ers nica" din Timişoara
Facultatea de Automatică şi Calculatoare
Departamentul Calculatoare
2, Vasile Pârvan Bv., 300223 – Timişoara, Romania
Tel: +40 256 403261, Fax: +40 256 403214
Web: http://www.cs.upt.ro

CLONEDETECTOR

PROGRAM PENTRU DEPISTAREA PROIECTELOR JAVA COPIATE

Proiect de diplomă

Adrian-Vlad LEP

Conducător:
Conf.dr.ing. Ioana ŞORA

Timişoara,
2011

Cuprins

7. Compararea a doua proiete

VI. Manualul utilizatorului

VII. Results

VIII. Concluzii

I. Introducere

1. Domeniul Temei

Înaintea "erei" calculatoarelor, copierea proiectelor presupunea efort din partea înfăptuitorului, acesta fiind nevoit să caute în cărţi şi să scrie manual un proiect chiar şi copiat, fără adaos de originalitate. Depistarea proiectelor copiate se realiza exclusiv de către o persoană fizică, prin analizarea tuturor candidaţilor.

Odată cu apariţia calculatorului şi transformarea informaţiilor în format electronic copierea a fost mult uşurată, determinând creşterea numărului de plagiaturi. Proiectele software şi cele şcolare au suferit şi ele această problemă, un factor suplimentar fiind şi apariţia proiectelor open source.

Detecţia clonelor devine astfel to mai grea cu ochiul liber, fiind nevoie de tool-uri special create pentru aceasta..

2. Problema

Problema încălcării drepturilor de autor a fost serios dezbătută şi luată în considerare. În directia proiectelor software au fost dezvoltate numeroase unelte şi încercate diverse abordări pentru depistarea plagiaturile. Rezultatele obţinute de aceste programe sunt încurajatoare dar mai sunt mulţi paşi de făcut în acesta direcţia până a putea spune că avem un tool ce poate depista cu o rată de succes acceptabilă proiect software copiate.

Problemele întâmpinate în găsirea unei soluţii generice sunt :

- *Cantitatea mare de informaţii*. Proiectele pot fi copiate între studenţii din aceeaşi an de studiu, de la studenţii din anul anterior, de acum doi ani s.a.m.d De asemenea ele pot fi găsite pe internet, diverse site-uri găzduind cod sursă pentru proiecte : sourceforge.net, code.google.com, etc.

- *Clonele sunt greu de depistat.* Compararea codului şi depistarea duplicarii pare o abordare a problemei, care în contextul unei cantităţi mari de informatiie devide o provocare de implementat. Totuşi această abordare este insuficientă, depistarea fiind imposibilă în cazul unor modificărilor (redenumiri de exemplu), modificări uşor de realizat cu uneltele actuale folosite în dezvoltare.

- *Greu de decis dacă sunt clone.* Depistarea unui grup de două sau mai multe proiecte software similare este doar prima etapă. Confirmarea presupunerilor şi stabilirea cu certitudine că acestea au fost copiate este o sarcină care se rezolvă prin inspecţie umană. Între un proiect copiat şi o metodă asemantoare de rezolvare de multe ori există o graniţă foarte greu de stabilit. Dacă cerinţă determină o abordare consacrată, apariţia asemenarilor între proiecte va fi evidentă şi deci depistarea a ceea ce este copiat sau nu devine foarte greu. De exemplu pentru implementarea căutării binare metoda este binecunoscută iar rezolvarea problemei nu va diferi foarte mult de la un proiect la altul

3. Obiective

Deoarece metoda tradiţională de compare manuală a proiectelor nu mai este fezabilă datorită cantităţii de informaţii ce trebuie analizate, pentru depistarea clonelor au fost elaborate, de diverse universităţi în special, programe pentru analizarea automată a proiectelor. Acestea sunt de un real ajutor pentru profesori şi nu numai. Aceste instrumente se diferenţiază între ele prin :

- *tipul de date analizat* : proiecte literare, proiecte software : Java, C, C# etc.
- *aria de căutare a duplicatelor* : proiectele dintr-o locaţie /dintr-un director, proiectele de pe un calculator sau căutarea pe internet a posibilei perechi.
- *tipul de clone depistate* : există diferite tipuri de clone pe care fiecare program le poate depista
- *dimensiunea maximă a datelor ce o pot analiza* : proiecte şcolare, proiecte (software sau nu) de dimensiuni mai mari, internetul.
- *algoritmul de depistare* : categorii de algoritmi vor fi descrişi în capitolul următor.

Platforma CloneDetector analizează static codul sursă şi depistează proiectele Java copiate pe baza unui set de reguli Prolog. CloneDetector foloseşte o abordare originală, combinând metodele de depistare bazate pe metrici şi cele bazate pe grafuri de apeluri (call graph) create din arbori sintactici abstracţi(ĂST). Prin acest instrument se doreşte depistarea clonelor de tipul 3, adică a proiectelor copiate în care pe lângă redenumire s-au operat schimbări de natură sintactică, s-au adăugat sau şters porţiuni de cod. Această platformă s-a dovedit un succes pe proiectele şcolare testate, având potenţialul de a deveni un instrumet larg folosit în depistarea copiilor.

Lucrarea de faţă descrie în detaliu algoritmul de detecţie folosit pentru depistarea clonelor în secţiunea IV. În secţiunea următoare vom introduce noţiunile teoretice folosite în elaborarea lucrării actuale. Secţiunea III vă prezentată modul de obţinere a ĂST-ului din codul sursă Java. Secţiunea V prezintă detalii de implementare a algorimului şi a metricilor folosite. Manualul de utilizare se găseşte în secţiunea VI iar rezultatele experimentale sunt descrise în secţiune VII.

II. Fundamente teoretice

1. Categorii de clone

În această secţiune ne propunem să definim şi să clasificăm clonele sau copiile software pe baza cercetărilor efectuate în acest domeniu.
Conform [11] plagiatul este reproducerea muncii altei persoane fără a recunoste provenienţă informaţiei. Cele mai întâlnite cazuri apar în mediul academic unde studenţii copiază materiale din cărţi, de pe internet sau de la colegi fără să citeze sursa lor de provenienţă. Plagiatul este o practică şi în alte domenii, un exemplu celebru fiind acela dintre Steven Spielberg şi Chase-Riboud care îl acuză pe primul că personaje, scene şi alte aspecte din cartea sa “Echo of Lion” au fost copiate în filmul Amistad.

Un alt exemplu, este procesul pornit între IBCOS Computers Ltd şi Barclays Mercantile Highland Finance Ltd. pentru copierea ilegală de software. Cazul era legat de un programator care s-a mutat de la o firmă la alta şi a refolosit codul dezvolatat la prima în produsele noii sale companii. Instanţă a decis că drepturile de copyright au fost încălcate.

Copierea proiectelor software presupune reutilizarea codului scris de altă persoană, în întregime sau parţial, încălcând astfel drepturile de autor. Conform [1] clonele software se împart în trei categorii în funcţie de natural lor :

Tipul 1 sunt clone identice, fără modificări (doar eventuale comentarii sau spaţii modificate).

Tipul 2 sunt copii identice din punct de vedere sintactic. Sunt posibile redenumiri de variabile, tipuri sau identificatori de funcţii.

Tipul 3 sunt clone care au suferit mai multe schimbări; prin modificarea, ştergerea sau adăugarea unor declaraţii, instrucţiuni.

După cum se observă tipul 1 şi 2 de clone sunt clar şi precis definite.Tipul 3 este mai vag definit repezentand o categorie mai mare de posibile clone. Din acest motiv unele unelte consideră unitar clonele de tipul 1 şi 2, în timp ce clonele de tipul 3 sunt tratate separat, datorită diferenţelor de abordare.

Clonele de tipul 1 şi 2 pot fi declarate automat că fiind copiate, dar pentru clonele 3 nu se pot formula metrici care să stabilească cu certitudine dacă sunt sau nu copiate. Astfel clonele de tipul 3 declarate suspecte trebuie analizate de către o persoană înainte de a fi pronunţate adevărate.

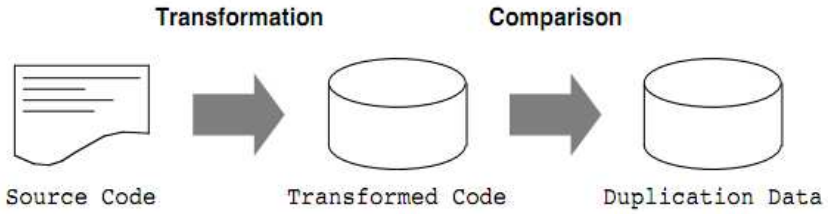
Există şi alte clasificări ale clonelor oferite de Balazinska în [12] sau de Kasper în [13], [14] , acestea fiind considerate cele mai detaliate clasificări actuale. Totuşi în această lucrare am adoptat clasificarea prezentată mai sus, fiind simplă, uşor de inţeles şi suficient de sugestivă.

În urmă analizei proiectelor, instrumentul creează o listă de posibili suspecţi. Aceasta poate fi completă, în cazul în care toate clonele adevărate apar în listă sau incompletă dacă unele lipsesc. Din aceast raport unele clone vor fi confirmate în timp ce altele se vor dovedi presupuneri false.

Un program eficient, se doreşte să raporteze toate sau cât mai multe clone ce sunt adevărate (să nu scape nimic) şi în acelaşi timp să raporteze cât mai puţine clone false. Un instrument ce raportează prea multe clone false este nefolositor, sporind cantitea de informaţie ce trebuie analizată.

2. Categorii de Algoritmi

Pentru depistarea proiectelor software copiate au fost abordate mai multe tehnici. Aceste se diferenţiază prin tehnică de comparare(algortmul) şi nivelul la care este analizată informaţia. Figura 2.1 este sugestivă, prezentând câteva tehnici şi caracteristicile lor.



