti šablono specifikacijos apribojimai. Tokiu būdu nuolatos yra palaikomas tvarkingas šablonas, kurio nebereikia pertvarkyti, nes jis nuolatos kuriamas pagal specifikaciją.

Iš visų nagrinėtų būdų šitas yra universaliausias, nes jis nesukurtas konkrečioms pertvarkoms, bet sukurtas taip, kad būtų galima realizuoti kokias tik norima pertvarkas. Tačiau konkrečiai šio atvejo trūkumas, kad ši SOUL loginė kalba yra integruota su konkrečia objektine kalba, ir negali būti pritaikomas kitoms kalboms. O šio darbo tikslas sukurti nepriklausomą nuo pertvarkomos programos kalbos įrankį. Kitame skyriuje nagrinėjamas šio įrankio modelis.

2. Pertvarkų kalba

Išnagrinėjus keletą pertvarkų radimo būtų, galima išskirti svarbiausius dalykus, kuriuos galima panaudoti, kuriant universalų – nepriklausomą įrankį:

a) Pertvarkų radimas naudojant matus – iš šio būdo galima paimti matų naudojimą, nes norint automatiškai aptikti pertvarkas, turi jos būti formalizuotos ir kartais reikia apibrėžti kriterijus, kuriais remiantis būtų jos ieškomos, daugeliu atveju tokie kriterijai gali būti tam tikri matai.

b) Pertvarkos naudojant grafus – šis būdas geras tuo, kad yra nepriklausomas nuo programavimo kalbos. Bet kurios kalbos programą galima pervesti į grafą, o tada jau galima nagrinėti šį grafą, kuriame paprasčiau aptikti pertvarkas, ir net galima aprašyti grafo savybes, kurios reikštų tam tikros pertvarkos galimą pritaikymo vietą.

c) Loginis programavimas pertvarkoms aptikti – suteikia galimybę formaliai aprašyti pertvarkas, bei automatiškai jas aptikti programose, tačiau šis išnagrinėtas konkretus pavyzdys netinkamas, nes yra susietas su viena programavimo kalba SmallTalk.

Kadangi pertvarkų pritaikymas programoms nepriklausomai nuo jų kalbų yra šio darbo vienas iš tikslų, reikia pasiekti, kad vieną kartą parašius pertvarką, jos nereikėtų perrašyti norint pritaikyti kitoms kalboms, ir jei ta pertvarka yra tobulinama, naują jos versiją vėl gi iškart galima pritaikyti visoms objektinėms kalboms. Aišku, ne visos pertvarkos bus nepriklausomos nuo pertvarkomos kalbos, nes pertvarkos gali būti aktualios vienai kalbai, ir visiškai nepritaikoma kitai kalbai. Šiam tikslui įgyvendinti galima pasinaudoti pertvarkų naudojant grafus idėja – sukurti pertvarkų kalbą, kuri būtų nepriklausoma nuo pertvarkomos programos.

Norint realizuoti pertvarkas, kurios yra nepriklausomos nuo objektinės kalbos, reikia turėti tarpinę formą į kurią būtų suvedamos skirtingų objektinių kalbų programos, jei ši tarpinė kalba visoms kalboms bus vienoda, galima bus teigti, kad ši kalba yra universali. Tada pertvarkos būtų atliekamos jau nebe su objektine programa, bet su gautais duomenimis naudojant tarpinę kalbą, kurią galima pavadinti – pertvarkų kalba, o rezultatai vėl suvedami į pradinę programą.

Šiame skyriuje nagrinėjama ši pertvarkų kalba, kokie yra keliami jai reikalavimai, ir kaip ji gali būti realizuota, kad būtų pasiekti norimi tikslai – pertvarkų nepriklausomumas nuo kalbos, ir automatiškas jų aptikimas.

2.1. Pertvarkų duomenys

Pertvarkų duomenys gaunami transformuojant objektines programas naudojant pertvarkų kalbą. Priklausomai nuo pertvarkų kalbos pilnumo, galima pilnai pervesti objektinę programą į tarpinę kalbą, arba dalinai, tai priklauso nuo to, kaip bus apibrėžta pertvarkų kalba. Tikslas yra, kad pertvarkų duomenys turėtų pilną informaciją apie pradinę programą, ir pagal juos būtų galima atstatyti pradinę programą, vadinasi turėtų būti abipusis atvaizdavimas: A → B → A (A – pradinė programa, B – pertvarkos duomenys). Tačiau A kalba gali visada skirtis, tačiau B kalba turi likti tokia pati norint išlaikyti universalumą (nepriklausymą nuo konkrečios kalbos), todėl kyla klausimas, galbūt galima iš pertvarkų duomenų atstatyti ir į kitą objektinę kalbą, nebūtinai į tą, iš kurios buvo transformuojama A → B → A‘. Tada būtų galima šią kalbą pritaikyti ne tik pertvarkom, bet ir programų pervedimui į kitas programavimo kalbas.

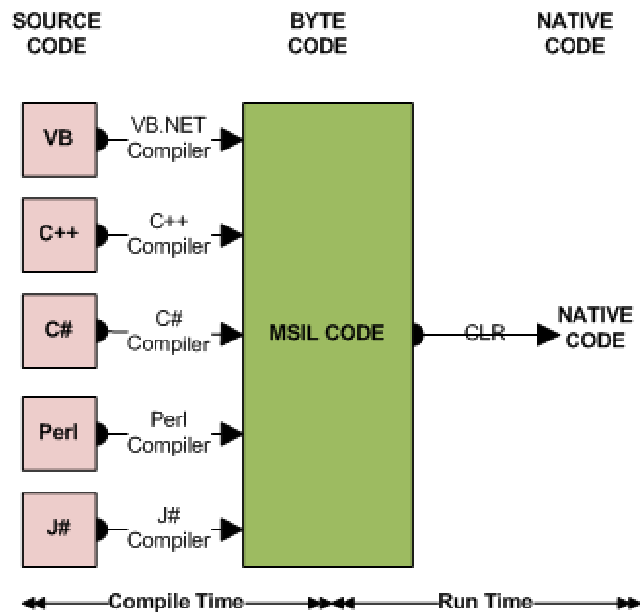
Pertvarkų duomenims keliami trys svarbiausi reikalavimai, pagal kuriuos bus atrinktas tinkamiausias būdas duomenims vaizduoti:

1) nepriklausomumas nuo konkrečios programavimo kalbos, duomenų formatas turi būti nesudėtingai atvaizduojamas ir nepriklausomas nuo pertvarkomos programos kalbos, taip pat nemokamas ir viešai prieinamas;
2) duomenų nepraradimas, turi būti galimybė atkurti pradinę programos struktūrą pagal šiuos duomenis, net ir dalinę informaciją turi būti galimybė vienareikšmiškai atstatyti programoje;
3) paprastumas, duomenys turi būti lengvai kuriami, bei nesudėtingai perskaitomi.

Toliau nagrinėjami tris galimi variantai šio tikslo įgyvendinimui, ir bus nuspręsta, kuris geriausiai įgyvendina iškeltus reikalavimus.

2.1.1. MSIL

Tokios kalbos pavyzdys galėtų būti *Microsof Intermediate Language* (*MSIL*), kuri yra tarpinė kalba tarp .Net platformos kalbų ir dvejetainio kodo. Visos .Net platformos kalbomis parašytos programos transformuojamos į šią tarpinę kalbą 1 paveikslėlis. Tai yra naudinga dėl to, kad tokiu atveju visos programos gali naudoti bendras klasių bibliotekas, nepriklausomai nuo to, kuria kalba jos parašytos. Net gi viename projekte gali būti kiekviena klasė parašyta kita programavimo kalba, svarbiausia, kad viskas transformuojama į vieną kalbos programą. Kyla klausimas, kodėl neprogramuojama būtent šia vidine kalba, o todėl, kad ji yra labai sudėtinga, ir neskirta programuoti žmogui. Ji yra sudėtinga, nes jai reikia apimti visų skirtingų kalbų ypatybes.



9 paveikslėlis, .Net platforma[1]

Net ir nesančias .Net kalbas yra įmanoma pervesti į šią kalbą, tam tiesiog reikia parašyti transformatorių. Juk galiausiai vis vieną visos kalbos pervedamos į dvejetainį kodą. Taip pat MSIL kalbos programą galima pervesti į bet kurią pradinę programą, to nėra gamintojų programinėje įrangoje, tačiau yra sukurta trečių šalių. Tai gi ši vidinė kalba kaip ir atitinka iškeltą tikslą, kad būtų nepriklausoma nuo konkrečios kalbos, ir kad būtų galima ją pervesti vėl į pradinę programą, net gi į kita kalba parašytą programą nei buvo pradinė. Tada tereiktų parašyti atskiras MSIL programas, kurios atliktų pertvarkas su transformuotomis programomis į MSIL, ir pertvarkytą programą vėl atkurtų į pasirinktos kalbos programą.

Atrodytų, kad kaip ir yra sprendimas universalioms pertvarkoms atlikti, tačiau iš tiesų MSIL visiškai netinkamas šiame darbe keliamam tikslui pasiekti. Visus teorinius aspektus ši kalba tenkina, tačiau didžiausias jos trūkumas yra sudėtingumas ir priklausomumas nuo vienos kalbų grupės (Microsoft .Net). MSIL – tai yra pakankamai žemo lygio programavimo kalba, todėl programuotojai tiesiogiai su ja neprogramuoja, todėl su ja kurti pertvarkas būtų sudėtinga, ir mažai kas galėtų jas naudotis. Taip pat iškiltų problema, ar tam tikros pertvarkų vietos atsirado ne dėl to, kad buvo atlikta transformacija, galbūt pradinės kalbos programoje tos problemos nebuvo, o MSIL programoje atsirado, lygiai taip pat negarantuojama, kad atlikus pertvarką MSIL programoje, ją pervedus vėl į aukšto lygio programą, ta pertvarka bus įvykdyta korektiškai ir nesukels naujų kodo problemų. Didžiausia problema yra tame, kad jei pradinėje programoje buvo nekokybiško kodo, jo vidinėje

[1] http://grounding.co.za/blogs/trevor/archive/2008/03.aspx

programoje gali nelikti, nes transformuojant į MSIL kalbą vyksta ir programos optimizavimas. Todėl ne vien dėl programų rašymo sudėtingumo, bet ir dėl pradinės programos kodo sintaksės praradimo ši kalba nėra tinkama sprendžiamai problemai, nes pertvarkų tikslas yra ne pakeisti funkcionalumą, bet padaryti paprastesnį ir aiškesnį kodą, programos tekstą, tvarkingesnę programos struktūrą, ko MSIL kalba padaryti nebūtų galima, nes ji perima tik funkcionalumą, bet ne sintaksę. Tai gi šis duomenų vaizdavimo būdas netenkina visų trijų reikalavimų.

2.1.2. XML panaudojimas

Kadangi programą reikia pervesti į tarpinę formą, reikia rasti tam tinkamą įrankį, tam galėtų tikti ir XML, kuris kaip ir yra skirtas saugoti ir keistis duomenimis, ir yra nepriklausomas nuo kitų programavimo kalbų. Sukūrus programų transformatorius, kurie perveda programą į XML naudojantis apibrėžtus elementus, galima pasiekti nepriklausomumo, ir nebūtų prarasta programos sintaksinė informacija, pavyzdys 4 lentelėje.

4 lentelė. Programos perrašymas XML

Objektinė kalba	XML

class Stack {
int pop = 0;
…
public Object peek ( ) {
return contents[pop]; }
public Object pop () {
return contents[--pop]; {
… }
	<class name="Stack">
	<variables>
	<var type="int" name="pop" value="0"/>
	</variables>
	<methods>
	<method name="peek" return="Object" returnvalue="contents[pop]">
	<parameters></parameters>
	<body></body>
	</method>
	<method name="pop" return="Object" returnvalue="contents[--pop]">
	<parameters></parameters>
	<body></body>
	</method>
	</methods>
	</class>

Tada reikėtų parašyti pertvarką, kuri atitinkamai analizuotų XML duomenis ir pateiktų pertvarkos rezultatus. Pertvarkų programas, kurios analizuotų XML duomenis galima būtų parašyti su bet kuria programavimo kalba, kuri gali ir nesutapti su pertvarkoma programa, nes ji analizuoja XML, o ne pradinės programos tekstą. Tarkime pradinė programos kalba yra C#, šia kalba parašyta programa transformuojama į XML duomenų failą, tada, tarkime, JAVA kalba parašyta pertvarkos programa gali sėkmingai analizuoti XML duomenis, juose aptikti reikiamą pertvarkos vietą. Rezultatas galėtų taip pat būti XML failas, arba tiesiogiai pertvarkymai pritaikomi pradinei programai, ir gaunama nauja pertvarkyta programa. Šis variantas patenkina abu reikalavimus –

nepriklausomumą nuo kalbos, ir informacijos nepraradimą, bei neiškraipymą, šio reikalavimo netenkino MSIL kalba, tačiau šis variantas netenkina paprastumo reikalavimo. Programą pervesti į XML failą nėra sudėtinga, tačiau parašyti pertvarką, kuri analizuoja XML duomenis yra išties sudėtinga, nes pradinės programos sintaksė yra vientisa ir visa susijusi tarpusavyje, o XML failo duomenys būtų padriki, programoje papildomai reikėtų programuoti loginius ryšius tarp duomenų, tik turint juos galima būtų kurti pertvarką. Tarkime, norima patikrinti, ar klasių paketuose nėra ciklo, tokiu būdu reikėtų rasti visus klasių tarpusavio ryšius, tą atlikt analizuojant XML būtų gana nepatogu, tai reikalautų daug griozdiško kodo, tačiau tai būtų įmanoma.