Author	Level	Transformed Code	Comparison Technique
Johnson 94	Lexical	Substrings	String-Matching
Ducasse 99	Lexical	Normalized Strings	String-Matching
Baker 95	Syntactical	Parameterized Strings	String-Matching
Mayrand 96	Syntactical	Metrics Tuples	Discrete comparison
Kontogiannis 97	Syntactical	Metrics Tuples	Euclidean distance
Baxter 98	Syntactical	AST	Tree-Matching

Fig.2.1 [15] Tipuri de algorimi

Vom prezenta succint câteva dintre ele oferind exemple de programe ce le implementează.

- **Comparare de şiruri**

Presupune interpretarea liniilor de cod ca nişte şiruri de caractere şi compararea lor pe baza diverşilor algoritmilor. Ea se împarte la rândul ei în două categorii :

Comparare textuală : Compară linii întregi de cod. Pentru a creşte performanta sunt calculate funcţii hash ale liniilor. Această abordare a fost introdusă de Ducasse în [2]

Comparare simbolică : Compararea este la nivel de simboluri, o linie de cod având de obicei mai multe simboluri. În locul unei analize a şirurilor de caractere se creaza un arbore de sufixuri şi se compară pe baza de functori. Propusă de Baker în ’95 [3].

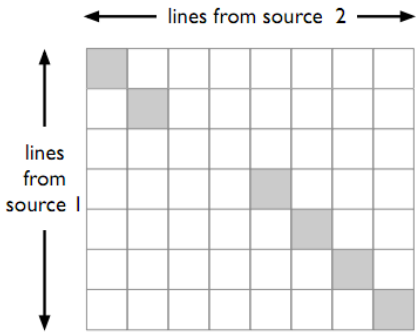


Fig. 2.2 [15] Comparare textuala

Practic în programele ce adoptă acesta variantă vor compară linii de cod dintr-un fişier al unui proiect cu cele din alt fişier dintr-un proiect diferit. Dacă găsesc o portine exact la fel ea este memorată ca fiind copiată, în figura de mai sus,Fig.2.2, asta realizându-se prin colorarea gri a pătratului corespunzător. Dacă proiectele ar fi identice în totalitate ar trebui să obţinem diagonală principală a matricei în totaliate gri, iar restul alb. Cum acest caz este mai rar întâlnit matricea poate arată ca în figura 2.3. Se obserca că există porţiuni din primul fişier care nu se regăsesc în al doilea , primul spaţiu lipsă pe diagonala principală. Se poate datora modificării codului. De asemenea există protiuni care se vor găsi decalat în al doilea fişier, cum este cazul celui de-al doiela spaţiu de pe diagonala principală. Acesta decalare poate apărea datorită adăugării unei noi porţiuni de cod.

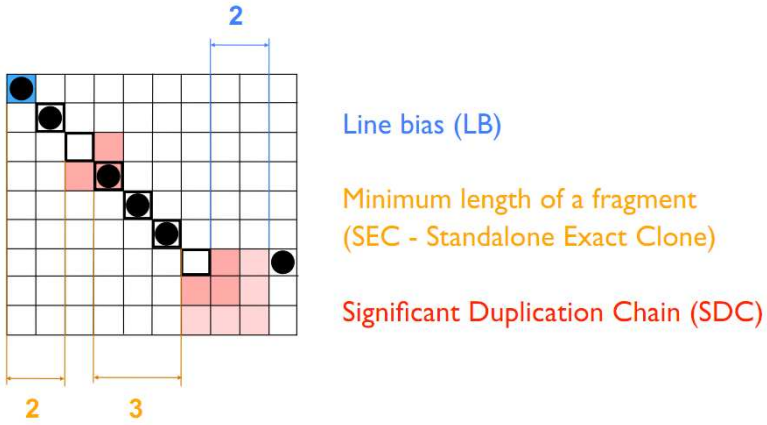


Fig. 2.3 [15] Interpretarea rezultatelor

Pentru a declara o porţiune de cod suspectă de copiat, întregul fişier sau proiect, trebuie să existe o porţiune de cod copiat de lungime minimă şi cu un decalaj maxim între bucăţi.
Pe baza acestor metode au fost dezvolate numeroase platforme pentru depistarea copiilor software dar şi pentru depistarea celor literarea sau a duplicarii de cod. Câteva exemple ar fi :

- *MOSS(Measure Of Software Similarity)* Este unul din cele mai cunoscute unelte în domeniu, dezvolatat de Universitatea Stanford în 1994. Este folosit şi astăzi mai ales pentru depistarea temelor software copiate. Programul poate analiză codul sursă pentru următoarele limbaje de programare : C, C++, Java, C#, Python, Visual Basic, Javascript, FORTRAN, ML, Haskell, Lisp, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, MIPS assembly, a8086 assembly, a8086 assembly, MIPS assembly, HCL2. Este disponibil ca serviciu web pe http://theory.stanford.edu/~aiken/moss/.
- *JPlag Este* o platformă dezvoltată de Universitatea din Karlsruhe. Se bazează pe algorimul de compărea de şiruri cunoscut sub numele :Karp-Rabin Greedy String Tiling. Oferă suport pentru limbajele Java, C#, C şi C++. Este disponibil la adresa dată ca servciu web https://www.ipd.uni-karlsruhe.de/jplag/
- *YAP* : Dezvolatat de universitatea din Sydnei, are trei versiuni schimbând în fiecare dintre ele algorimul de comparare. Acum folosesc acelaşi algoritm ca şi JPlag, trecând de la vechiul algoritm Heckel folosit în versiunea 2. Este folosită ca serviciu web http://luggage.bcs.uwa.edu.au/~michaelw/YAP.html

După cum observăm principalii contribuitor în acest domeniu au fost universităţile, ele incercad să găsească soluţii la problema copiatului. Majoritatea au adoptat varianta serviciilor web prin care orice profesor se poate înscrie şi trimite un set de teme, urmând să primească rezultatul. Această metodă este sigură, studenţii fiind împiedicaţi să încerce să păcălească

unealta. Totuşi pentru noi a fost dificil să obţinem acces pentru a compara metoda noastră cu aceste abordări şi am ales ca tool de comparare din acesta categorie DUDE, dezvoltat de Radu Marinescu [10].

- **Compararea pe baza de metrici**

Presupune obţinerea unor metrici ce caracterizează o bucată de cod. Se vor compara vectorii de metrici obţinuţi pentru diverse porţiuni şi vor fi considerare egali cele ce au sub o anumite distanţă admisă. Diferite metrici au fost propuse de către : [3],[4] şi [5],[17],[18].

- Halstead :
  - Numărul de operatori distincti
  - Numărul de operanzi distincti
  - Numărul total de referiri(folosiri) ale operatorilor
  - Numărul total de referiri ale operanzilor.
- McCabe
  - Complexitatea ciclomatică
- Dunsmore
  - Adâncimea ierarhiilor de clase dintr-un modul
- Kolmogorov complexity

Exemple de sisteme ce implementează acest tip de algoritmi :

- SID descris în detaliu în [19] foloseste metrica Kolmogorov.
- Platforma de depistat programe copiate dezvolatata de Ottenstein [33] de la Universitatea Purdue, bazata pe metricile lui Halstead

- **Compararea arborelui sintactic abstract (AST)**

Această abordare se bazează pe obţinerea arborelui sintactic abstract al programului, în prima fază, iar pe baza acestuia analizându-se similaritatea codului. Programul e partitionat în bucăţi de AST care sunt comparaţi prin diverşi algoritmi.

Analiza AST-ului obţinut poate fi realizată în mai multe feluri. O variantă este *compararea grafului de apeluri a programului*. Dependinţele unei funcţii fată de date sau alte funcţii pot fi reprezentate printr-un graf. Clonele ar trebui să aibe subarbori identici. Diferiţi algoritmi au fost propuşi şi implementaţi în [6], [7] şi [8].

Code Digger este un exemplu de program dezvoltat bazându-se pe această abordare de către Peter Bulychev şi Marius Minea. Compararea aborilor de bazează pe algoritmul anti-unification.

Această categorie de algoritmi sunt mult mai exacţi ei fiind capabili să depisteze şi bucăţi de cod copiate care au suferit modificări mai avansate. Pe de altă parte, aceşti algoritmi fiind mai complexi au un timp de rulare mai mare, fiind greu de folosit pe proiecte software mai mari.

- **Alţi algoritmi**

Pe lângă aceste tipuri clasice de abordări au apărut programe ce folosesc algoritmi de căutare pe internet, de explorarea datelor şi alţii.

- Siff compara fisiere mari similare intr-un sistem. Se bazeaza pe apoximarea fisierului printr-un fingerprint, un sir de caractere scurt specific fiecarui fisier. Se calculeaza checksumul fiecarui fingerprint, iar acestea vor fi apoi rapid comparate intre ele.

- SCSSB (System Call Short Sequence Birthmark) si IDSCSB (Input Dependant System

Call Subsequence Birthmark) descris în [16] propun o metoda de detectare a duplicarii de cod în proiecte mari software bazata pe aplerurile de sistem. Aceasta abordare nu se preteaza sistemelor software mici, dar la sisteme mari sucesiunea de apeluri catre kernel-ul sistemului de operare sunt o metrica ce caracterizeaza sistemul sau portiuni ale lui în intregime.

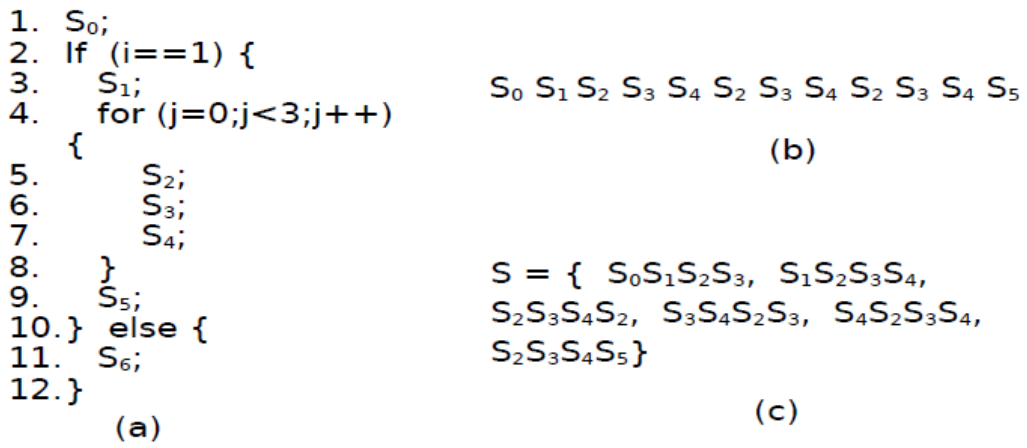


Fig.2.4 a)Exemplu de cod b) Apelurile catre sistem generate pentru intrarea i=1 c) Lista de apeluri generate pentru i =4

În figura de mai sus avem un scurt exemplu pentru a ilustra această tehnică. Se observă astfel lista de apeluri ce se va obţine executând codul dat pentru diferitele valori de intrarea ale lui i. Prin compararea acestor liste se pot depista porţiuni de cod clonate. Această comparare este una dinamică, deorece pentru a obţine lista apelurilor către sistem proiectele analizate vor fi executate.

Întrucât plagiatul proiectelor software şi duplicarea de cod sunt similare ca metode de detecţie, diferenţă fiind făcută de baza de informaţii ce e comaprata, algoritmii descrişi pot fi folosiţi pentru a atince oricare din cele două scopuri.

Observăm existenţa numeroaselor abordări pentru depistarea clonelor software. Eficientă lor fiind analizată în funcţie de timpul de execuţie şi de acurateţea cu care depistează anumite tipuri de clone. Inurma analizării unui set de platforme cunoscute s-a realizat că fiecare are avantaje pentru unele categorii de clone. Din punct de vedere al rapidităţii, uneltele ce implemeteaza compararei bazate pe şiruri de caractere sau simboluri sunt mult mai rapide decât cele ce analizează arbori sintactici dar acurateţea lor fiind mai redusă. O descriere mai detaliată a avantajelor si dezavantajelor fiecărei platforme se poate găsi în publicaţiile [22] şi [23] care revizuiesc o mare parte a uneltelor prezentate.

III. JTransformer

Pentru analizarea proiectelor suspecte de copiat suntem nevoiţi să transformăm codului sursă, scris în Java, într-un arbore sintactic, reprezentabil în Prolog. Această transformare a fost realizată folosind JTransformer 3.0.1.

JTransformer este un motor de interogare şi de transformare a codului sursa Java, disponibil ca plugin pentru Eclipse. Cu acesta am creat Arborele Sintactic Abstract (AST) pentru un proiect Java, luând în considerare codul sursa complet, tipurile de fişiere şi membrii clasei. AST-ul Java este reprezentat intern ca o bază de date Prolog, facilitând analize puternice şi consturirea unor reprezentări abstracte ale proiectului într-un mod uşor, în doar câteva linii de cod.

JTransformer are un nivel detaliat de interpretare a codului Java, luând în considerare atribute, metode, instrucţiuni, chiar şi commenturi pentru clasele din proiect. Pentru clasele din librării nu sunt luate în considerare detaliile de interpreatere. Se reprezintă în baza de date prolog doar interfaţa claselor.

Pentru fiecare nod din AST se creează o axiomă(faptă). Aceste axiome sunt legate între ele prin ID-uri unice. De exemplu, pentru o clasa se creează o axiomă classT. Lista complete de axiome posibile se găseşte la [9].

În continuare vom lua în considerare un exemplu pentru a vedea modul în care este transformat codul Java într-o baza de cunoştinţe Plolog.

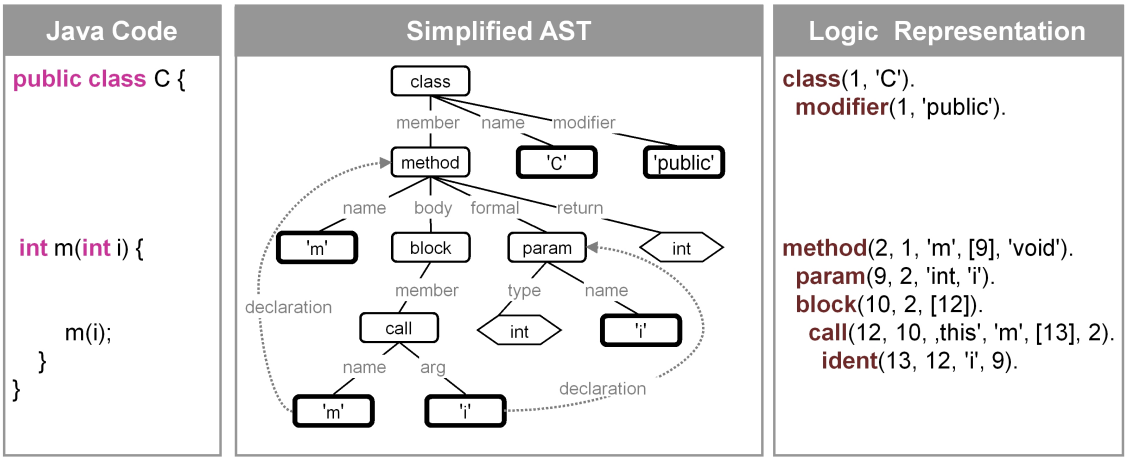


Fig. 3.1 [9] Exemplu de transformare

În imaginea de mai sus avem în partea stângă codul sursă Java, iar în partea dreaptă reprezentarea Prolog creată. Se observă că pentru definirea clasei se creează două intrări în baza de date, creindu-se o intrare separată pentru modificatorul claselor. De asemenea metoda Java e reprezentată în prolog prin axiome de tipul method, param – care conţin parametrii metodei şi block – corpul metodei. Block va fi părinte pentru celelalte instrucţiuni din interiorul metodei. Ierahia stabilită între axiome, prin identificatori, este prezentată în schema din mijlocul imaginii. Transformarea prezentată în Figura 3.1 este simplificată comparativ cu cea generată de JTransformer. Inea nu sunt incluse identificatori care fac legătura din spre părinţi sper copii.

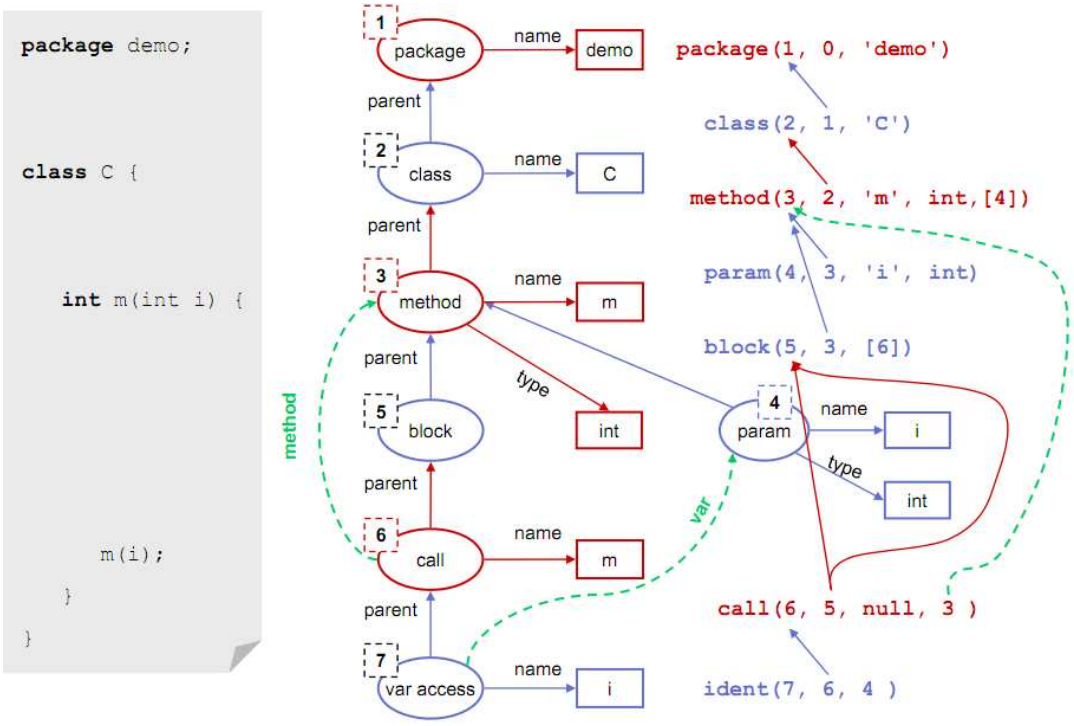


Fig. 3.2.[9] Relatiile intre clauze

Figura 3.2. prezinta relatiile din interiorul bazei de date Prolog, modul în care un identificator leaga două clauze intre ele. O clauză poate fi legată de mai multe clauze în mod direct sau prin redirectari.