Galima daryti išvadą, kad šis variantas yra tinkamas keliamiems tikslams įgyvendinti, tačiau galbūt yra dar geresnis variantas, kuris tenkina visus tris reikalavimus: nepriklausomumą nuo kalbos, duomenų nepraradimą, bei paprastumą. Kitame skyrelyje nagrinėjamas būdas, kuris tenkina visus tris reikalavimus.

2.1.3. Loginė kalba

Labai svarbu transformuojant programą ne tik perduoti duomenis apie programos objektinę struktūrą, bet ir ryšius toje struktūroje, duomenys turėtų sudaryti didelį jungų grafą, kuriame būtų aiškūs ryšiai. Daug svarbiau turėti informaciją apie tai kas yra daroma programoje, o ne kaip yra daroma. Vadinsi reikalinga deklaratyvi informacija apie programą. Deklaratyviai informacijai kaupti ir nagrinėti tinka loginės kalbos, nes jos ne tik aprašo informaciją, bet ir aprašo ryšius tarp jų. Loginės programos turi faktus, kuriais aprašomi duomenys, bei taisykles, kurios aprašo ryšius tarp faktų, būtent to ir reikia, norint perteikti visą informaciją apie programos struktūrą. Faktų panaudojimo pavyzdys 5 lentelėje.

5 lentelė. Programos perrašymas faktais

Objektinė kalba	Loginė kalba

`class Stack {`
`     int pos = 0;`
`     …`
`  public Object peek ( ) {`
`     return contents[pos]; }`
`  public Object pop () {`
`     return contents[--pop]; {`
`… }`
	`Class(„Stack").`
	`Var(„Stack", „int", „pos");`
	`...`
	`Method(„Stack", „Object", „peek", [], {return contents[pos]}).`
	`Method(„Stack", „Object", „pop", [], {return contents[--pop]}).`
	`...`

Šis pavyzdys parodo, kad yra perkeliama visa informacija apie programos struktūrą: klasės, metodai, kintamieji, bei ryšiai: metodas, ar kintamasis priklauso tokiai klasei. Taip pat šiais faktais būtų galima perrašyti bet kurios kitos objektinės kalbos programą, nes faktai nepririšti prie konkrečių raktinių kalbos žodžių, bet susieti su objektinių kalbų savybėmis. Vadinasi tokiais pačiais

loginiais faktais galima aprašyti bet kurią objektinę kalbą, tai patenkina pirmąją sąlygą – nepriklausomumą nuo objektinės kalbos. Taip pat yra perteikiama visa informacija apie programą, kitaip sakant, turint loginės programos duomenis yra įmanoma atstatyti pradinę programą, šis reikalavimas taip pat patenkinamas, nes yra perkeliami visi duomenys.

Yra kuriama SmallTalk kalbos loginė faktų aibė SOUL loginėje kalboje. Tačiau SOUL kalba yra priklausoma nuo SmallTalk, todėl tiesiogiai panaudoti šios faktų aibės kaip pertvarkų duomenų negalima, nes netenkina nepriklausomumo reikalavimo, tačiau pasinaudoti šioje kalboje apibrėžtais faktais yra galima, nes SmallTalk irgi yra objektinė programavimo kalba. Nagrinėjant SOUL ir SmallTalk integraciją, galima teigti, kad šiame konkrečiame atvejyje yra pritaikyti šiame darbe keliami tikslai, tik jie yra daug siauresni, nes SOUL yra SmallTalk dalis. Tačiau šis pavyzdys įrodo, kad loginė kalba objektinių programų analizavimui yra tinkama. Vadinasi reikia surasti tinkamą loginę programą, kuri yra nepriklausoma nuo jokios kitos objektinės programos, tai gali būti ir Prolog loginė programavimo kalba, kuri yra populiariausia ir plačiausiai žinoma tarp programuotojų, tačiau galima pritaikyti ir kitas logines kalbas.

Išnagrinėjus šiuos tris skirtingus atvejus galima teigti, kad MSIL netinkamas, nes jį galima naudoti tik Microsoft .Net kalboms, taip pat yra retai naudojamas praktiškai, per sudėtingas dėl savo žemo programavimo lygio. Kiti būdai duomenims atvaizduoti yra tinkami tiek XML, tiek loginiai faktai. Todėl norint nuspręsti, kuris iš šių dviejų būdų yra geresnis, reikia išnagrinėti, kurio atvejo duomenimis patogiau naudotis rašant pertvarkų programas.

2.2. Pertvarkų programos

Ši pertvarkų kalbos dalis yra svarbiausia, nes čia sukuriamos pertvarkos, kurios turi būti pritaikytos objektinėms programoms. Kadangi ši kalba yra nepriklausoma nuo pertvarkomos programos, tai natūralu, kad ši pertvarkų programa naudos pertvarkų duomenis, o ne tiesiogiai objektinę programą. Norint sėkmingai sukurti pertvarkų programas reikia turėti pakankamai informacijos apie pertvarkomą programą, o ši informacija yra laikoma pertvarkų duomenyse. Todėl priklausomai nuo to, koks duomenų tipas bus pasirinktas, priklauso ir pertvarkų programų rašymas.

Kadangi MSIL tipo kalbos buvo atmestos, kaip per daug sudėtingos šio darbo tikslams įgyvendinti, galimi variantai yra XML + objektinė kalba, arba loginė kalba. XML kauptus duomenis gali nagrinėti bet kurios populiariausios objektinės programos. Loginės kalbos atveju duomenys būtų kaupiami pačios kalbos teikiamais būdais. Nors loginė kalba pristatyta kaip optimaliausias variantas, tačiau negalima atmesti ir XML panaudojimo.

Geriausia išnagrinėti trūkumus ir privalumus nagrinėjant pavyzdžius. Pertvarka būtų labai paprasta: abstrakti klasė gali būti tik tokia, kurioje abstrakčių metodų yra daugiau nei pusė, jei yra priešingai, abstrakčius metodus reikia iškelti į naują abstrakčią klasę. Šiai pertvarkai daug duomenų nereikia, pertvarkų duomenys pateikti 6 lentelėje.

6 lentelė. Pertvarkos duomenys

Loginiai pertvarkų duomenys	XML pertvarkų duomenys
class(a). class(b). class(c). method(a, m1, true). method(a, m2, false). method(a, m3, true). method(a, m4, false). method(a, m5, false). method(b, m6, false). method(c, m7, true).	<classes> <class name="a"> <methods> <method name="m1" isAbstract="true"/> <method name="m2" isAbstract="false"/> <method name="m3" isAbstract="true"/> <method name="m4" isAbstract="false"/> <method name="m5" isAbstract="false"/> </methods> </class> <class name="b"> <methods> <method name="m6" isAbstract="false"/> </methods> </class> <class name="c"> <methods> <method name="m7" isAbstract="true"/> </methods> </class> </classes>

Pertvarkos programą parašyti logine kalba yra gana paprasta, tiesiog reikia sintaksiškai teisingai užrašyti užduotį, kurią galima performuluoti taip: klasė turi būti pertvarkoma, jei paimant po vieną klasę, suskaičiavus abstrakčių metodų kiekį klasėje, tada suskaičiavus visų metodų kiekį, gaunamas jų santykis, kuris yra mažesnis už pusę, ir nelygus 0 (kas reikštų, kad abstrakčių metodų nėra). Taip performulavus užduotį, ją galima įgyvendinti pažodžiui, ir gaunama tokia pertvarkos programa:

Prolog pertvarkos programa:

```
haveAbstract(C,N) :- class(C),
    findall(C,method(C,_,true),List1),
    countList(List1,A), findall(C,method(C,_,_),List2),
    countList(List2,B), N is A / B, N < 0.5, N =\=0.
```

Pertvarkos rezultatas:

```
?- haveAbstract (Class,Ratio).
    Class = a,
    Ratio = 0.4 ;
    false.
```

Šioje programoje panaudojamas viena sisteminė Prolog taisyklė: *findall*, kuri grąžina visų faktų aibę, kurie tenkina taisyklę, bei aibės ilgio skaičiavimo taisyklę, kuri realizuota taip:

```
countList([],0).
countList([_|Tail],N) :- countList(Tail,N1), N is N1+1.
```

Šiam uždaviniui irgi buvo galima panaudoti sisteminę taisyklę *length(List, Count)*. Lygiai taip pat ir sisteminę funkciją *findall* galima buvo realizuoti per naujo, kitaip sakant, galima puikiai atlikt pertvarką ir neprisirišus prie konkrečios loginės kalbos sisteminių taisyklių. Atsakymą galima suprast, kad klasė „a“ turi mažiau nei pusė abstrakčių metodų (0.4), todėl ją reiktų skaidyti. Tai gi su logine kalba ši pertvarka įgyvendinta gana paprastai.

XML duomenis analizuojančią pertvarkos programą galima rašyti su bet kuria kalba, šis pavyzdys pateikiamas su C# kalba. Programuojant ne logine kalba, nepavyks taip paprastai perkelti užduoties į programą, kaip pateikta Prolog pavyzdžiu, taip pat neišvengiamai reikės papildomai skirti daug dėmesio programos aprašams sukurti, kurie yra būtini norint paleisti programą, susikurt paketą, apsirašyti klasę.

C# pertvarkos programa:

```
using System;
using System.Xml;

namespace XMLRefact
{
    class Program
    {
        static void Main(string[] args)
        {
            XmlTextReader textReader = new XmlTextReader("data.xml");
            XmlDocument doc = new XmlDocument();
            XmlNodeList methods, classes;
            string classname;
            bool isAbstract;
            int abstr = 0, all = 0;
            float ratio;

            doc.Load("data.xml");
            classes = doc.SelectNodes("/classes/class");

            foreach (XmlNode instance in classes)
            {
                classname = instance.Attributes["name"].Value;
                methods = instance.SelectNodes("methods/method");

                foreach (XmlNode module in methods)
                {
                    isAbstract =
Convert.ToBoolean(module.Attributes["isAbstract"].Value);
                    if (isAbstract) abstr++;
```

```
                    all++;
                }

                ratio = (float)abstr / all;
                if ((abstr != 0) && (ratio < 0.5))
                    Console.WriteLine("class: " + classname + ";
abstract methods ratio: " + ratio);
                abstr = all = 0;
            }
        }
    }
}
```

Pertvarkos rezultatas:

```
class: a; abstract methods ratio: 0,4
```

Rezultatas abiejų programų yra toks pats, programų sudėtingumas yra skirtingas. Pertvarkom atlikti reikia naudoti deklaratyvius duomenis, o juos apdoroti geriausia su logine programa, nes loginės kalbos ir priklauso deklaratyviom kalbom. Kuriant pertvarkos programa su logine programa, gana aišku kaip užrašyti pertvarką, tiesiog reikia aiškiai suformuluoti užduotį, tada išnagrinėti joje esančius ryšius tarp duomenų ir juos aprašyti logine seka. Pertvarką rašant struktūrine ar objektine programavimo kalba daug darbo reikia skirti, kaip išgauti reikiamus duomenis ir juos analizuoti. Jei duomenys būtų aprašyti ne XML, o paprastu tekstiniu formatu pagal tam tikrą sugalvotą struktūrą, duomenų analizė taptų dar sudėtingesnė, nes reiktų analizuoti tekstinę struktūrą, kaip tuo tarpu loginiai duomenys patys savaime yra programos dalis. Logine kalba nesudėtingai aprašyti įvairius ryšius programose, ir tikrinti, ar tų ryšių yra laikomasi.

Jei loginis programavimas išties yra tinkamas programoms transformuoti, kyla klausimas, kaip turėtų būti aprašomi duomenys. Pats paprasčiausias variantas yra kiekvienai pertvarkai parašyti atskirą pertvarkos kalbą. Tada reiktų paimt iš programos tik tuos duomenis, kurių reikia atlikt pertvarkai. Pavyzdžiui, iškelti abstrakčius metodus iš klasių, kuriose jų yra mažiau nei pusė. Tokiai pertvarkai įgyvendinti užtenka tik informacijos apie klasių pavadinimus ir metodų modifikatorių, ar jos yra abstrakčios. Tokią klasę sudarytų tik du faktai. Tačiau, jei norima realizuoti daug pertvarkų, ir kiekvienai pertvarkai kuriamos naujos kalbos, gali susidaryti daug skirtingų faktų, kurie laiko tą pačią informaciją, vien dėl to, kad bus skirtingai pavadinti. Todėl reiktų kalbą nuolat plėsti, o ne kurti naujas, taip pat iškart nevertėtų sukurti kuo pilnesnės kalbos, nes gali būti sudėtinga rašyti pertvarkų programas. Kiekvienai pertvarkai naudojami faktai būtų sąjungos būdu papildomi į turimą pertvarkų kalbos faktų aibę, tokiu būdu nebebūtų dubliuojami faktai.