In reprezentarea reală Prolog generată de plugin vor fi mai multe detalii. Încontinuare sunt exemplificate cateva clauze :

classT(#id, #parent, 'name', [#def_1,...])
methodT(#id, #classT, 'name', [#param_1,...], TYPE, [#exception_1,...], #body)
paramT(#id, #parent, TYPE, 'name')
blockT(#id, #parent, #enclMethod, [#statement_1,...])
callT(#id, #parent, #encl, #expr, 'name', [#arg_1,...], #method)
identT(#id, #parent, #encl, 'name', #symbol)

Întotdeauna primul număr este identificatorul unic. Al doilea parametru este identificatorul părintelui, în cazul metodei va fi id-ul clasei de exemplu. Aceştia pot fi urmaţi de identificatori ce relaţionează clauzele sau de parametrii caracteristi axiomei curente. Parametrii caracteristii fiecărei axiome sunt numele, de exemplu. După cum putem observa un termen al unei clauzei poate fi o listă, cum avem la metoda lista de parametrii de exemplu.
Termenul #encl trimite către metoda din care face, deci clauzele pot referii pe lângă părinţii direcţi şi părinţii părinţilor ei, mergând până la nivel de clase . Mai multe detalii despre modul de reprezentare al informaţii vor fi prezentate şi în capitole ulterioare în momentul în care vom fi nevoiţi.

JTransformer a fost alegerea protrivita pentru noi, datorită capacităţii sale de a genera o reprezentare Prolog detaliată din codul sursă Java. De asemenea, platforma este uşor de folosit datorită integrării cu eclipse, cel mai popular mediu de dezvoltare în Java. Viteza de transformare este impresionantă, fiind capabilă să transforme proiecte de dimensiuni mare, chiar de 1,000,000 LOC în câteva secunde..

IV. Descrierea aplicatiei

În secţiunea următoare vom descrie platforma propusă de noi, discutând arhitectura aplicaţiei, algoritmul de detecţie folosit şi interfaţa cu utilizatorul.

CloneDetector a fost dezvoltat pentru a depista proiecte şcolare Java. El a fost implementat în Prolog, un limbaj folosit cu precădere în domeniu inteligentei artificiale. Paradigma de programare din care face parte şi Prolog face uşor de reprezent probleme din viaţa reală prin fapte şi reguli de legătură. Sunt diferenţe mari de implementare între Prolog şi limbaje procedurale sau orientate pe obiecte, datorate în mare parte maşinii de inferenţă. Aceasta implementează algoritmul backtracking încercând să genereze toate soluţiile posibile pentru o anumită regulă. Pentru noi această abordare a fost un avantaj putând reprezenta uşor şi intuitiv regulile algoritmului şi beneficiind apoi de maşină de inferentă care caută soluţiile pentru regulile descrise.

Compilatorul folosit a fost SWI-Prolog acesta fiind probabil cel mai complet de pe piaţă, oferind un mediu de depanare grafic foarte folositor şi o gama largă de librării.SWI-Prolog suportă execuţia multi-threading o facilitate importantă care a îmbunătăţit timpul de execuţie al aplicaţie. Este o platformă bine menţinută ce suportă toate marile tipuri de sisteme de operare (Windows, Linux, Mac OSX) pe arhitecturi de 32 de biţi câţ şi pe 64.

1. Arhitectura aplicatiei

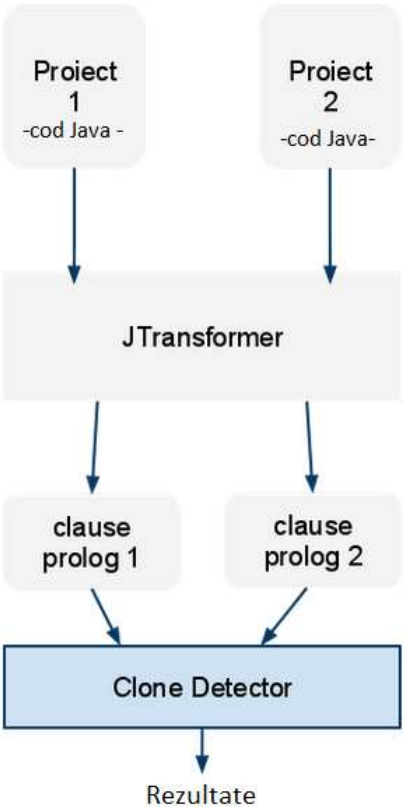


Fig. 4.1 Arhitectura sistemului

Ţinând cond că sursele proiectelor analizate sunt scrise în Java, iar noi dorim o reprezentare Prolog a acestora am folosit plugin-ul eclipse JTransformer pentru a transformă aceast cod într-o reprezentare Prolog. Codul va fi transformat în fapte care prin relaţiile dintre ele alcătuiesc un arbore sintactic abstract. Ca ieşire, JTransformer generează un fişier pentru fiecare proiect cu reprezentare proiectului. Pentru a îmbunătăţii performanţele am ales ca acesta să genereze un fişier gata compilat, cu extensia qlf (quick load file). Încărcarea acestor fişiere fiind de aproximativ 50 de ori mai rapidă.

În Fig.4.1 avem o descriere generală a sistemului. Pentru compararea a două proiecte, din codul Java se va genera cu ajutorul plugin-ului Eclipse JTransformer arborele sintactic abstract (AST) al fiecărui proiect. AST-ul va fi format din clauze Prolog şi în fişiere compilate Prolog.

CloneDetector va încărca în baza de cunoştinţe aceste fişiere şi pe baza algoritmului descris în continuare va analiza dacă proiectele au fost copiate.

Întrucât compararea manuală a proiectelor două cate două ar fi fost ineficientă, consumând mult timp din partea profesorului, s-a încercat automatizarea acestei sarcini. Programul suportă posibilitatea de a primi ca parametru un director, comparând toate proiectele, două cate două, găsite în acesta.

2. *Principiul algoritmul de comparare*

Algoritmul de comparare propus de acesta lucrare este unul original combinând abordarea bazată pe metrici cu cea bazată pe grafuri de apeluri. Metricile descriu porţiuni de cod, iar combinarea lor pot diferenţia unic aceste portiniuni, ca un fel de amprentă a codului. Grafurile de apeluri sunt obţinute din AST, luând în calcul apelurile realizate de o metodă către alte metode. Obiectivul nostru era de a obţine un tool capabil să depisteze clone de tipul 1, 2 dar mai ales cele de tipul 3, imun la redenumiri, cu un timp de procesare acceptabil. Cele două abordări au avantaje şi dezvantaje iar noi am încercat să luăm avantajele fiecăruia şi să minimizam dezavantajele. Metricile sunt mai rapid de calculat, dar sunt inexacte, fiind greu să descrii o porţiune de cod doar prin metrici, în timp ce arborii de apeluri sunt exacţi dar costisitori din punct de vedere computaţional.

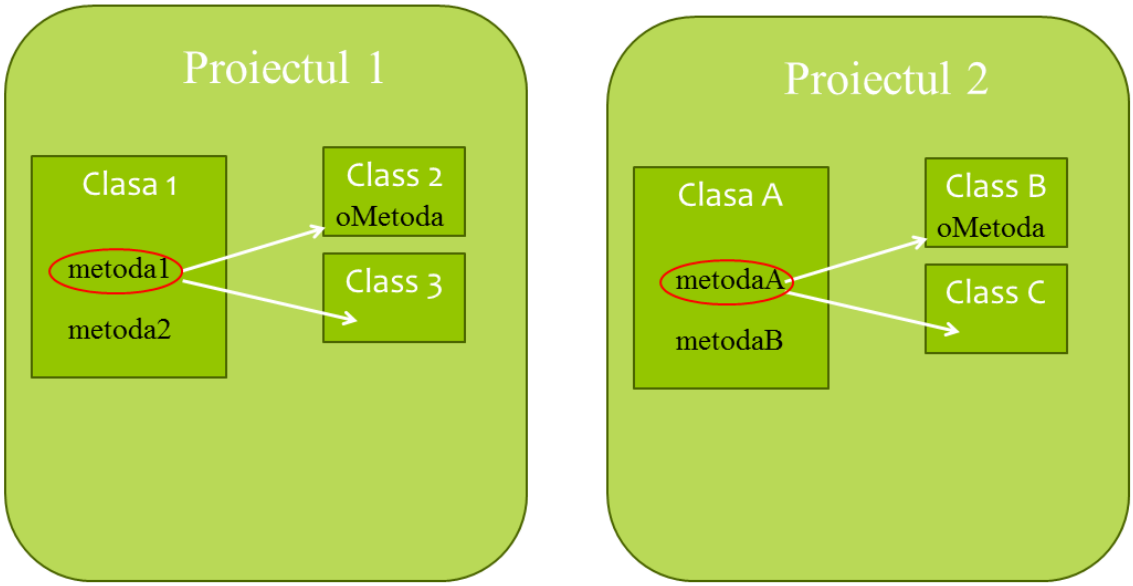


Fig. 4.2 Descrierea algoritmului

În continuare vom explica paşii algoritmului folosindu-ne de figura 4.2 pentru exemplificare. Pentru compararea a două proiecte instrumentul începe să genereze toate combinaţiile posibile dintre clasele primului proiect şi cel de al doilea. Analiza doreşte să stabilească la final care clasă este identică cu care, memorând perechile de clase copiate şi probabilitatea lor de a fi copiate.

Două clase vor fi considerate identice dacă vor avea similaritati la nivel de clasă, dacă metodele lor vor fi similare şi dacă clasele care vor fi apelate din metodele celor două clase vor fi asemănătoare. Deci clasele 1 respectiv A din figura vor fi considerate identice dacă în prima etapă metricile la nivel de clasă sunt similare. Urmează compararea la nivel de metodă unde încercăm să împerechem metodele similare din cele 2 clase. Metoda 1 va fi similară cu metoda 2 dacă metricile la nivelul celor două metode sunt apropiate, după care se analizează la nivel de apeluri. Cele două metode vor fi similare dacă metoda oMetoda din Class2 va fi similară cu metoda oMetoda din ClassB şi dacă Class2 va fi similară cu ClassB. La al doilea nivel, compararea claselor şi metodelor apelate(Class2, ClassB, oMetoda), se analizează doar metricile clasei şi a metodei.

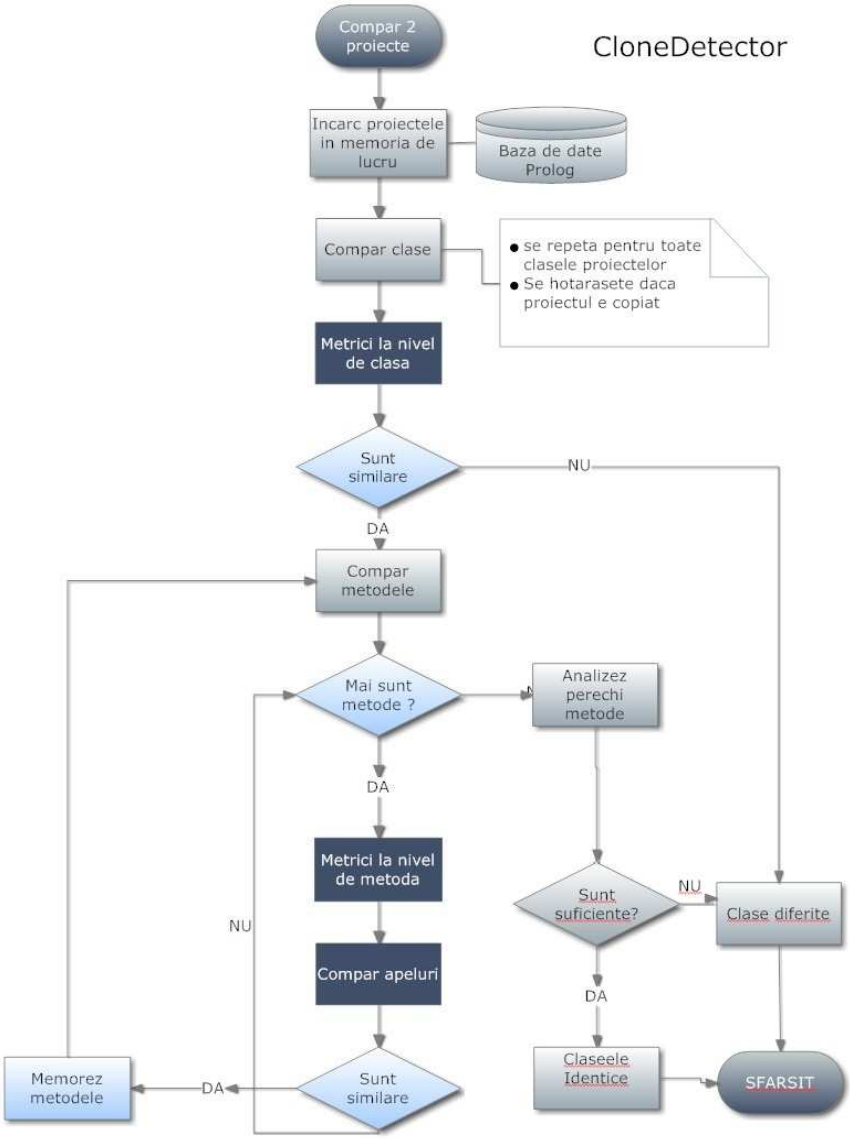


Fig. 4.3 Reprezentare schematica algoritm

O abordare recursivă la nivel de apeluril (luare în calcul şi a apelurilor pentru metodele aplelate) ar conduce la o aprofundare a anlizei şi deci creşterii duratei de timp. S-ar genera de asemenea numeroase cazuri excepţionale, apeluri ciclice, problema moştenirii etc, pentru tratarea cărora ar trebui complicat algoritmul. O astfel de abordare ar atrage atât avantajele dar mai ales dezvantajul algorimilor bazaţi pe grafuri de apeluri, adică timpul lung de procesare. Analiza doar a primului nivel a grafului a fost suficientă. Combinând-o cu metricile ea obţinând rezultate bune.

După analiza primului apel a metodei 1 din cele două clase se vor analiza şi restul apelurilor către clasele 3 respectiv C, după care se va trece la următoarea metodă.

Practic, analiza claselor trece printr-un set de filtre. Trecerea de un filtru presupune o probabilitate mai mare ca cele două clase să fie copiate. Trecerea filtrelor la nivel de clasă determină asignarea acestora ca fiind suspecte puţin probabile de copiat. Dacă filtrele la nivel de metodă sunt satisfăcute, clasele devin suspecte cu probabilitate medie de a fi copiate, iar trecerea de filtrele comportamentale vor determină clasele să fie considerate suspecte foarte probabile.

O reprezentare schematică a ceea ce am exemplificat mai sus este realizată în figura 4.3. Metricile considerate reprezentative şi alese pentru compararea la nivel de clasă şi metodă sunt enumerate în continuare, implementarea lor fiind arată în capitolul următor.

A.Metrici la nivel de clasă :

1. numărul de atribute
2. numărul de metode
3. numărul de interfete implementate
4. moşteneşte sau nu o superclasa.
5. numărul de metode similare

B.Metrici la nivel de metodă :

1. numărul de cicluri while, do while, for
2. numărul de conditii if, if else
3. numărul de operatori -, *, /
4. modificatorul static
5. semnătura metodei

3. *Justificarea metricilor alese*

Înainte de a ne hotăra asupra acestor metrici am testat pe un set de proiecte dacă acestea sunt suficiente pentru a descrie unic clase sau metode. Doream ca prin această cercetare să vedem dacă proiectele diferite au metrici diferite, dacă cele similare au metrici asemănătoare şi care este diferenţă între metricile proiectelor copiate sau nu. Imaginea 4.4 prezintă o bucată din studiu, acesta fiind disponibil şi online pe docs.google.

Nume proiect	Nume clasa	nr atribute	nr metode	nr interf	nr superclase	Nume metoda	nr op (+, - ,*, /)	cicluri while/for	if	nr param	ret val
passc - xml	XMLReader	6	7	1	1	startDocument	(0,0,0,0)	0		0	void
						endDocument	(0,0,0,0)	0		0	void
						startElement	(5,0,0,0)	0	1	4	void
						endElement	(1,0,0,0)	0	0	3	void
						characters	(1,0,0,0)	0	1	3	void
						storeList	(0,0,0,0)	0	0	1	void
						loadList	(3,0,0,0)	1	1	0	List
	Student	3	6	0	0	Student/constuctor	(0,0,0,0)	0	0	3	
						getNrMatricol	(0,0,0,0)	0	0	0	int
						getMedie	(0,0,0,0)	0	0	0	double
						toString	(3,0,0,0)	0	0	0	String
						cresteMedia	2,0,0,0	1	1	0	bool
						vaFiZiBuna	1,0,0,0	0	1	0	bool
	Main	0	1	0	0	main	(6,0,0,0)	0	0	1	void
	Horoscop	2	7	0	0	constructor	(0,0,0,0)	0	0	1	
						changeSource	(0,0,0,0)	0	0	1	void
						loadList	(0,0,0,0)	0	0	0	void
						printList	(0,0,0,0)	0	0	0	void
						findByNr_matr	(0,0,0,0)	1	0	1	student
						findByMedie(double)	(0,0,0,0)	1	0	1	List<Student>
						catiVorAvealMediaScaz	(0,0,0,0)	1	0	0	int
<<interfata>>	DataAccessObject	0	2	0	0	loadList					List
						storeList					void
webserver	ServerGUI	4	2	0	1	cosntructor					
	BrowseListener	6	8	0	0						
Passc 1	Boala	3	5	0	0						
	FisaPacient	4	6	0	0						
	Main	0	1	0	0						
	Medic	3	5	0	0						
	Pacient	1	2	0	0						
	Simptom	1	3	0	0						
	TraseuPacient	2	4	0	1						

Fig. 4.4 Studiu realizat pe un set de proiecte

Studiu a relevat că încă de la nivel de clasă există puţine coliziuni, adică puţine clase care au aceleaşi metrici, acestea fiind de obicei cele de pe acelaşi nivel al unei ierarhii de moştenire. Totuşi comparând la nivel de metoda clasele vor fi diferite total pentru proiecte diferite. Şi atunci vine întrebarea de ce nu am adoptat varianta doar cu metrici fără graf de apeluri ? Proiectele ar fi copiate dacă metricile ar fi identice. Problema ce ne-a determinat să combinăm abordările este dată de comportamentul studenţilor care copiază. Aceştia au tendinţă să redenumească, lucru care nu ne afecta, dar câteodată mai schimbă sau adaugă cod creând clone de tipul 3. Acestea nu ar putea fi depistate dacă metricile ar fi comparate identic, de aceea noi vom compară asemănarea metricilor nu egalitatea lor, acceptând unele diferenţe.

Dacă ne uităm în tabelul din figura 4.4 observăm că acceptând aceste diferenţe la nivel de clasă şi metodă vom obţine coliziuni, astfel analiza apelurilor fiind justificată.

Pe lângă aceste metrici, enumerate mai sus, care sunt calculate direct din reprezentarea AST a proiectelor, sunt setate filtre care iau în considerere, la nivel de clasă, numărul metodelor ce au fost grupate şi nivelul lor de asemenare. Pe baza acestor filtre asociem nivelul de asemenarea între doua clase.

4. Compararea metricilor

Compararea valorilor obţinute pentru metrici nu se face exact, după cum am menţionat anterior, adică numărul de if-uri dintr-o metodă nu va trebui să fie exact acelaşi şi în cealaltă. Se acceptă aceste diferenţe tocmai pentru a permite prinderea plagiaturilor schimbate prin adăugarea de noi condiţii sau alte modificări. Am adoptat două variante de a defini aceste tolerante, procentual sau prin diferenţă absolută.