Galima daryti išvadą, kad jei pertvarkos duomenis patogu vaizduoti tiek XML, tiek loginiais faktais, tai pertvarkas rašyti logine kalba tikrai yra efektyviau nei objektine ar struktūrine kalba.

Beliko išnagrinėti, kaip patogiausia atvaizduoti programos rezultatus, kad būtų kiek įmanoma pilniau įgyvendinta pertvarkos užduotis. Šiame pavyzdyje atsakymas buvo tiesiog informacinis, tačiau galbūt norima daugiau nei informacijos, bet ir automatinio pertvarkos įgyvendinimo, tai irgi svarbi šios pertvarkų kalbos dalis.

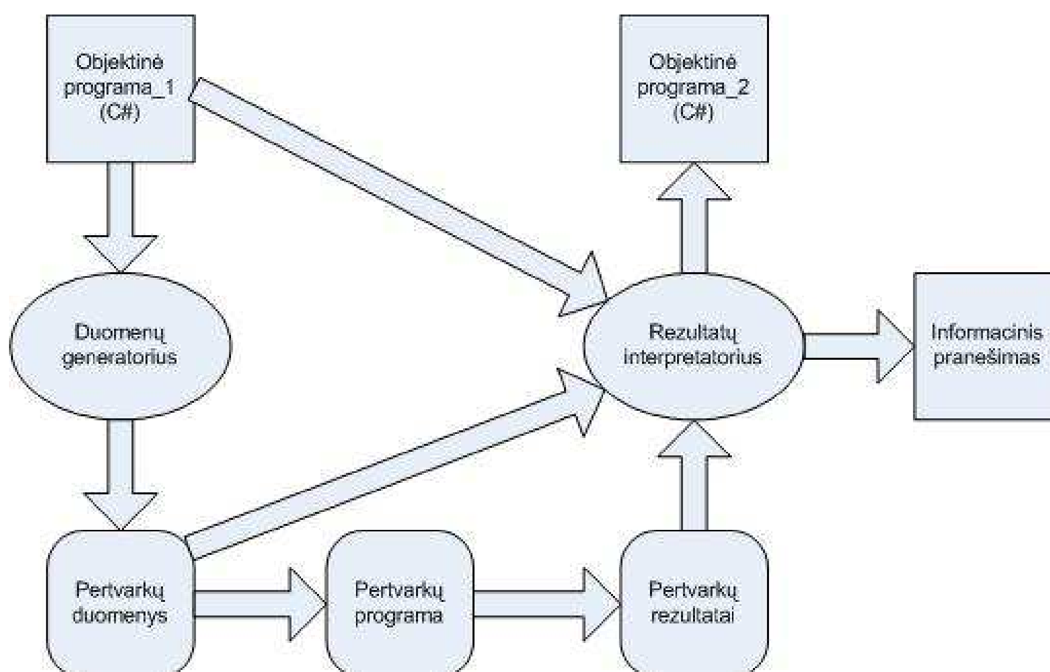
2.3. Pertvarkų rezultatai

Pertvarkų rezultatai turi būti tokie, kad tenkintų pertvarkos užduotį. Pertvarka gali būti parašyta, kad tik informuotų programuotoją apie pertvarkos taikymo vietas, arba programa ne tik aptiktų pertvarkų vietas, bet ir jas įgyvendintų perdarant pradinę programą. Visais atvejais programos rezultatas turi būti interpretuojamas vienareikšmiškai.

Pirmasis atvejis yra, kai pertvarka reikalauja tik informuoti, kur galima pritaikyti pertvarką, arba ir pasiūlo, kaip tą pertvarką atlikti. Tai paprasčiausias atvejis, kurio pavyzdys jau buvo pateiktas, tiek Prolog tiek C# programų rezultatai buvo pakankamai informatyvūs, kad galima būtų padaryti reikiamas išvadas apie pertvarkos rezultatus. Akivaizdu, kad rezultato atveju lankstesnė yra objektinė pertvarkų programa, nes ten neribotai galima pateikti duomenis. Loginėms programoms rezultatus pateikti yra sudėtingiau, bet iš dalies įmanoma, tam galima pasitelkti duomenų kaupimo būdą naudojant predikatus *assert, retract*. Šis atvejis ir būtų naudojamas dažniausiai, nes galbūt ne visas aptiktas vietas programuotojas nori pertvarkyti, galbūt pertvarką nori atlikti savaip. Esant tokiems reikalavimams loginės programos irgi yra tinkamos, nors ir nusileidžia kitu būdu kuriamoms pertvarkoms. Tačiau, kadangi svarbiausia dalis yra pertvarkos programos sukūrimas, nes nuo šios dalies priklauso visos pertvarkos sėkmė, tai loginio programavimo taikymas pertvarkoms atlikti yra pats tinkamiausias.

3. Universalus įrankis

Išnagrinėjus pertvarkų kalbą, reikia analizuoti ir jos realizavimo galimybes. Norint sėkmingai naudotis pertvarkų programa, reikia bent vienos papildomos programos, kuri pertvarkomą programą transformuotų į pertvarkų duomenis, kurių reikia, norint atlikti pertvarką. Antras įrankis galėtų būti pertvarkos rezultatų interpretatorius, kuris, jei yra reikalavimas, pertvarkytų pradinę programą pagal gautus pertvarkos rezultatus. Taip pat ši įrankis galėtų ir tiesiog programuotojui patogesniu būdu pateikti rezultatus, kurie yra tik informaciniai, nereikalaujant automatiškai atlikti pertvarkos. Šio įrankio schema pateikta 10 paveikslėlyje.



10 paveikslėlis. Universalaus įrankio schema

Šį įrankį geriausia sukurti su ta pačia kalba, kaip ir pertvarkoma programa. Kitaip sakant, šis įrankis jau bus priklausomas nuo konkrečios programavimo kalbos, tačiau jis galės naudotis pertvarkų programomis, kurios universalios, nes jos nagrinėja pertvarkų duomenis. Vadinasi pertvarką reikės įgyvendinti tik vieną kartą, o duomenų generatorių kiekvienai kalbai atskirai. Toliau bus nagrinėjamos abi įrankio dalys atskirai.

3.1. Duomenų generatorius

Ši programa kuriama tam, kad gauti pakankamai informacijos iš pertvarkomos programos, kad būtų galima atlikti reikiamą pertvarką. Pertvarkų kalbos skyriuje pateiktam pavyzdžiui reikėtų vos dviejų duomenų: klasių pavadinimų, bei jų metodų su savybę, ar jis abstraktus. Kaip tokią gauti

informaciją, kiekvienoje kalboje gali būti skirtingų variantų. Pats blogiausias variantas būtų nagrinėti programą, kaip tekstinį dokumentą, bet tai būtų galima taikyti tik kraštutiniu atveju, kai naudojama programavimo kalba nesuteikia jokios informacijos apie savo programas. Dauguma šiuolaikinių objektiškai orientuotų programavimo kalbų turi bibliotekas, kurios analizuoja programas, teikia informaciją apie jų struktūras, Java ir C# kalbos turi Reflection bibliotekas, kurios teikia tokią informaciją. Turint tokias bibliotekas, realizuoti tokį įrankį nėra sudėtinga. Pavyzdžiui, norint gauti visų klasių pavadinimus užtenka parašyti tokį metodą, kuris duomenis išveda į konsolę:

C# metodas *class(Name)* pertvarkos duomenims gauti

```
using System.Reflection;

public void AllClass(Assembly assembly)
{
    Type[] types = assembly.GetTypes();
    foreach (Type type in types)
    {
        Console.WriteLine("class(" + type.FullName + ")");
    }
}
```

JAVA metodas *class(Name)* pertvarkos duomenims gauti

```
   import java.lang.reflect.*;

      public void AllClass(Class[] classes)
      {
            for (Class c : classes)
              System.out.println("class(" + c.getName() + ")");
      }
```

Tiesiog metodui perduodamas norimos klasės *Assembly* objektas, kuris saugo visą informaciją apie programos klases. Tada gaunami visi tipai, kurie yra klasės, ir kiekvieno pavadinimai. Su metodais yra šiek tiek sudėtingiau, bet analogiška:

C# metodas *method(ClassName, MethodName, IsAbstract)* pertvarkos duomenims gauti

```
using System.Reflection;

public void AllMethods(Assembly assembly)
{
    Type[] types = assembly.GetTypes();
    foreach (Type type in types)
    {
        foreach (MethodBase m in type.GetMethods(
            BindingFlags.Instance |
            BindingFlags.NonPublic |
            BindingFlags.Public))
        {
            Console.WriteLine("method("+
                type.FullName + ", " +
```

```
                m.Name+", "+
                m.IsAbstract.ToString()+
                ")");
        }
    }
}
```

JAVA klasė *method(ClassName, MethodName, IsAbstract)* pertvarkos duomenims gauti

```
import java.lang.reflect.*;

public class AllMethods {
   public static void main(String args[])
   {
      try {
         Class c = Class.forName(args[0]);
         Method m[] = c.getDeclaredMethods();
         for (int i = 0; i < m.length; i++)
         System.out.println("method("+
           c.getName()+ ", " +
           m[i].toString()+ ", " +
           Modifier.isAbstract(
           m[i].getModifiers()).toString());
      }
      catch (Throwable e) {
         System.err.println(e);
      }
   }
}
```

Tai gi iš šių dviejų metodų galima būtų sukurti duomenų generatorių. Analogišku būdu galima gauti ir kitokią informaciją apie programą: apie kintamuosius, parametrus, ką grąžina metodai, koks jų matomumas, ir t.t. Tačiau negalima pasiekti metodų realizacijos, bet jei norima tokią informaciją turėti, tada galima tiesiog nagrinėti programos klases kaip tekstinę bylą. Tada galima būtų transformuoti ir kiekvieną metodo sakinį į pertvarkos duomenį. Bet tokios informacijos reikia tik nedaugeliui pertvarkų, tik toms, kurios nagrinėja metodų kūnus.

Galima teigti, kad duomenų generatoriaus realizacija nėra sudėtinga užduotis populiariausioms programavimo kalboms. Net jei programavimo kalba ir neturi bibliotekų, kurios teikia informaciją apie programą, galima naudoti parastą variantą: analizuoti programų išeities tekstus, kaip tekstines bylas, tai yra kraštutinis atvejis, bet įgyvendinamas. Jie pertvarkų kalba didinama po truputį, realizuojant vis naują pertvarką, šį generatorių irgi galima lygiagrečiai plėsti, tačiau galima ir pasinaudoti esama SOUL pertvarkų duomenų aibe, ir iškart sukurti pakankamai pilną įrankį įgyvendinti daugumai pertvarkų. Tačiau, kadangi SOUL duomenys susieti su SmallTalk kalba, negalvojant apie universalumą, gali būti nelankstu ją pritaikyti, todėl praktiškiau būtų kurti šį įrankį pagal iškylantį poreikį.

3.2. Rezultatų interpretatorius

Turint duomenų generatorių, taip pat sukūrus pertvarkų programas, reikia ir atvaizduoti jų duomenis. Vien tik žiūrėti programos rezultatą gali būti pakankamai neinformatyvu, norint suprasti pertvarkos rezultatą, todėl yra geriau sukurti įrankį, kuris transformuotų programos rezultatą į programuotojui norimą formą.

Paprasčiausias atvejis, kai pertvarka tiesiog tikrina tam tikrą sąlygą, ir atsakymas gali būti „taip“ arba „ne“. Pavyzdžiui, ar klasė turi nenaudojamų kintamųjų. Tokio tipo pertvarkom rezultatų interpretatorius gali būti visai paprastas, tiesiog programa, kuri grąžina tekstinę informaciją vartotojui, jos pradiniai duomenys būtų pertvarkos programos rezultatas. Kadangi nuspręsta pertvarkų programoms naudoti loginę kalbą, tai tokios programos rezultatas būtų toks:

Tikrinama konkreti klasė	Tikrinamos visos klasės

`?- haveUnusedVar(a).`
`true.`
	`?- haveUnusedVar(Class).`
	`Class = a ;`
	`Class = c.`

Rezultato rašymas į failą:

```
write_to_file(File, Class, Name) :-
open(File, append, Stream),
write(Stream, (Name, Class)),
nl(Stream),
close(Stream).
```

Rezultatas gali būti tiesiog išvesti klasių pavadinimai, arba dar ir pertvarkos pavadinimas. Tada rezultatų interpretatoriui tiesiog reiktų nuskaityti informaciją ir pateikti vartotojui patogiu būdu.

Sudėtingesnis variantas yra, kai pertvarka reikalauja daugiau nei informatyvaus atsakymo. Tarkim pertvarka reikalauja ne tik išvesti klases, kuriose yra nenaudojamų kintamųjų, bet ir nori, kad jie būtų ištrinti. Šiuo atveju pertvarkų programos rezultatas būtų analogiškas, tiesiog klasių ir kintamųjų pavadinimai. Tačiau rezultatų interpretatorius turi būti sudėtingesnis. Jis turi ne tik išvesti rezultatą, bet ir jį pritaikyti pradinei programai. Vadinasi šiam įrankiui neužtenka tik rezultato, reikia ir pradinės programos, kitais atvejais gali reikėti ir pertvarkos duomenų, jei, pavyzdžiui, rezultatas yra ne tai, ką reikia ištrinti, bet tai, ką reikia pervadinti.

Iškyla problema, ar galima modifikuoti jau sukurtas klases. Vėl gi pats primityviausias variantas yra redaguoti išeities tekstą tiesiogiai, tačiau tai yra kraštutinis atvejis, nes yra sudėtingai realizuojamas ir sunkiai testuojamas. Tačiau geresnio varianto nėra. *Reflection* biblioteka galima

naudoti tik informacijos gavimui, tačiau nieko redaguoti negalima. Vadinasi pertvarkų įgyvendinimas galimas tik tiesiogiai keičiant klasių išeities tekstus. Todėl šį įrankį realu realizuoti tik nesudėtingoms pertvarkoms. Vadinasi daugeliu atveju šis įrankis bus naudojamas tik pertvarkos rezultatų pateikimui vartotojui. Nenaudojamų kintamųjų pertvarką galima realizuoti ir praktiškai. Tiesiog reikia atitinkamoje klasėje surast pertvarkos pateiktus kintamuosius, ir juos ištrinti.

Paprasčiausiu atveju šį įrankį galima naudoti tik pertvarkų rezultatų atvaizdavimui. Jei duomenų generatorių galima sukurti vieną kartą turint pakankamai pilną pertvarkų kalbą, kurios užtenka atlikti daugumą pertvarkų, tai rezultatų interpretatoriui reikėtų kiekvienai pertvarkai kurti atskirą programą, jei programos pakeitimai yra sudėtingi. Aišku, jei skirtingų pertvarkų rezultatas yra toks pats, pavyzdžiui, iškelti metodus į naują klasę, tai nebūtina kurti naujo rezultatų interpretatoriaus. Dažniausiai programuotojai nori valdyti savo programos pokyčius, todėl geriau gauti pertvarkos rezultatus kaip rekomendacijas. Tada programuotojas pats nuspręs, ar jam juos įgyvendinti praktiškai, ar ne. Tačiau galima šį įrankį realizuoti kaip papildinį programų kūrimo aplinkoje. Taip pat galima pasinaudoti esamais pertvarkų įrankiais, kurie nemoka aptikti pertvarkų, bet moka jas įgyvendinti. Pertvarkos programos rezultatas parodys, kurioje vietoje reikia atlikti pertvarką, o tada su kitu įrankiu pats programuotojas gali tą vietą pažymėti ir pertvarkyti, jei jam tai atrodo būtina.

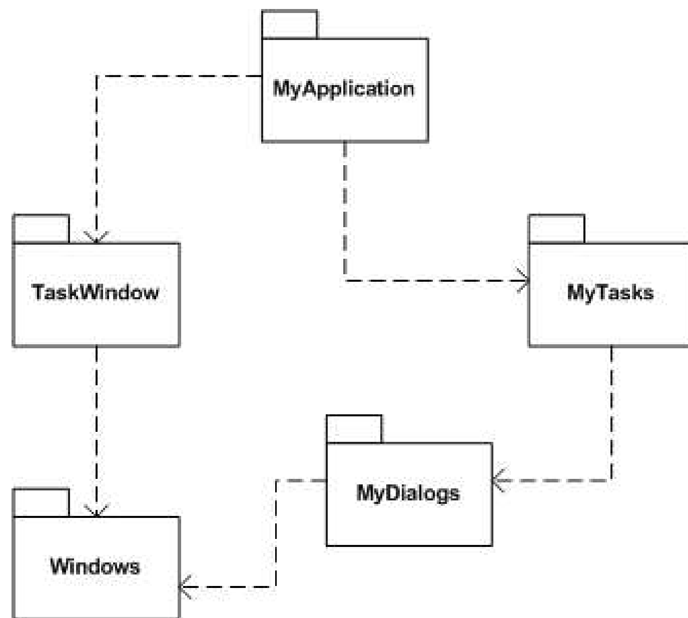
Kitame skyriuje nagrinėjamos kelios pertvarkos. Detaliai pateiktos jų pertvarkų kalbos, nagrinėjama, kaip jos gaunamos. Per tuos pavyzdžius parodoma, kad šis metodas yra pakankamai paprastas ir priimtinas norint atlikti įvairias pertvarkas.

4. Pertvarkų pavyzdžiai

Geriausia suprast šio įrankio teikiamas galimybes ir privalumus yra per pavyzdžius. Šiame įrankyje bus nagrinėjamos dvi pertvarkos: ciklų radimas paketuose, ir metodų matomumo zonų nustatymas. Svarbiausias dėmesys bus sutelktas į pertvarkų programos parašymą, jei galima realizuoti pertvarkos programą, vadinasi galima realizuoti ir visą įrankį. Net ir nerealizavus rezultatų interpretatoriaus, pertvarkos programa turi realią naudą, nes jos rezultatais galima pasinaudoti tiesiog kaip informacija apie programą.

4.1. Paketų ciklai

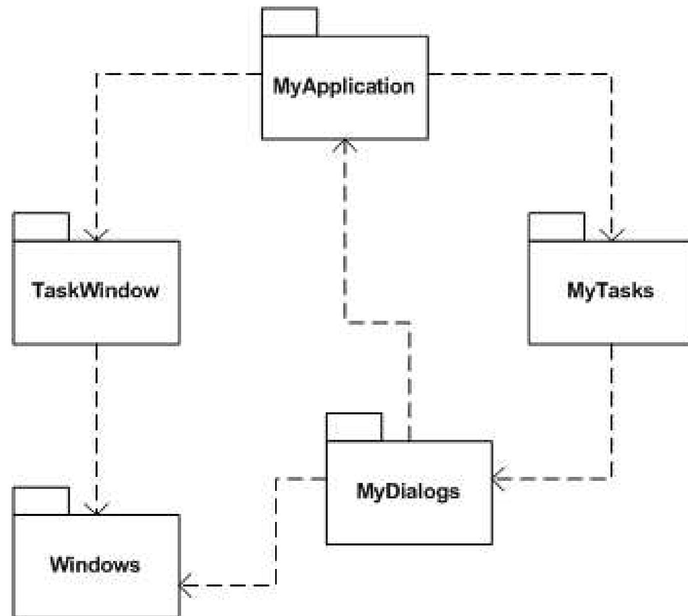
Didelėse objektiškai orientuotose programose neišsiverčiama be paketų, kurie sugrupuoja klases pagal tam tikras savybes, bei kriterijus. Ryšiai tarp paketų teikia aukšto lygio informaciją apie sistemą. Sistemos dažniausiai projektuojamos būtent nuo paketų ryšių, tada leidžiamasi į paketų lygį, ir detalizuojamos klasės ir jų ryšiai. Tvarkingame projekte paketai būna sujungti beciklių orientuotu grafu. Šiame skyriuje nagrinėjamas pavyzdys [Mar96] 3 paveikslėlyje.



11 paveikslėlis. Paketai be ciklų

Esant tokiai paketų hierarchijai, kiekvienas paketas gali būti keičiamas, nes žinoma tiksliai, kokie paketai nuo jo priklauso, ir kokie ne. Pavyzdžiui, jei keičiamas *MyDialogs* paketas, žinoma, kad reikės peržiūrėti ir *MyTasks*, bei *MyApplication* paketus. Visiems kitiems paketams yra nesvarbu, ar *MyDialogs* keičiamas, ar ne, nes jie neturi jokio ryšio su juo. Tačiau, jei atsirado

reikalavimas pakeisti vieną MyDialogs paketo klasę, ir joje reikia naudoti MyApplication vienos klasės metodą, tada ryšiai tarp paketų persipiešia, kaip parodyta 4 paveikslėlyje.



12 paveikslėlis. Paketai su ciklu

Šis pakeitimas sukuria ciklą tarp paketų: *MyApplication* → *MyTasks* → *MyDialogs*→ *MyApplication*. Tokie ciklai tarp paketų sukuria svarbių problemų. Bet kurio cikle esančio paketo pakeitimas gali sukelti neplanuotų pasekmių, jei tas programuotojas nežino, kad susidarė ciklas. Todėl esant ciklui, reikia pritaikyti iškart visus ciklo paketus, jei norima keisti tik vieną. Praktiškai žiūrint, ciklo paketai tampa kaip vienu dideliu paketu. Todėl tokių ciklų yra būtina išvengti, o tai nėra sudėtinga atlikti. Tiesiog iškelti cikle naudojamą metodą į abstrakčią klasę, tada šią klasę paveldėti, o ciklą sudarančioji klasė tiesiog sukuria ryšį jau su abstrakčia klase, kuri yra tame pačiame pakete. Šiai problemai spręsti galima suformuluoti pertvarkos užduotį: surasti paketų ciklus. Šiame pavyzdyje naudojami tokie pertvarkų duomenys:

```
package(myApp).  //MyApplication
package(myTask). //MyTasks
package(myDial). //MyDialogs
package(taskWin). //TaskWindow
package(win).     //Windows
```

Šiai pertvarkai įgyvendinti užtenka tokių duomenų:

```
packageUsed( myApp, [taskWin,myTask] ).
packageUsed( taskWin, [win] ).
packageUsed( myTask, [myDial] ).
packageUsed( myDial, [win, myApp] ).
```

```
packageUsed( win, [] ).
```

Šiuos duomenis nėra sudėtinga gauti iš pertvarkomos programos. *Reflection* bibliotekos tokią informaciją suteikia. Šiai pertvarkai įgyvendinti šių dviejų rūšių duomenų užtenka. Pertvarkos programa *cycles(Package, Reference)* atrodytų taip:

```
cycles(X,P) :- cycles(X, X, P,[]).
cycles( X, _, [],A) :- member(X,A),!.
cycles( X, Y, P, A) :-
    packageUsed( Y, ArcList ),
    member( Z, ArcList ),
    not(member( Z, A)),
    P = [Z|PTail],
    cycles( X, Z, PTail,[Z|A] ).
```

Kaip ir visos loginės programos ji veikia dvejopai: įvedus konkrečią klasę, atsakoma, ar ji sudaro ciklą, ar ne, neįvedus jokios klasės, išvedami visi esantys pertvarkų duomenyse ciklai:

7 lentelė. Pertvarkos rezultatai

Programos rezultatai	Užduotis
`?- cycles(taskWin,Reference).` `false.`	Kai tikrinamas konkretus paketas. Atsakymas: paketas nesudaro ciklų su kitais paketais.

`?- cycles(myDial,Reference).`
`Reference = [myApp, myTask, myDial] ;`
`false.`
	Kai tikrinama konkretus paketas. Atsakymas: paketas sudaro ciklą, išvedami visi paketai, kurie dalyvauja cikle.

`?- cycles(Package,Reference).`
`Package = myApp,`
`Reference = [myTask, myDial, myApp] ;`
`Package = myTask,`
`Reference = [myDial, myApp, myTask] ;`
`Package = myDial,`
`Reference = [myApp, myTask, myDial] ;`
`false.`
	Kai tikrinami visi apibrėžti paketai. Išvedami cikluose dalyvaujantys paketai, ir tų ciklų kiti paketai. Čia pakartojamas tas pats ciklas tiek kartų, kiek turi paketų.

Toks sprendimas jau tenkina pertvarkos užduotį: rado paketų ciklus, tačiau galima įvesti papildomą reikalavimą, kad būtų galima nuspręsti, kuriame pakete reikia nutraukti ciklą. Tai galima padaryti suskaičiavus paketų stabilumo matą [Mar97]: I = Ce / (Ca + Ce), kur Ce – vidinis paketo klasių skaičius, kurios priklauso nuo išorinių paketų, Ca – išorinių klasių skaičius, kurios naudoją

šio paketo klases. Jei I yra 0, tai reiškia, kad paketas yra maksimaliai stabilus, nes neturi ryšių su išorinėmis klasėmis. Jei I yra lygus 1, tai rodo maksimaliai nestabilų paketą, kuris nėra naudojamas jokiame kitame pakete, tik jis kreipiasi į kitus paketus. Kuo paketas yra stabilesnis, tuo jo klasės turi būti abstraktesnės, jei I yra 0, vadinasi pakete turi būti tik abstrakčios klasės. Šiuo matu galima pasinaudoti pašalinant paketuose ciklus. Reikia rasti stabiliausią klasę, kai reikia paveldėti abstrakčią klasę, kurioje būtų metodai, kurių reikia į ją besikreipiančiai klasei. Tokiu būdu bus panaikintas ciklas. Norint šiuo matu papildyti pertvarkos programą, reikia praplėsti pertvarkos duomenis, įvedant informaciją, kiek kiekvienas paketas turi ryšių su kitais paketais, šią informacija nesudėtingai galima gauti *Reflection* pakalba. Tai gi įterpiami dar tokie duomenys apie paketus *packageUsed(Package, Ca,Ce)*:

```
packageUsed( myApp,3,1 ).
packageUsed( taskWin, 1,1 ).
packageUsed( myTask, 1,2 ).
packageUsed( myDial, 2,1 ).
packageUsed( win, 0,2 ).
```

Taip pat reikia sukurti papildomą taisyklę, kuri skaičiuotų mato reikšmę:

```
packageStability(P,I) :-packageUsed(P, Ca,Ce),
                        I is Ce / (Ca+Ce).
```

Ir papildyti pagrindinę pertvarkos programą:

```
cycles(X,P,I) :- cycles(X, X, P,[]), packageStability(X,I).
```

Tada rezultatai gaunami tokie:

8 lentelė. Pertvarkų rezultatai

Programos rezultatai	Užduotis
`?- cycles(taskWin,Reference, I).` `false.`	Kai tikrinamas konkretus paketas. Atsakymas: paketas nesudaro ciklų su kitais paketais.