Procentual presupune calcularea procentului dintre două metrici, de exemplu 80% între numărul metodelor din două clase reprezintă că prima clasă are 80% din numărul metodei celei de a doua, fără a ştii dacă sunt sau nu egale. Inacest caz am putea stabili că acest procent, 80% să fie minimul acceptat pentru a continuă analiza a două clase. Astfel dacă o clasă are doar 50% ca număr de metode fată de alta, nu se va încercă să se grupeze cele două. Această abordare procentuală este avantajoasă pentru proiecte, clase sau metode mai mari. Pentru clase mici procentele au variaţii mari. Între o clasă cu 1 metodă şi alta cu 2 metode avem 50% procentul, la fel ca între una cu 10 metode şi alta cu 20. In primul caz sunt şanse ca acele clase să fie copiate, în vreme ce în cel de-al doilea putem spune aproape cu certitudine ca cele două clase sunt total diferite.

Diferenţa absolută între metrici. Inacest caz calculăm diferenţa absolută între metrici şi impunem ca aceasta să fie mai mică ca o valoare maximă tolerată. Astfel dacă avem 2 metode respeciv 7 în alta clasă diferenţa (delta) între ele va fi 5. O astfel de diferenţa o considerăm prea mare pentru a continua analiza claselor, dacă valoarea maximă adminsa ar fi 3. Această metodă de a exprima tolerantele este bună pentru proiecte mici, în care valorile metricilor variază puţin, dar penru proiecte mari ar putea fi problematică. Dacă avem 20 şi 24 de metode în două clase ar trebui să le comparăm, cele 4 metode putând fi obţinute prin spargerea unor metode mari.

CloneDetetor foloseşte cu precădere cea de-a două metodă de a defini tolerantele între două metrici, datorită proiectelor mici analizate, dar în anumite cazuri am considerat utilă şi prima metodă, cea procentual, înlănţuind condiţii procentuale sau diferenţiale prin şi/sau logic.

Alegerea acestor diferenţe maxime/procente minime a fost externalizata în fişierul profiles.pl. Am creat aici două profile unul loose şi altut tight, cu tolerante diferite. Se poate alege între cele două în funcţie de grupul de proiecte analizate. Profilul loose este mai permisiv, permiţând variaţii mai mari între metrici. Este ideal pentru proiecte a căror rezolvări pot diferi mai mult, nefiind sugerate anumite căi de la început. Profilul tight este folositor pentru teme a căror rezolvare este clasică, cunoscută sau a fost sugerată. Dacă am rula pe acelaşi set de teme profilul loose ar genera mai mulţi suspecţi, în timp ce profilul tight ar genera mai puţini dar cu şanse mai mari de a fi copiate.

5. Stabilirea probabilitatii de copiere

Pe baza tolerantelor setate în fişierul de configurare, se vor lua în continuare decizii legate de nivelul de similitudine între anumite componente (metode, clase) acestea putând fi : diferite sau cu anumite grade de asemănare.

Metodele pot avea două grade de similitudine : *High* sau *Low*.

Metodele grupate cu grad de asemănare *Low* sunt metode care sunt considerate suspecte de copiat, dar cu o probabilitate mai mică. Ele sunt declarate astfel dacă trec de filtrele la nivel de metoda dar nu trec de compararea grafurilor de apeluri.

Metodele grupate cu grad de asemănare *High* sunt metode care sunt considerate suspecte de copiat cu o probabilitate foarte mare. Ele au trecut atat de filtrele la nivel de metodă cât şi de cele la nivel de grafuri de apeluri.

Clasele au trei grade de asemănare : *Low, Medium, High*

Clasele suspecte Low sunt clasele care trec de filtrele la nivel de clasă, dar nu şi de cele la nivel de metodă. Sunt puţin probabile să fie copiate, numărul de clase asemănătoare la acest nivel putând fi mare. Nu au o pondere important în luarea deciziei dacă două proiecte trebuie afişate sau nu ca suspecte de copiat.

Clasele suspecte Medium sunt clase care trece de filtrele la nivel de clasă şi care au metode asemănătoare. Totuşi numărul de metode ce sunt foarte probabil copiate (grupate High) nu este suficient de mare pentru ca aceste să fie declarate foarte suspecte. Acest număr se setează în fişierul de profile. Putem considera că sunt clase ce trec de filtru la nivel de clasă şi cel la nivel de metodă, dar nu îndeplinesc condiţiile la nivel de graf de apeluri.

Clasele suspecte High sunt clase foarte probabil copiate. Ele au trecut de filtrele la nivel de clasă, iar metodel lor sunt asemănătoare atât la nivel de metrici cât şi la nivel de graf de apeluri. Existentă unor astfel de grupuri de clase între două proiecte este luat în considerare, acele proiecte putând fi copiate.

O facilitate oferită de CloneDetector este aceea de a exclude unele clase, pachete sau foldere din analiza. Aceasta se dovedeşte foarte utilă în cazul în care toate proiectele au porţiuni de cod date de la început. De exemplu, pentru unele teme de laborator se oferă un set de clase utilitare de la care să se pornească. Acestea ar fi cu siguranţă în toate proiectele şi tool-ul le-ar grupa. Dacă numărul lor ar fi semnificativ ele ar influenţa verdictul, multe proiecte devenind astfel suspecte datorită acestor clase. Nu dorim acest lucru, deorece s-ar pierde mult timp pentru analiza suspecţilor neadevăraţi. Pentru excluderea claselor am creat un fişier commonClasses.pl în care putem scrie clasele ce sunt comune între toate proiectele şi pe care dorim să le excludem din analiză.

V. Implementarea programului

Primul lucru pe care trebuie să îl facem este să încărcăm cele două proiecte în baza de cunoştinţe Prolog. Ele vor fi salvate pe disc în format qlf. Încărcarea lor în Prolog se realizează cu comanda consult care compilieaza fişierele .pl iar pe cele qlf le încarcă direct.

Totuşi simplă încărcare în baza de date generează o problemă, coliziuni între identificatori. După cum am menţionat anterior fiecare faptă are ca prim termen un identificator unic în cadrul unui proiect. Aceştia încep de la 10001 şi cresc în funcţie de dimensiunile proiectului. Totuşi aceşti identificatori nu vor fi unici dacă unim două baze de cunoştinţe în acelaşi spaţiu. Se vor suprapune identificatorii şi ne va fi imposibil să deducem care din ce proiect face parte. De aceea încărcarea simplă în baza de cunoştinţe, cu consult nu este o soluţie.

Soluţia găsită presupune folosirea modulelor din Prolog.

1. Module

Un modul Prolog este o colecţie de predicate care definesc o interfaţă publică

prin intermediul unui set de predicate furnizate şi a operatorilor. Module Prolog sunt definite de un standard ISO. InSWI-Prolog sistemul e modul este derivat din sistemul de Quintus modulul Prolog.

În sistemele clasice Prolog, toate predicatele sunt organizate într-un singur namespace şi orice predicat poate apela orice predicat. Deoarece fiecare predicat într-un fişier poate fi apelat de oriunde în program, devine foarte greu să găsim dependenţele şi să modificăm un predicat, fără să riscăm să stricăm programul general. Acest lucru este valabil pentru orice limbaj, dar chiar şi mai rău pentru Prolog din cauza nevoii sale frecvenţe de predicate de "ajutor".

Un modul Prolog încapsulează un set de predicate şi defineşte interfaţă. Modulul poate importa alte module, ceea ce creează dependenţe explicite. Având în vedere dependenţele explicit şi o interfaţa bine definită, devine mult mai uşor să schimbăm organizarea internă a unui modul fără a strica aplicatia generală.

Prolog conţine două module speciale . Primul este modulul system. Acesta conţine toate predicatele din librăria standar Prolog, cea care se încarcă la lansare programului. Acesta nu importă alte module. Al doilea modul special se numeşte user, acesta fiind spaţiul de lucru implicit la pornirea Prolog-ului. Iniţial este gol, dar importă modulul system. Astfel toate predicatele predefinite devin disponibile.

Modulele au două proprietăţi declarate, numele modului şi predicatele pe care acel modul le exportă. Astfel noi am creat două module şi le-am folosit pentru a încarcă în fiecare din ele un proiect. Predicatele din proiectele încărcate nu vor fi exportate, declarate vizibile. Vom declară vizibile nişte predicate de acces la date pentru fiecare modul. Am făcut convenţia ca predicatele exportate din primul modul să se termine în 1 iar cele din cel de-al doilea modul să se termine cu 2.

În proiectul nostru avem create două module mod1 şi mod2. Inacestea se vor încărca pe rând proiectele comparate. Modulele se creează în mod static, declarând modulul cu module/2, într-un fişier pe prima linie ca în Fig.5.1. După cum se observă declaraţia va conţine numele, urmată de predicatele exportate. Nu se vor exporta predicatele generate de JTransformet, cum ar fi classT, methodT s.a pentru a evita problema descrisă. Pentru încărcare celor dou module în modulul de start, user, folosim apelul use_module/2.

După cum se observă din figura în acest fişier nu am plasat logica de interogare a bazei de cunoştinţe ci o apelăm prin redirectare, impelementarea fiind realizată în alt fişier.

Am ales această abordare deoarece cele două proiecte încărcate în module diferite, vor fi interogate la fel şi ar fi fost greşit să implementăm aceleaşi lucru în două locuri. Ar fi mult mai greu de modificat şi întreţinut astfel. Logica de interogare a bazei de cunoştinţe va fi implementată în utilities.pl, fişier importat de ambele module.