`? ?- cycles(myDial,Reference,I).`
`Reference = [myApp, myTask, myDial],`
`I = 0.333333 ;`
`false.`
	Kai tikrinama konkretus paketas. Atsakymas: paketas sudaro ciklą, išvedami visi paketai, kurie dalyvauja cikle, bei šio paketo stabilumo matas.

`?- cycles(Package,Reference,I).`
`Package = myApp,`
`Reference = [myTask, myDial, myApp],`
	Kai tikrinami visi apibrėžti paketai. Išvedami cikluose dalyvaujantys paketai, ir tų ciklų kiti paketai. Čia pakartojamas tas

`I = 0.25 ;`
`Package = myTask,`
`Reference = [myDial, myApp, myTask],`
`I = 0.666667 ;`
`Package = myDial,`
`Reference = [myApp, myTask, myDial],`
`I = 0.333333 ;`
`false.`
	pats ciklas tiek kartų, kiek turi paketų. Taip pat išveda ir stabilumo matą, iš kurio matoma, kuris yra stabiliausias, ir perspektyviausias atlikti pertvarkai iki galo.

Dabar programuotojas jau žino, kuriam paketui reikia kurti abstrakčią klasę, kad panaikinti ciklą. Šią pertvarkos programą galima būtų dar plėsti, įvedant pertvarkos duomenyse ne tik paketų ryšių skaičių, bet ir konkrečiai, kokios klasės, kokius metodus kviečia. Turint tokius duomenis galima praplėsti pertvarką, kad suradus pertvarkomą paketą, išvestų, kuriuos metodus reikia realizuoti paveldėjus abstrakčią klasę, kitaip sakant, kurie šio paketo metodai yra naudojami kitų paketų, kurie sudaro ciklus. Pertvarkos praplėtimui reikalinga informacija pateikiama tokiais duomenimis:

```
packageRef( myApp, app, calc, myDial).
packageRef( myTask, task, list, myApp).
packageRef( myTask, task, count, myApp).
packageRef( myDial, dial, div, myTask).
packageRef( taskWin, win, form, myApp).
packageRef( win, window, visible, taskWin).
packageRef( win, window, enable, myDial).
```

Turint tokią detalią informaciją apie paketus galima patikrinti, ar teisingai buvo sugeneruoti duomenys, kurie pateikė kiekvieno paketo naudojimo aibę. Tam galima užrašyti tokią programą ir jos rezultatai:

<table>
<tr><td colspan="2">Programa:</td></tr>
<tr><td colspan="2"><pre>packageUse2(P,S) :-
      package(P),
      findall(L,packageRef(L,_,_,P),L),
      list_to_set(L,S).</pre></td></tr>
<tr><td>Rezultatai <code>packageUsed2:</code></td><td>Rezultatai <code>packageUsed/2:</code></td></tr>
<tr><td><pre>?- packageUsed2(Package, List).
Package = myApp,
List = [myTask, taskWin] ;
Package = myTask,
List = [myDial] ;
Package = myDial,
List = [myApp, win] ;</pre></td><td><pre>?- packageUsed(Package, List).
Package = myApp,
List = [taskWin, myTask] ;
Package = taskWin,
List = [win] ;
Package = myTask,
List = [myDial] ;</pre></td></tr>
</table>

`Package = taskWin,`
`List = [win] ;`
`Package = win,`
`List = [].`
	`Package = myDial,`
	`List = [win, myApp] ;`
	`Package = win,`
	`List = []`

Iš šio pavyzdžio galima matyti, kad galima įvairiai generuoti pertvarkai reikalingus duomenis. Pirmu atveju duomenų generatorius gavo visą reikiamą informaciją ,ir pateikė kaip pertvarkų duomenis, antru atveju, duomenų generatorius perdavė smulkią, bet neapdorotą informaciją, ir ją apdorojo pertvarkų programa, kuri gavo tuos pačius duomenis. Taip pat pasinaudojus detaliais paketų priklausomybių duomenimis galima gauti informaciją ir apie Ce ir Ca koeficientus reikalingus stabilumo matui skaičiuoti:

Programa:

```
packageUsed3(P,A,E) :-package(P),
      findall(L, packageRef(L,_,_,P), L), length(L, A),
      findall(M, packageRef(P,_,_,M), M), length(M, E).
```

Rezultatai `packageUsed3`:	Rezultatai `packageUsed/3`:

`?- packageUsed3(Package,A,E).`
`Package = myApp,`
`A = 3,`
`E = 1 ;`
`Package = myTask,`
`A = 1,`
`E = 2 ;`
`Package = myDial,`
`A = 2,`
`E = 1 ;`
`Package = taskWin,`
`A = 1,`
`E = 1 ;`
`Package = win,`
`A = 0,`
`E = 2.`
	`?- packageUsed(Package,A,E).`
	`Package = myApp,`
	`A = 3,`
	`E = 1 ;`
	`Package = myTask,`
	`A = 1,`
	`E = 2 ;`
	`Package = myDial,`
	`A = 2,`
	`E = 1 ;`
	`Package = taskWin,`
	`A = 1,`
	`E = 1 ;`
	`Package = win,`
	`A = 0,`
	`E = 2.`

Sėkmingai realizuotas ir šių duomenų gavimas. Galima daryti išvadą, kad pertvarkų kalbą galima naudoti ne tik pertvarkų programų kūrimui, bet ir išvestinių duomenų kūrimui, nes yra paprasčiau parašyti logine kalba programą apdorojančia duomenis, nei su duomenų generatoriumi. Pradžioje pertvarkos duomenų aibė buvo didesnė: paketų ryšiai su kitais paketais, paketų ryšių skaičius, dabar visą šią informaciją galima gauti vien tik iš detaliau sugeneruotų duomenų, apie konkrečių metodų panaudojimą. Turint šią informaciją galima toliau vystyti šią pertvarką. Tai gi turint programą, kuri randa ciklus, bei turint duomenis apie detalius ryšius, galima parašyti tokią programą, kuri išveda visus metodus, kuriuos galima iškelti į abstrakčią klasę:

Programa:

```
    abstract(P,C,M,K,I) :- cycles(P,L,I),
```

```
            packageRef(P,C,M,K), member(K,L).
```

Rezultatas:

```
?- abstract(PFrom,Class,Method,PTo,I).
PFrom = myApp,
Class = app,
Method = calc,
PTo = myDial,
I = 0.25 ;
PFrom = myTask,
Class = task,
Method = list,
PTo = myApp,
I = 0.666667 ;
PFrom = myTask,
Class = task,
Method = count,
PTo = myApp,
I = 0.666667 ;
PFrom = myDial,
Class = dial,
Method = div,
PTo = myTask,
I = 0.333333 ;
false.
```

Iš šio rezultato matyti, kaip susikuria ciklas. Patarimas naikinti ciklą iškeliant į abstrakčią klasę tą metodą, kurio paketo stabilumo koeficientas yra mažiausias. Tokios informacijos yra pakankama, kad būtų galima sukurti rezultatų interpretatorių. Yra žinoma, kurį metodą reikia iškelti į abstrakčią klasę, ir žinoma, kuriame pakete reikia pakeisti ryšį ir buvusios klasės į abstrakčią klasę, kad naudotų tik tuos metodus, kurių reikia, taip išvengiant ciklo.

4.2. Metodų matomumas

Metodų pasiekiamumas yra įvairesnis nei klasių, klasėse yra tik du pasirinkimai, o metoduose net 5 skirtingos matomumo zonos C# kalboje. Visos šios zonos lygiai taip pat yra svarbios naudoti paketuose, kaip ir klasių zonos. Tačiau objektiniame programavime metodams atsiranda daug daugiau įvairių reikalavimų, neužtenka padaryti metodo matomo tik jo klasėje, arba matomas visur, dažnai prireikia, kad metodas būtų matomas tik tame pakete, bet nebūtų matomas už paketo ribų, tam būtų galima padaryti klasę vidinio matomumo, bet jei klasė turi ir kitų metodų, kurie turi būti pasiekiami kituose paketuose, toks sprendimas tampa netinkamu ir reikia naujos matomumo zonos. Lygiai taip pat yra su klasių paveldėjimu, kartais yra norima, kad metodas nebūtų matomas kitose klasėse, bet būtų matomas vaikinėse klasėse, arba norima, kad dar ir būtų matomas tame pakete, o kituose paketuose būtų matomas tik per paveldėjimą. Metodai turi daug daugiau paskirčių nei

klasės. Ne visi metodai turi būti pasiekiami programuotojui, tačiau galbūt turi būti pasiekiami išvestinėse klasėse, tokių praktinių situacijų pasitaiko labai dažnai. Todėl metodų atveju yra daug platesnis prieinamumo ratas, kuris pateikiamas 9 lentelėje.

9 lentelė. Matomumo zonos

Raktažodis	Matomumo zona
`private` arba `nieko`	Metodas pasiekiamas tik šioje klasėje
`internal`	Pasiekiamas tik šiame pakete
`protected`	Pasiekiamas tik šioje klasėje ir visose vaikinėse klasėse
`protected internal`	Pasiekiamas tik šiame pakete ir visose vaikinėse klasėse
`public`	Pasiekiamas visuose paketuose

4.2.1. Nekokybiško programavimo atsiradimas

Dažniausia programuotojų klaida, kaip ir klasių atveju, yra naudoti tik viešą ir privatų prieinamumą prie metodų, tačiau tai atspindi tik labai nedidelę dalį realios paketų struktūros. Daugiausia kyla problemų dėl viešų metodų, nes jų būna labai daug, metodų būna dešimtimis kartų daugiau nei klasių, todėl tinkamai naudoti metodų matomumo zonas yra žymiai svarbiau nei klasių, todėl programuotojai turi gerai išmanyti jų programavimo kalbos siūlomus variantus.

Dažnai iškart kuriant klasę galima žinoti, ar ji bus viešai prieinama, ar tik iš esamo paketo, tačiau su metodais yra kur kas sudėtingiau. Metodai dažnai yra rašomi ir perrašomi, todėl iškart nuspręsti jų matomumo zoną yra gana sudėtinga, todėl dažniausiai programuotojai juos padaro viešus, arba privačius, o kartais tik viešus. Tačiau vėliau ištaisyti metodų matomumo zonas pamirštama, arba tas darbas tampa labai sudėtingu, todėl šiuo atveju labai praverstų automatinis pertvarkos aptikimas pasinaudojant loginiu programavimu. Juk būna labai nepatogu, kai norima pasinaudoti klase, kuri atlieka keletą funkcijų, bet joje metodų randama keliolika kartų daugiau nei atrodo reikėtų, ir tada tokios klasės naudojimas labai apsunkinamas. Tarkime yra klasė, kuri suranda pigiausią lėktuvo bilietą pagal pasirinktus miestus ir datas. Pagal klasės specifikaciją, tikimąsi rasti tokius metodus: *Kaina(string isMiesto, string iMiesta, date kada),* ir dar yra keletas šio metodo perkrovų: *Kaina(string isMiesto, string iMiesta, date pirmynData, date AtgalData), Kaina(string isMiesto, string iMiesta, date pirmynData, date atgalData, integer KainosRiba)* ir panašių, taip pat yra ir kitų metodų, kurie randa iš kokių miestų į kokius skraidoma, grąžina oro bendroves, kurios skraido tarp tų miestų ir kitų, tačiau šioje klasėje yra ir tokių metodų kaip oro mokesčių apskaičiavimo, oro taršos kvotos, ir kitų, kurie neturėtų būti pasiekiami klasės vartotojams, tačiau jie

buvo naudojami kitose klasėse, ir programuotojas jų nepadarė privačių, tačiau palikęs juos viešus, padarė matomus šio paketo naudotojams, todėl šiuo atveju naudotojas gauna informaciją, kuri jo darbą apsunkina, jis nežino, kuriuos metodus jam reikia naudoti, kurių ne, kurie jam yra naudingi, o kurie ne, todėl yra labai svarbu tinkamai nustatyti visų metodų matomumo zonas. Nagrinėjamas atvejis, kai metodams yra suteikiamas pati griežčiausia – labiausiai ribojanti pasiekiamumą matomumo zona.

4.2.2. Metodų matomumo pertvarkos programa

Su metodais yra kur kas sudėtingiau nei su klasėmis, nes yra daug daugiau variantų, ir taip pat metodų matomumas priklauso nuo jų klasių matomumo, metodo matomumas negali būti platesnis, nei klasės matomumas, kitaip sakant, klasė apriboja metodo pasiekiamumą. Taip pat yra paranku rasti metodus, kurie yra sukurti, bet niekur nepanaudoti. Metodų matomumo zonos yra viena kitos poaibiai, siauriausia *private*, toliau eina *internal, protected*, ir platesnis *internal protected*, plačiausia matomumo zona yra *public.* Tokia eilės tvarka ir bus apibrėžiamos loginės užklausos. Papildomai yra sukurta užklausa, kuri randa nenaudojamus metodus.

Šioms užklausoms įgyvendinti reikia papildyti turimą faktų aibę keleta naujų faktų: *classHierarchy(Assembly, Class, SubAssembly, SubClass)* – nurodo paveldėjimo ryšius tarp klasių, nurodant klasės paketą, pačią klasę, vaikinės klasės paketą, bei pačią vaikinę klasę; *methods(Assembly, Class, Method)* – nurodomi visų klasių metodai, pirmiausia metodo klasės paketas, klasė ir metodas; *methodInUsed(Assembly, Class, Method, InAssembly, InClass)* – nurodomi metodų panaudojimai, pirmiausia metodo klasės paketas, klasė, metodas, tada nurodoma, kuriame klasės pakete jis panaudotas, bei kurioje klasėje, nereikia detalizuoti, kuriame metode naudojamas, jei metodas naudojamas tik savo klasėje, tai irgi reikia apibrėžti. Šiuos faktus automatiškai sugeneruoti iš turimo projekto nėra sudėtinga, kaip ir klasių atveju reikia pasinaudoti System.Reflection .Net klasėmis. Bendras visoms užklausoms predikatas bus toks: *methodAccess(Method, Class, Assembly, Access)* – nurodomas metodas, metodo klasė, klasės paketas, bei matomumo zona. Kuriant užklausas bus remiamasi 10 lentelės duomenimis.

10 lentelė. Matomumas paketuose

Raktažodis C#	Šioje klasėje	Šiame pakete	Visuose paketuose	Vaikinėse klasėse
`private (nieko)`	X			
`internal`	X	X		
`protected`	X			X

protected internal	X	X		X
public	X	X	X	X

4.2.2.1. Nenaudojamų metodų aptikimas

Pirmiausia bus pateikta nenaudojamų metodų aptikimo užklausa. Pirmiausia, kaip ir visoms kitoms užklausoms, reikia patikrinti, ar įvesti pradiniai duomenys yra korektiški, ar toks metodas išties egzistuoja atitinkamame pakete nurodytoje klasėje. Tada reikia nuskaityti visus ryšius, kur yra naudojamas šis metodas, taip sukuriant klasių, kuriose metodas naudojamas sąrašą, ir tiesiog reikia patikrinti, ar tas sąrašas yra tuščias, jei taip, vadinasi metodas nėra naudojamas. Užklausa atrodytų taip:

```
methodAccess(Method, Class, Assembly, notUsed) :-
  methods(Assembly, Class, Method),
  findall(
  InClass,methodInUsed(Assembly, Class, Method,_, InClass),[]
  ).
```

Šioje užklausoje panaudotas standartinis predikatas *findall*, kuris grąžina visas konkretaus predikato konkrečias reikšmes: *findall(+Template, :Goal, -Bag)*, kur pirmas narys yra ieškomas domenas, antras yra predikatas, kuriame ieškoma, trečiasis narys grąžina rezultatų sąrašą. Jei sąrašas apsibrėžiamas kaip tuščias [], vadinasi šis predikatas bus teisingas tik su tokiais metodais, kurie nėra naudojami nei viename ryšyje apibrėžtame fakte *methodInUsed*. Atlikus šią užklausą pagal 3 priede pateiktus duomenis, rezultatas gaunamas toks:

```
?- methodAccess(Method,Class,Assembly,notUsed).
Method = aF1, Class = a1, Assembly = a ;
Method = aA3, Class = a3, Assembly = a ;
Method = bB1, Class = b1, Assembly = b ;
```

Taip pat galima užklausą pateikti ir kitaip, pavyzdžiui, jei norima gauti tik vieno *„a“* paketo nenaudojamus metodus: *?- methodAccess(Method,Class,a,notUsed).*, arba tik konkrečios klasės *„a1“* nenaudojamus metodus: *?- methodAccess(Method,a1,a,notUsed).* Programuotojas gali spręsti, ar šie metodai yra nereikalingi, ir gali juos ištrinti, arba savarankiškai pakeisti jų matomumo zoną, kurios pertvarkos įrankis negalėjo pasiūlyti, nes metodas neatitinka nei vienos zonos kriterijų.

4.2.2.2. *Private* metodų aptikimas

Ši metodo zona yra labiausiai apribota, todėl ją aptikti yra gana nesudėtinga, tereikia patikrinti visas klases, kuriose yra naudojamas metodas, ir jei visos klasės sutampa su to metodo klase, bei sutampa paketai, vadinasi metodas turi būti privatus. Gaunama tokia užklausa:

```
methodAccess(Method, Class, Assembly, private) :-
methods(Assembly, Class, Method),
findall(
InClass,methodInUsed(Assembly, Class, Method,_,
InClass),[HeadClass|Tail]
),
findall(
InAssembly,
methodInUsed(Assembly, Class,
Method,InAssembly,_),[HeadAssembly|_]
),
HeadClass = Class, HeadAssembly = Assembly, Tail = [].
```

Pirmiausia patikrinama metodo aprašymo korektiškumas, tada gaunamos visos klasės ir tų klasių paketai, kuriose tas metodas yra naudojamas. Tariant, kad apsibrėžiant faktus, metodų panaudojimas konkrečioj klasėj bus apibrėžtas viename fakte, o jei jis yra toje klasėje naudojamas kelis kartus, tai bus nebekartojama. Todėl patikrinus rezultato sąrašo pirmą narį, ir jam esant tokiam pačiam kaip metodo klasė, o sąrašo pabaiga yra tuščia, galima daryti išvadą, kad metodas turi būti privatus, paketo sąrašo pabaigos nebūtina tikrinti, nes jo ilgis sutampa su klasės sąrašo ilgiu, ir esant klasių sąrašui tuščiam, taip ir paketų sąrašas bus tuščias. Atlikus analogišką užklausą privačių metodų gavimui, gaunamas toks rezultatas:

```
?- methodAccess(Method,Class,Assembly,private).
Method = aA1, Class = a1, Assembly = a ;
Method = aA2, Class = a2, Assembly = a ;
Method = bA2, Class = b2, Assembly = b ;
```

Analogiškai galima patikrinti ir konkretaus paketo privačius metodus, ar konkrečios klasės, kaip ir nenaudojamų metodų aptikimo atveju. Reikia atkreipti dėmesį, kad atlikus visus šiuos predikatus, negali kartotis metodai keliuose iš jų, predikatai aptinka labiausiai apribotą matomumo zoną metodams, nes paprasčiausiu atveju galima būtų visus metodus padaryti viešais, todėl griežtai iš kiekvienos zonos aibės išimami kitos zonos metodai, nes vienos zonos yra kitų zonų poaibiai, pavyzdžiui, *public* zonos poaibiai yra visos kitos zonos.

4.2.2.3. *Internal* metodų aptikimas

Ši zona, kaip ir *protected* zona, yra platesnė už *private* zoną. Šios zonos metodai turi pasižymėti ta savybę, kad jie pasiekiami tik savo klasėje ir savame pakete, iš šio apibrėžimo išplaukia, kad privatūs metodai yra šios zonos poaibis, todėl iš šios zonos gautų metodų reikės atimpti privačius metodus, kurie gaunami ankstesne užklausa. Tiesiog reikia patikrinti, ar metodas naudojamas tik savame pakete, tam bus įvedamas pagalbinis predikatas *allEquals*, kuris patikrins, ar paduotame sąraše yra tik vieno konkretaus domeno įrašai, šiuo atveju, ar metodo naudojamų paketų sąrašas sudarytas tik iš to paties metodo, kuriame yra pats metodas. Šio predikato apibrėžimas:

```
allEquals([Head|Tail],Assembly) :-
 Head = Assembly, Tail = [].
allEquals([Head|Tail],Assembly) :-
 Head = Assembly, allEquals(Tail, Assembly).
```

Pirmu predikatu yra tikrinama, ar sąrašo pirmas elementas yra lygus paketui, ir ar likęs sąrašas yra tuščias, jei ne, tada kartojama su trumpesniu sąrašu (atmetus pirmąjį elementą) tol, kol sąrašas bus tuščias, arba bus rastas kitoks paketas. Pavyzdžiui, ?- *allEquals([a,a,a],a)* grąžins *true*, o predikatas *?- allEquals([a,a,b,a],a)* grąžins *false*. Pritaikius šį predikatą kuriamai užklausai, ji gaunama tokia:

```
methodAccess(Method, Class, Assembly, internal) :-
 methods(Assembly, Class, Method),
 findall(
 InAssembly,methodInUsed(Assembly, Class, Method,InAssembly,_),List
 ),
 allEquals(List,Assembly),
 not(methodAccess(Method, Class, Assembly, private)).
```

Patikrinamas metodo egzistavimas, tada gaunamas visų paketų, kuriuose šis metodas yra naudojamas sąrašas, po to patikrinama, ar tas sąrašas susideda tik iš to paties paketo, ir atsisakoma privačių metodų poaibio, nes pirmąsias reikšmes jie tenkina. Atlikus šios užklausos pavyzdį:

```
?- methodAccess(Method,Class,Assembly,internal).
Method = aB1, Class = a1, Assembly = a ;
Method = aB2, Class = a2, Assembly = a ;
Method = bA1, Class = b1, Assembly = b ;
```

Gaunami nauji metodai, kurie nėra privatūs, bet yra vidinio matomumo zonos, jei užklausoje nebūtų paskutinio predikato eliminuojančio privačius metodus, atsakyme dar būtų ir privatūs metodai, nes jie yra naudojami tik savo klasėje, kas tenkina vidinių metodų matomumo zoną.

4.2.2.4. *Protected* metodų aptikimas

Kita platesnė zona už privačią, bet tokio paties platumo kaip ir vidinio matomumo zona yra apsaugota zona, kuri pasiekiama tik toje pačioje klasėje, bei visose vaikinėse klasėse, net jei jos yra kituose paketuose. Šios zonos poaibis yra privati zona, tačiau vidinė zona nėra poaibis. Šiai užklausai taip pat yra sukurtas papildomas predikatas *checkProtected,* kuris tikrina, ar visos sąraše pateiktos klasės yra vaikinės turimai klasei, jei taip, tada metodas yra šios zonos. Šis predikatas apibrėžiamas sudėtingiau, nei *allEquals*, nes yra daugiau galimų variantų. Jo apibrėžimas:

```
checkProtected([HeadC|TailC],[HeadA|TailA],C,A) :-
 isSubClass(A, C, HeadA, HeadC),
 TailC = [],TailA = [].

checkProtected([HeadC|TailC],[HeadA|TailA],C,A) :-
 isSubClass(A,C,HeadA,HeadC),
 checkProtected(TailC,TailA,C,A).

checkProtected([HeadC|TailC],[HeadA|TailA],C,A) :-
 A = HeadA, C = HeadC,
 checkProtected(TailC,TailA,C,A).

checkProtected([HeadC|TailC],[HeadA|TailA],C,A) :-
 A = HeadA, C = HeadC,
 TailC = [], TailA = [].
```

Reikia tikrinti, ar klasė yra paveldėta, ar ta pati, todėl kiekvienu atveju yra po du predikatus. Šiame predikate taip pat naudojamas naujas predikatas *isSubClass*, kuris patikrina, ar duotos klasės yra susietos per paveldėjimą, nebūtinai pirmu lygiu. Apsaugota matomumo zona leidžia pasiekti šį metodą, kuris yra tėvinėje klasėje nebūtinai iš pirmo lygio vaikinių klasių. Šio metodo predikatas apibrėžiamas taip:

```
isSubClass(AssemblyD, DirectClass, Assembly, Class) :-
 classHierarchy(AssemblyD,DirectClass, Assembly,Class).
isSubClass(AssemblyD, DirectClass, Assembly, Class) :-
 classHierarchy(AssemblyD, DirectClass, AssemblyS, SubClass),
 isSubClass(AssemblyS, SubClass, Assembly, Class).
```

Tai klasikinis protėvio predikato apibrėžimas, šis predikatas naudoja *classHierarchy* faktus nurodančius klasių paveldėjimo ryšius. Sukūrus šiuos du naujus predikatus, galima apibrėžti ir *protected* matomumo zonos metodų aptikimo užklausą:

```
methodAccess(Method, Class, Assembly, protected) :-
methods(Assembly, Class, Method),
findall(
InClass,methodInUsed(Assembly, Class, Method,_,
InClass),ClassList
),
findall(
InAssembly,methodInUsed(Assembly, Class,
Method,InAssembly,_),AssemblyList
),
checkProtected(ClassList,AssemblyList,Class,Assembly),
not(methodAccess(Method, Class, Assembly, private)) .
```

Pirmiausia patikrinamas metodo egzistavimas, tada gaunamas klasių ir paketų sąrašas, kuriuose metodas yra naudojamas. Tie paketai paduodami *checkProtected* predikatui, kuris patikrina jų ryšius, kadangi į šią zoną pakliūna ir privatūs metodai, reikia juos išimti panaudojant jau apibrėžtą privačių metodų užklausą. Atlikus šią užklausą turimiems testiniams duomenims, gaunamas toks rezultatas:

```
?- methodAccess(Method,Class,Assembly,protected).
Method = aC1, Class = a1, Assembly = a ;
```

Šis metodas pagal pateiktus duomenis yra panaudotas *a2* ir *c1* klasėse, kurios pagal klasių ryšių faktus yra apibrėžtos kaip paveldinčios klasės: *a1* → *a2* → *c1*. Šis pavyzdys patikrina visus galimus variantus, paveldėjimą tame pačiame pakete, paveldėjimą kitame pakete, bei naudojimą toje pačioje klasėje, tačiau negali būti panaudotas tame pačiame pakete kitoje klasėje, kuri nėra išvesta klasė.