Modulul doi va fi identic cu modulul 1 schimbându-se doar numărul de la finalul fiecărui predicat din 1 în 2.

```
:-module(mod1,[myClass1/2,load1/1,calcNrAtrib1/2,clearDatabase1/0,calcNrMet1/2,
               calcNrInterf1/2,calcNrExtends1/2,methodsOfClass1/3,nrOfCycles1/2,
               nrOfIf1/2,nrOperators1/3,callT1/3,isInterface1/1,isStatic1/1,
               paramMet1/2,param1/3]).

:-include('./utilities.pl').
:-include('./commonClasses.pl').

load1(Prj):- load(Prj).

myClass1(ClassId,Name):- myClass(ClassId,Name).

calcNrAtrib1(IdClasa,Nr):- calcNrAtrib(IdClasa,Nr) .

clearDatabase1:-clearDatabase.

calcNrMet1(IdClasa,Nr):-calcNrMet(IdClasa,Nr).

calcNrInterf1(IdClasa,Nr):-calcNrInterf(IdClasa,Nr).

calcNrExtends1(IdClasa,Nr):-calcNrExtends(IdClasa,Nr).

methodsOfClass1(IdClasa,IdMethod,Name):-methodsOfClass(IdClasa,IdMethod,Name).

nrOfCycles1(MethodId,Nr):-nrOfCycles(MethodId,Nr).

nrOfIf1(MethodId,Nr):-nrOfIf(MethodId,Nr).

nrOperators1(IdMethod,Operator,Nr):-nrOperators(IdMethod,Operator,Nr).

callT1(MethodId1,CalledClassId1,CalledMetId1):-
        callDep(MethodId1,CalledClassId1,CalledMetId1).
isInterface1(ClassId):-isInterface(ClassId).
```

Fig. 5.1 Interfata catre modulul 1

În concluzie, ar trebui să privim modulele ca două bucăţi din toată baza de cunoştinţe Prolog. Toată baza de cunoştinţe este încărcată în modulul user, Fig. 5.2, predicatele predefinite, predicatele implementate de noi ce conţin logică progamului şi cele două module. Din modulul user nu se pot accesa decât predicatele vizibile ale modulului 1 respectiv 2, celelate pot fi apelate doar prin intermedium celor vizibile , adică din cadrul modulului.

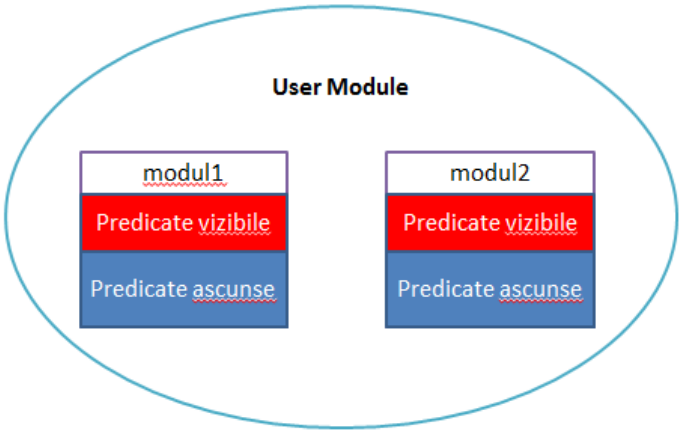


Fig. 5.2 Reprezentare module

2. *Imparăţirea pe fişiere*

În continuare vom descrie împărţirea pe fişiere în cadrul programului şi o scurtă descriere a fiecăruia.

Proiectul CloneDetector este format din 4 fişiere de bază şi două de configurare :

- cloneDetector.pl
- utilities.pl
- IProject1.pl
- IProject2.pl
- commonClasses.pl
- profiles.pl

Fişierul cloneDetector.pl este fişierul de lansare a programului şi cel mai complex. Conţine logica de nivel înalt a uneltei, luând deciziile pe bază informaţiilor obţinute despre cele două proiecte. Se încarcă automat la execuţie în modulul user şi va încărca cele două module : mod1, mod2.

Utilities.pl este fişierul încărcat în ambele module. El conţine predicate de interogare a bazei de cunoştinţe, obţinând informaţii despre proiecte, calculând metrici. Este al doilea ca şi complexitate. Nici un predicat din el nu va fi vizibil în modulul user, predicatele fiind apelate prin redirectare : apelăm predicatul unui modul care va apela predicatele din acest fişier.

IProject1.pl respectiv IProject2.pl sunt fişierele de modul. Întrucât în baza de cunoştinţe exportată de JTransformet nu se pun automat directive de modul, am creat aceste două fişiere care creează modulul şi interfara către cele două proiecte încărcate la un moment dat în baza de cunoştinţe. Figura 5.1 este o poză a fişierului IProject1.pl. După cum spuneam ele definesc modulul la început. Tot în aceste fişiere se încarcă în module fişierul comun utilities.pl prin directivă *include*.

Fişierul commonClasses.pl este un fişier de configurare în care ne putem defini clasele ce dorim să le excludem din analiză. Se foloseşte când avem proiecte care au cod dat de la început şi nu vrem să ne afecteze compararea, generând prea mulţi suspecţi. O descriere a modului de completarea acestor fişiere se găseşte în capitolul VI. Manualul utilizatorului.

În profiles.pl sunt setate profilele de execuţie, aspect asupra căruia vom reveni. Şi acesta este un fişier de configurare.

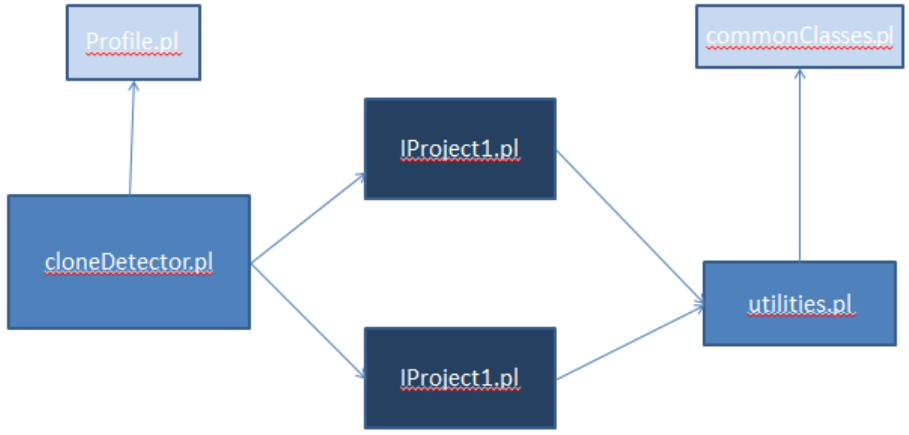


Fig. 5.3 Relatii intre fisier

În Fig. 5.3 prezentăm relaţiile dintre fişiere. Săgeata reprezintă ce fiser interogeaza pe altul, vârful săgeţii fiind îndreptat spre cel folosit. Nuanţele de albastru sugerează asemănările dintre ele, fişierele deschise fiind de configurare, cele închise definind modulele iar celelalte două cuprinzând logica programului de nivel înalt şi de interogare a bazei de date.

3. Diferenţierea claselor proiectelor de clasele librariilor

JTransformer generează fapte atât pentru clasele scrise de noi cât şi pentru clasele din librăriile incluse, chiar şi din librarial standard Java. Nu dorim să grupăm clase care nu sunt scrise de noi sau să le luăm în considerare în calculalea unor metrici. De aceea am creat predicatul myClass care exclude din analiză clasele librăriilor şi clasele pe care noi le declarăm comune pentru mai multe proiecte.

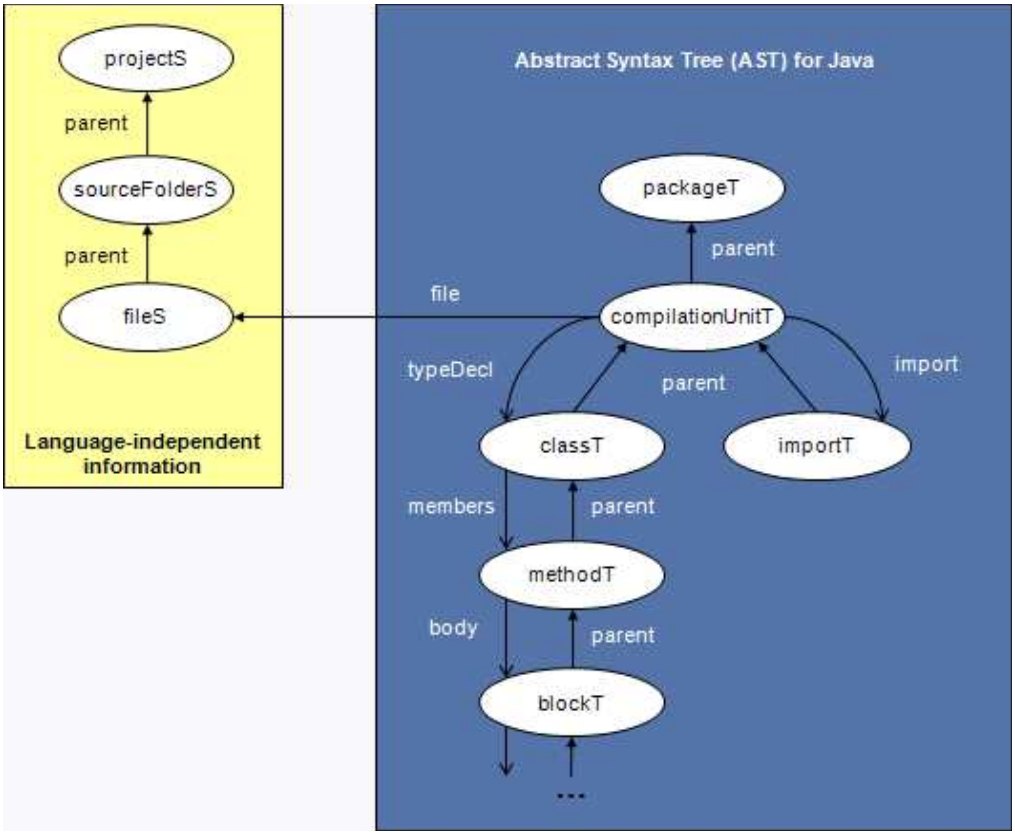


Fig. 6.4 Legaturi intre fapte specifice Java si independente de limbaj

JTransformer generaza două tipuri de fapte, independente de limbaj şi dependente, după cum se poate vedea şi în figura 6.4. Relaţiile dintre ele sunt prezentate prin săgeţi, arcul săgeţii fiind îndreptat spre părinte. Intre majoritate clauzelor există arce în ambele direcţii. Arcele se stabilesc în baza de cunoştinţe prin identificatori.