4.2.2.5. *Protected internal* metodų aptikimas

Ši matomumo zona yra apsaugotos ir vidinės zonos junginys, vadinasi, aibių požiūriu, į ją įeina tiek *private* metodai (jie į visas zonas įeina), tiek *protected* ir *internal*. Šios matomumo zonos aptikimo užklausai taip pat reikės papildomo predikato *checkProtectedInternal,* kuris patikrins, ar metodas yra ne tik tame pakete, bet ir ar paveldimose klasėse tame arba kituose paketuose. Jis

apibrėžiamas analogiškai *checkProtected* predikatui, tik tikrinama ne tik paveldėjimas, bet ir ar yra tas pats paketas.

```
checkProtectedInternal([HeadC|TailC],[HeadA|TailA],C,A):-
 isSubClass(A, C, HeadA, HeadC),
 TailC = [],TailA = [].

checkProtectedInternal([HeadC|TailC],[HeadA|TailA],C,A) :-
 isSubClass(A,C,HeadA,HeadC),
 checkProtectedInternal(TailC,TailA,C,A).

checkProtectedInternal([HeadC|TailC],[HeadA|TailA],C,A) :-
 A = HeadA, not(isSubClass(A, C, HeadA, HeadC)),
 checkProtectedInternal(TailC,TailA,C,A).

checkProtectedInternal([HeadC|TailC],[HeadA|TailA],C,A) :-
 A = HeadA, not(isSubClass(A, C, HeadA, HeadC)),
 TailC = [], TailA = [].
```

Principas yra toks pats, yra dvi pabaigos sąlygos, kai klasės yra viena kitos vaikas, arba jos yra tame pačiame pakete, jei dar sąrašas nėra tuščias, tikrinamos abi tos sąlygos, ir rekursyviai kartojamas tas pats predikatas, kol sąrašas bus tuščias. Jei metodo panaudojimo klasės yra tik išvestinės klasės, arba to paties paketo, vadinasi šis metodas turi būti *protected internal*. Panaudojus šį predikatą, gaunama tokia aptikimo užklausa:

```
methodAccess(Method, Class, Assembly, protected_internal) :-
methods(Assembly, Class, Method),
findall(
InClass,methodInUsed(Assembly, Class, Method,_,
InClass),ClassList
),
findall(
InAssembly, methodInUsed(Assembly, Class,
Method,InAssembly,_), AssemblyList
),
checkProtectedInternal(ClassList,AssemblyList,Class,Assembly
),
not(methodAccess(Method, Class, Assembly, private)),
```

```
not(methodAccess(Method, Class, Assembly, internal)),
not(methodAccess(Method, Class, Assembly, protected)).
```

Patikrinamas, ar metodas tikrai egzistuoja, tada nuskaitomas klasių bei paketų sąrašas, ir tikrinama anksčiau apibrėžtu predikatu, ar tenkinama ši matomumo zona, kadangi buvo minėta, kad ši zona apima ir kitas tris zonas, jas reikia pašalinti iš galimų atsakymų aibės panaudojant anksčiau apibrėžtus predikatus. Atlikus šią užklausą, gaunamas toks rezultatas:

```
?- methodAccess(Method,Class,Assembly,protected_internal).
Method = aD1, Class = a1, Assembly = a ;
```

Jei nebūtų eliminuoti kitos zonos iš šios aibės, atsakymas būtų apėmęs ir visas kitas zonas. Pateikdame pavyzdyje 3 priede parodyta, kad šis metodas naudojamas tik savo klasėje, vaikinėje klasėje, bei šio paketo kitoje klasėje. Jie jis nebūtų panaudotas kitoje šio paketo klasėje, matomumas iškart taptų *protected*, o jei nebūtų panaudotas vaikinėje, jis taptų *internal*.

4.2.2.6. *Public* metodų aptikimas

Ši zona yra pati plačiausia, į jos aibę pakliūna visos kitos matomumo zonos aibės, todėl galima daryti išvadą, jei metodas nepriklauso nei vienai prieš tai buvusiai aibei, tai arba jis nenaudojamas apskritai, arba jis yra viešas, kadangi apskritai nenaudojamų metodų aptikimas taip pat yra apibrėžtas, tada galima ir jį atimti iš visų metodų aibės, ir taip liks tik vieši metodai, tai gi šios matomumo zonos aptikimo užklausa yra pati paprasčiausia:

```
methodAccess(Method, Class, Assembly, public) :-
methods(Assembly, Class, Method),
not(methodAccess(Method, Class, Assembly, private)),
not(methodAccess(Method, Class, Assembly, internal)),
not(methodAccess(Method, Class, Assembly, protected)),
not(methodAccess(Method, Class, Assembly,
protected_internal)),
not(methodAccess(Method, Class, Assembly, notUsed)).
```

Tiesiog reikia patikrinti, ar metodas išties egzistuoja, ir tada pašalinti visus metodus iš visų metodų aibės pagal atitinkamą matomumo zoną. Įvykdžius šią užklausą gaunamas toks rezultatas pagal pateiktus testinius duomenis:

```
?- methodAccess(Method,Class,Assembly,public).
Method = aE1, Class = a1, Assembly = a ;
Method = cA1, Class = c1, Assembly = c.
```

Tokiu būdu nustatoma visų metodų matomumo zona. Jau turint apibrėžtus visas užklausas, galima jas panaudoti įvairiau, pavyzdžiui norima gauti visus vieno paketo metodus ir tų metodų matomumo zonas, o ne konkrečios zonos. Tam reikia atlikti tokią užklausą: *?- methodAccess(Method,Class,a,Access).* Gautas atsakymas:

```
Method = aF1,  Class = a1,  Access = notUsed ;
Method = aA3,  Class = a3,  Access = notUsed ;
Method = aA1,  Class = a1,  Access = private ;
Method = aA2,  Class = a2,  Access = private ;
Method = aB1,  Class = a1,  Access = internal ;
Method = aB2,  Class = a2,  Access = internal ;
Method = aC1,  Class = a1   Access = protected ;
Method = aD1,  Class = a1,  Access = protected_internal ;
Method = aE1,  Class = a1,  Access = public ;
```

Gauti visi *„a"* paketo metodai, nurodant jų klasės bei matomumo zonas. Lygiai taip pat galima šią užklausą panaudoti rasti konkrečios klasės visų metodų matomumo zonas, užklausa būtų tokia: *?- methodAccess(Method,c1,c,Access).*, kur ieško klasės *„c"* esančios *„c"* pakete visų metodų ir jų matomumo zonų.

4.3. Rezultatai

Šiame skyriuje įgyvendinti du praktiški pertvarkų pavyzdžiai, kurie realiai gali būti pritaikomi programuotojų. Šie pavyzdžiai parodė, kad išties nesudėtinga realizuoti pertvarkas, parodyta, kaip tai atlikti iš eilės. Kiekvieną sudėtingesnę pertvarką galima skaidyti į paprastesnius uždavinius ir tada jungti. Svarbiausia tinkamai realizuoti duomenų generatorių, nes jei juo nebus sukurti reikiami duomenys, nebus galima atlikti ir pertvarkų. Tačiau kaip buvo išnagrinėta, kad galima praktiškai gauti visą informaciją apie programas, ir jas vienareikšmiškai pavaizduoti pertvarkų duomenyse, vadinasi šita problema kaip ir išspręsta, lieka tik klausimas, kaip tobulinti pertvarkų duomenų aibę, kurią galima kaupti po truputį, arba galima pasinaudoti 1 priede esančiu SOUL programos duomenų aibe. Su rezultatų interpretatoriumi iškilo problemų, pasirodo yra sudėtinga automatizuotai keisti programas, tam reikia turėti specialius įrankius. Primityviausias būdas yra redaguoti tiesiogiai programų išeities tekstus, bet tai gali sukelti klaidų pavojų. Todėl geriausia šį įrankį naudoti duomenų atvaizdavimui vartotojui patogiu būdu. Jei duomenų generatorių užtektų sukurti vieną kartą turint pilną skirtingų reikiamų duomenų aibę, tai rezultatų interpretatorių reikėtų pildyti kiekvienai pertvarkai atskirai.

Pačių pertvarkų programų kūrimo būdas loginiu programavimu pasiteisino. Pavyzdžiai parodo, kad yra daug paprasčiau atlikti duomenų perrinkimą loginėmis taisyklėmis, negu rašant pertvarkom objektines programas. Loginės programos yra lengvai kuriamos, ir nesudėtingai keičiamos. Kiti programuotojai gali greitai į jas įsigilinti, ir pakeisti pagal savo poreikį.

Kiekviena pertvarka nagrinėja tik tokią pradinės programos dalį, kiek jos buvo pervesta į pertvarkos duomenis. Taip pat nebūtina transformuoti visos pradinės programos į pertvarkų duomenis, tiesiog reikia paimti tik tokią informaciją, kuri yra būtina pertvarkai atlikti. Šiuo įrankiu kuriant daugiau pertvarkų, turėtų susiformuoti pertvarkų duomenų aibė, kuri daugiau nebesiplės, tai yra detaliai padengs visą programą. Tada naujos pertvarkos bus rašomos tiesiog nagrinėjant šiuos duomenis. Šiems pertvarkų duomenims gauti reiks kiekvienai kalbai sukurti atskirus duomenų generatorius, nes iš kiekvienos kalbos skirtingai yra pasiekiama reikalinga informacija apie pačią programą. Net jei pertvarkų duomenys būtų generuojami tiesiai iš klasių išeities tekstų, kiekvienos kalbos sintaksė yra skirtinga, todėl ir skirtųsi generatorius. Tačiau turint duomenų generatorių, kuris automatiškai sukuria reikalingus duomenis pertvarkoms atlikti, toliau tereikia rašyti naujas pertvarkų programas.

Rezultatų interpretatorius yra sudėtingiausiai realizuojamas, jei norima pertvarką automatiškai atlikti pradinėje programoje. Pertvarkyti programas turint tik jų išeities tekstus yra sudėtinga. Todėl geriausia būtų integruoti rezultatų interpretatorių su programų kūrimo aplinka, jei ta aplinka suteikia galimybę diegti joje papildinius (angl. plug–in). Tada rezultatų interpretatorius gali būti daug lankstesnis, nes naudojasi pačios kūrimo aplinkos teikiamomis galimybėmis išeities teksto redagavimui. Dauguma pertvarkų įrankių taip ir veikia, pavyzdžiui, *ReSharper* yra papildinys *Visual Studio* aplinkai. Tokiame papildinyje galima sujungti ir duomenų generatorių, ir pertvarkų programų rašymą, tada būtų vienas įrankis, kuris atliktų tiek pertvarkų aptikimą, tiek jų realizaciją keičiant pertvarkomą programą.

Galima daryti išvadą, kad šio darbo tikslai pasiekti. Pateiktas modelis automatinių pertvarkų radimui, bei nesudėtingam jų kūrimui nepriklausomai nuo pertvarkomos programos kalbos. Taip pat pateikti kelios naudingos pertvarkos, ir kaip jos turėtų būti realizuojamos šiuo įrankiu. Nebuvo pateiktas rezultatų interpretatoriaus veikimas, nes jo kūrimas priklauso kiekvienu atveju nuo turimos programų kūrimo aplinkos.

Išvados ir pasiūlymai

Sėkmingai sukūrus automatinių pertvarkų metodus, galima daryti išvadą, kad šiame darbe įgyvendintas principinis tikslas – rastas būdas automatiškai aptikti nekokybišką kodą, bei realizuoti tokias pertvarkas nepriklausomai nuo programavimo kalbos. Pertvarkų kūrimas loginiu programavimu yra patogus dar ir tuo, kad su juo yra susipažinę dauguma programuotojų, o jei ir nesusipažinę, tai nėra sudėtinga ją įvaldyti, tiesiog reikia suprasti jos logiką, o sintaksė yra labai nesudėtinga.

Pateikus pertvarkos kūrimo pavyzdį loginės kalbos priemonėmis, galima daryti išvadą, kad įmanoma sukurti ir universalų pertvarkų įrankį. Tokio įrankio naudojimas būtų labai patogus ir nesudėtingas, nereikalaujantis ypatingų programuotojo resursų. Jam tiesiog reikėtų nuspręsti, kada atlikti norimą pertvarką, ir netgi nebūtina pasirinkt konkrečios pertvarkos, šis universalus įrankis galės pasiūlyti visus jame realizuotus pertvarkymus, kuriuos įmanoma pritaikyti analizuojamai programų sistemai, programuotojui liks tik nuspręsti, ar jis nori pasiūlytą vietą pertvarkyti. Programuotojas galės naudotis tiek esamomis pertvarkomis, tokiu atveju netgi nereikės mokėti loginės programavimo kalbos, tiek ir jei norės be jokių apribojimų galės praplėst tokio įrankio pertvarkų aibę, sukurdamas savo pertvarkos programą, naudodamasis turima pertvarkų duomenų aibe. Taip pat realizuoti ir rezultatų interpretatorių pasirinktu būdu.

Šis įrankis gali aptikti automatiškai pertvarkas programose, bet jų automatinis įgyvendinimas pradinėje programoje yra sudėtingas uždavinys. Šiame darbe apsiribota informaciniu rezultatų išvedimu vartotojui, ir pakeitimus realizuoti programoje paliekama programuotojui, tačiau visgi tam tikrom pertvarkom įmanoma realizuoti ir programos pertvarkymą, tai paliekama spręsti kiekvienai pertvarkos programai atskirai.

Šį įrankį galima naudoti bet kurioms pertvarkoms, kurios analizuoja programos struktūrą, o ne funkcionalumą, nebent, jei programuotojas sugebės aiškiai aprašyti matus, pagal kurias turi būti ieškoma pertvarkos vieta, tokiu atveju šis įrankis tampa neribotu, tiesiog viskas priklauso nuo programuotojo gebėjimų mokėti pertvarką aprašyti loginėmis taisyklėmis.

Kitas privalumas yra, kad pertvarkų programomis galima keistis su kitais programuotojais, net gi, jei jie programuoja kitokia programavimo kalba, nes įgyvendintas nepriklausomumas nuo kalbos, nebent, jei pertvarka naudoja specifinį tam tikrai kalbai būdingą bruožą, bet ir tai šį įrankį padaro parankų, nes su juo parašyti pertvarkos programą yra paprasčiau nei su ta pačia programavimo kalba.

Literatūros sąrašas

[BM08] P. Bulychev, M. Minea. Duplicate code detection using anti-unification. SYRCoSE 2008

[Bra03] F. M. Bravo. A Logic Meta-Programming Framework for Supporting the Refactoring Process. Vrije Universiteit Brussel. 2003

[CMM06] G. F. Carneiro, M. G. Mendonça, J.C.Maldonado. Automatic Detection of Refactoring Opportunities. Experimental Software Engineering International Week 2006

[DDN00] S. Demeyer, S. Ducasse, O. Nierstrasz. Finding Refactorings via Change Metrics. ACM Press 2000

[EL96] K. Erni, C. Lewerentz. Applying Design-Metrics to Object-Oriented Frameworks. Third International Software Metrics Symposium, 1996, p. 64

[EM02] E. Emden, L. Moonen. Java Quality Assurance by Detecting Code Smells. CWI. 2002

[Fow00] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2000

[Fow08] M. Fowler. Refactorings in Alphabetical Order http://www.refactoring.com/catalog/index.html 9,95KB, 2008

[GFT06] M. Goldstein, Y. A. Feldman. S. Tyszberowicz. Refactoring with Contracts. AGILE 2006 Conference 2006

[Haa03] R. de Haan. Using ASF+SDF for the Verification of Annotated Java Programs. Universiteit Van Amsterdam 2003

[Mar96] R. C.Martin. „Granuality“, http://www.objectmentor.com/resources/publishedArticles.html 1996

[Mar97] R. C.Martin. „Stability“, http://www.objectmentor.com/resources/publishedArticles.html 1997

[MBM06] W. Meuter, J. Brichau, K. Mens. SOUL Manual (draft) 2006

[MDJ01] T. Mens, S. Demeyer, D. Janssens. Object-Oriented Refactoring Using Graph Rewriting. Technical Report 2001

[MED05] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens. Formalizing Refactorings with Graph Transformations. Copyright Wiley, 2005

[Mef06] K. Meffert. Supporting Design Patterns with Annotations. 13th Annual IEEE International Symposium 2006

[Men05] T. Mens. On the Use of Graph Transformations for Model Refactoring. UMH 2005

[Mey92] B. Meyer. Applaying „Design by contract“. Interactive Software Engineering 1992

[MT01] T. Mens, T. Tourwe. A Declarative Evolution Framework for Object-Oriented Design Patterns. Vrije Universiteit Brussel 2001

[MT04] T. Mens, T. Tourwe. A Survey of Software Refactoring. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 2004

[MTM03] T. Mens, T. Tourwe, F. Munoz. Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring. Vrije Universiteit Brussel 2003

[MTR07] T. Mens, G. Taentzer, O. Runge. Analysing Refactoring Dependencies Using Graph Transformation. Springer, 2007

[Opd92] W.F. Opdyke. Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks, PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[Rei01] S. P. Reiss. Consistent Software Evolution. Brown University 2001

[Rob99] D. B. Roberts. Practical analysis for refactoring. University of Illinois at Urbana-Champaign. 1999

[Rue05] H. P. Rueda. Characterization and Detection of Concerns in Java Code. 2005

[SLL99] F. Simon, S. Löffler, C. Lewerentz. Distance based cohesion measuring. Technical University Cottbus

[SS07] K. Stroggylos, D. Spinellis. Refactoring – Does it improve software quality? Fifth International Workshop on Software Quality 2007

[SSL01] F. Simon, F. Steinbrückner, C. Lewerentz. Metrics based refactoring. IEEE Computer Society 2001

[TBM02] T. Tourwe, J. Brichau, T. Mens. Using Declarative Metaprogramming To Detect Possible Refactorings. Vrije Universiteit Brussel 2002

[TM03] T. Tourwe, T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. Vrije Universiteit Brussel 2003

[Vik06] A. Viklund. Declarative Meta Programming http://prog.vub.ac.be/DMP/ 2006

[VTB00] K. De Volder, T. Tourwe, J. Brichau. Logic Meta Programming as a Tool for Separation of Concerns, Programming. Vrije Universiteit Brussel, 2000

[WD01] R. Wuyts, S. Ducasse. Symbiotic Reflection between an Object-Oriented and a Logic Programming Language. Universitat Bern, Switzerland 2001

1 priedas. SOUL LiCoR predikatai

classInPackage(class,package)

classInPackageNamed(class,packageName)

instanceVariableInClassChain(InstanceVariable,Class)

superclassOf(SuperClass,SubClass)

abstractClass(class)

classAbove(Class,Super)

classBelow(Sub,Root)

classesUnderstandingSelectorlist(classList,selectorList)

classInHierarchyOf(Sub,Root)

classUnderstands(class,selector)

instanceVariableInClass(InstanceVariable,Class)

methodInClass(Method,Class)

protocolInClass(Protocol,Class)

baseClass(Class)

class(Class)

classVariable(ClassVar)

instanceVariable(InstVar)

method(method)

rootClass(Class)

abstractMethod(method)

abstractMethodInClass(method,class)

methodWithNameInClass(method,selector,class)

methodWithName(method,selector)

ir kiti

SOUL LiCoR metodų rinkinys kodo generavimui

compileClass(ClassName)

compileClassCanonical(classname,superclassname,instvarlist,classvarlist,namespace)

compileClassMethod(class,protocol,code)

compileClassWithVars(classname,superclassname,instvarnames,classvarnames)

compileConstructorInClass(Code,Class)

compileMethod(class,code)

compileMethodInClassWithVisibility(Code,Class,Visibility)

removeClass(class)

removeMethodWithName(class,methodName)

2 priedas. Paketų ciklų pertvarkos predikatai

PREDIKATAS

packageUsed(Package, list of package used).

FAKTAI

packageUsed(myApp, [taskWin,myTask]).

packageUsed(taskWin, [win]).

packageUsed(myTask, [myDial]).

packageUsed(myDial, [win, myApp]).

packageUsed(win, []).

PREDIKATAS

cycles(Package, list of package used).

FAKTAI

```
cycles(X,P) :-cycles(X,P,_).
cycles(X,P,I) :- cycles(X, X, P,[]), packageStability(X,I).
cycles( X, _, [],A) :- member(X,A),!.
cycles( X, Y, P, A) :-
   packageUsed( Y, ArcList ),
   member( Z, ArcList ),
   not(member( Z, A)),
   P = [Z|PTail],
   cycles( X, Z, PTail,[Z|A] ).
```

PREDIKATAS

packageUsed (Package, Outside, Inside).

FAKTAI

packageUsed(myApp,3,1).

packageUsed(myTask, 1,2).

packageUsed(myDial, 2,1).

packageUsed(taskWin, 1,1).

packageUsed(win, 0,2).

packageStability(P,I) :-packageUsed(P, Ca,Ce), I is Ce / (Ca+Ce).

PREDIKATAS

packageRef (Package, Class, Method, UseInPackage).

FAKTAI

packageRef(myApp, app, calc, myDial).

packageRef(myTask, task, list, myApp).

packageRef(myTask, task, count, myApp).

packageRef(myDial, dial, div, myTask).

packageRef(taskWin, win, form, myApp).

packageRef(win, window, visible, taskWin).

packageRef(win, window, enable, myDial).

PREDIKATAS

package (Package).

FAKTAI

package(myApp).

package(myTask).

package(myDial).

package(taskWin).

package(win).

abstract(P,C,M,K,I) :- cycles(P,L,I), packageRef(P,C,M,K), member(K,L).

packageUsed2(P,S) :- package(P),
  findall(L,packageRef(L,_,_,P),L), list_to_set(L,S).

packageUsed3(P,A,E) :-package(P),
  findall(L, packageRef(L,_,_,P), L), length(L, A),
  findall(M, packageRef(P,_,_,M), M), length(M, E).

3 priedas. Metodų matomumo pertvarkų predikatai

Klasių apibrėžimai:

PREDIKATAS

class(Class, Assembly).

FAKTAI

class(a1, a).

class(a2, a).

class(b1, b).

class(b2, b).

class(a3, a).

class(c1, c).

Klasių ryšių apibrėžimas:

PREDIKATAS

classInUsed(Assembly, Class, InAssenbly, InClass).

FAKTAI

classInUsed(a, a1, a, a2).

classInUsed(a, a1, ab, b1).

classInUsed(b, b2, c, c1).

classInUsed(a, a2, b, b2).

classInUsed(b, b1, b, b2).

Klasių paveldėjimo ryšiai:

PREDIKATAI

classHierarchy(Assembly, Class, SubAssenbly, SubClass)

FAKTAI

classHierarchy(a, a1, a, a2).

classHierarchy(a, a2, c, c1).

Metodų apibrėžimai klasėse:

PREDIKATAS

methods(Assembly, Class, Method).

FAKTAI (pagal klases)

*methods(a, a1, aA1). /*private*/*

*methods(a, a1, aB1). /*internal*/*

*methods(a, a1, aC1). /*protected*/*

*methods(a, a1, aD1). /*protected internal*/*

*methods(a, a1, aE1). /*public*/*

*methods(a, a1, aF1). /*not used*/*

*methods(a, a2, aA2). /*private*/*

*methods(a, a2, aB2). /*internal*/*

*methods(a, a3, aA3). /*not used*/*

*methods(b, b1, bA1). /*internal*/*

*methods(b, b1, bB1). /*not used*/*

*methods(b, b2, bA2). /*private*/*

*methods(c, c1, cA1). /*public*/*

Metodų panaudojimo faktai:

PREDIKATAS

methodInUsed(Assembly, Class, Method, InAssembly, InClass).

FAKTAI

*/*private*/*

methodInUsed(a, a1, aA1, a, a1).

methodInUsed(b, b2, bA2, b, b2).

methodInUsed(a, a2, aA2, a, a2).

*/*internal*/*

methodInUsed(a, a1, aB1, a, a1).

methodInUsed(a, a1, aB1, a, a3).

*/*methodInUsed(a2, a, aB2, a2, a).*/*

methodInUsed(a, a2, aB2, a, a3).

methodInUsed(b, b1, bA1, b, b2).

*/*protected*/*

methodInUsed(a, a1, aC1, a, a1).

methodInUsed(a, a1, aC1, a, a2).

methodInUsed(a, a1, aC1, c, c1).

*/*protected internal*/*

methodInUsed(a, a1, aD1, a, a1).

methodInUsed(a, a1, aD1, a, a3).

methodInUsed(a, a1, aD1, c, c1).

*/*public*/*

methodInUsed(a, a1, aE1, a, a1).

methodInUsed(a, a1, aE1, a, a3).

methodInUsed(a, a1, aE1, c, c1).

methodInUsed(a, a1, aE1, b, b1).

methodInUsed(a, a1, aE1, b, b2).

methodInUsed(c, c1, cA1, b, b1).