Noi am folosit aceste relaţii între fapte pentru a determină care sunt clasele nostre. Pentru clasele din librării nu se generează clauze independente de limbaj. Predicatul myClass generează toate clasele unui proiect pornind de la axioma classT şi verificând ca aceasta să aibe axiomele independente de limbaj generate.

```
myClass(AllCls,Name) :-
classT(AllCls,Cu,Name,_),
compilationUnitT(Cu,_,Fid,_,_),
fileS(Fid,Src,Path),
sourceFolderS(Src,Pid,_),
projectS(Pid,_,_,_,_),
findall(PartPath,common(Path,PartPath),Result),
count(Result,Nr), Nr=0.
```

Pentru clasele ce fac parte din proiectul nostru dar dorim noi să le excludem, de obicei ele fiind date la începutul proiectului şi deci le regăsim în toate proiectele, se generează aceste axiome independente de limbaj. Pentru a le distingem am creat un predicat special common ce verifică dacă o clasă a fost definită ca fiind comună.

```
common(Path,PartPath):-commonClass(PartOfPath),
        ( atom_concat(PartPath,PartOfPath,Path);
          atom_concat(PartOfPath,PartPath,Path) ).
```

4. Implementarea metricilor

In continuare vom prezenta modul de implementare a metricilor propuse cu exemplificari din codul sursa. Intrucat pentru implementare lor sunt apelate predicate din mai multe fisiere vom specifica, unde este cazul din ce fisier fac parte. Este important sa retinem ca predicatele care se termina în 1 sunt din fisierul IProject1, ce descrie modulul 1, incarcand în el primul proiect iar cele ce se termian în 2 sunt din fisierul IProject2 care indeplineste aceeasi functionalitate pentru proiectul 2.

A. Metrici la nivel de casa

Fiecare metrica este un filtru de care clasele suspecte pot trece sau nu. Trecerea lor presupune existenta asemanorilor si continuare catre urmatorul filtru. Picarea unui filtru intrerupe compararea acestor clase. Ordinea în care sunt puse filtrele are importanta, putand imbunatati viteza de executie a programului. Daca punem filtre usor de calculat si care caracterizeaza mai bine clasele, vom opri compararea claselor diferite mai repede.

In cod filtrele la nivel de clasa sunt apelate de predicatul *compareClassLevel/2* care primeste ca termeni identificatorii unici ai celor doua clase comparate.

```
compareClassLevel(ClassID1,ClassID2):-
        areInterfaces(ClassID1,ClassID2),
        compareNrAtrib(ClassID1,ClassID2),
        compareNrMet(ClassID1,ClassID2),
        compareNrInterf(ClassID1,ClassID2),
        compareNrSuperClass(ClassID1,ClassID2).
```

Ordinea filtrelor aleasa de noi este prezentata în portiunea de cod de mai sus.

Primul filtru *areInterfaces/2* verifica ca id-urile primite sa fie ambele a unor clase sau a unor interfete. JTransformer va genera atat pentru clase cat si pentru interfete o fapta de tipul classT, iar pentru interfete va mai genera una în plus de tipul interfaceT.

```
        %returnes true if class is an interface.
        isInterface(ClassId):-
                interfaceT(ClassId).

        areInterfaces(ClassID1,ClassID2):-
                (  isInterface1(ClassID1),
                isInterface2(ClassID2));

                (  not(isInterface1(ClassID1)),
                   not(isInterface2(ClassID2))).
```

Filtrul al doilea numara numarul de atribute al unei clase si verifica ca doua clase au diferenta dintre numarul de atribute mai mic ca MaxDifference si procentul dintre cele doua mai mare ca MinProcent. Acest tolerante se initializeaza în functie de profilul setat.

```
compareNrAtrib(Class1,Class2):-
        profile(Profile),
        classDelta(Profile,[MaxDifference,MinProcent,_,_,_,_,_]),
        calcNrAtrib1(Class1,Nr1),
        calcNrAtrib2(Class2,Nr2),
        (
        (delta(Nr1,Nr2,Difference),     Difference =< MaxDifference),
        (procent(Nr1,Nr2,Procent),      MinProcent =<Procent)         ).
```

Numărul de atribute din ficare clasă se obţine apelând interfeţele pentru fiecare modul în care sunt încărcate cele două proiecte comparate. Apelurile vor fi redirectate către predicatul din fişierul utilities.pl calcNrAtrib/2. Pentru înţelegerea predicatului, vom explică mai întâi cum se utilizează *findall/3*. Findall este un predicat oferit de Prolog pentru a genera toate soluţiile unei reguli. Generează o listă de soluţii, memorată în ultimul termen. Primul termen este cel ce va fi adăugat în listă pentru fiecare soluţie găsită, iar al doile este regua care va genera soluţiile. Astfel pe exemplu de mai jos, căutăm toate faptele fieldT care au ca părinte clasa primită ca parametrul. Identificatorul , notat cu X, va fi adăugat penru fiecare soluţie în lista Result. Numărul de elemente din lista rezultată va fi numărat prin clauza count şi returnat prin termenul Nr. Această abordare se adopata la multe predicate ce calculează metrici, iar înţelegerea modului de funcţionare a predicatului predefinit findall este esenţială. Dacă nu am fi pus findall, ci am fi apelat filedT pur şi simplu am fi obţinut tot aceste soluţii dar nu deodată, s-ar fi aşteptat intereactiunea utilizatorului pentru a genera următoarea soluţie.

```
calcNrAtrib(IdClasa,Nr) :-
        findall(X,fieldT(X,IdClasa,_,_,_),Result),
        count(Result,Nr).
```

Trebuie să justificăm de ce am ales doar să numărăm atributele unei clase, în loc să verificăm şi tipul lor. Verificarea tipului unui atribut şi existenţa unuia de acelaşi tip în clasa comparată este uşor pentru tipurile elementare (String, int, float...), dar dacă avem ca atribute clase proprii lucrurile se complică deorece nu ne putem baza pe numele lor în comparare. Ar trebui să verificăm asemănarea lor nu după nume ci după metrici, eventual după atributele lor, metodele s.a.m.d. Astfel am merge recursiv dintr-o clasă în alta, iar în unele cazuri am obţine chiar cicluri dacă o clasă conţine ca atribut o alta care o va conţine la rândul ei pe ea, dependentă dublă între ele. Astfel ne trebuiau nişte metrici independente de tip, care să nu genereza cicluri infinite, pentru compararea claselor. Deşi mai slabe, prin necomparearea tipurilor, metricile alese la nivel de clasă nu generează cicluri şi sunt temelia analizei. Verificare tipurilor se va face ulterior la nivel de metodă şi ne vom baza pe aceste metrici pentru asta.

Predicatul compareNrMet(ClassID1,ClassID2) este asemănător în implementare cu cel ce calculează numărul de atribute, calculând numărul de metode. De această dată toleranta metodelor este exprimată doar prin diferenţa absolută între numărul lor, în funcţie de profilul ales.

```
compareNrMet(ClassID1,ClassID2):-
        profile(Profile),
        classDelta(Profile,[_,_,MaxDifference,_,_,_,_]),
        calcNrMet1(ClassID1,Nr1),
        calcNrMet2(ClassID2,Nr2),
        delta(Nr1,Nr2,Difference),
        Difference =< MaxDifference.
```

Următoarele două predicate se leagă de relaţiile de moştenire ale claselor, o caracteristică importantă în programarea orientată pe obiecte. Verificăm că numărul de interfeţe pe care o clasă îl implementează să fie aproximativ acelaşi iar dacă o clasă moşteneşte pe alta atunci şi posibilă copie va trebui să extindă o clasa.

```
compareNrSuperClass(ClassID1,ClassID2):-
        calcNrExtends1(ClassID1,Nr1),
        calcNrExtends2(ClassID2,Nr2),
        Nr1 = Nr2.

compareNrInterf(ClassID1,ClassID2):-
        profile(Profile),
        classDelta(Profile,[_,_,_,MaxDifference,_,_,_]),
        calcNrInterf1(ClassID1,Nr1),
        calcNrInterf2(ClassID2,Nr2),
        delta(Nr1,Nr2,Difference),
        Difference =< MaxDifference.
```

Nu am mai prezentat cum este interogată baza de cunoştinţe pentru ultimele filtre pentru că este asemănător cu calcularea numărului de atribute. Se foloseşte findall dar pe fapte diferite *methodT, extendsT, implementsT*.

B. Metrici la nivel de metoda

În compararea a două clase, după filtrele la nivel de clasă , CloneDetector urmează să compare fiecare metodă a claselor încercând să le grupeze. Primul nivel de analiză a metodelor este la nivel de metrici. Predicatul din care porneşte analiza la nivel de metodă este *compareMethodLevel/4* care apeleaza predicatul *methodMetrics/2* pentru fitrele la nivel de metodă.

Metricile la nivel de metoda sunt inlantuiete astfel. Am pus primul filtru cel ce ia în considerare ciclurile din programe deoarece aceste sunt rareori modificate si reprezinta o semnatura puternica a metodei.

```
methodMetrics(MethodId1,MethodId2):-
        whileFilter(MethodId1,MethodId2),
        operatorsFilter(MethodId1,MethodId2),
        ifFilter(MethodId1,MethodId2),
        areStatic(MethodId1,MethodId2),
        compareSigniture(MethodId1,MethodId2).
```

whileFilter verifică numărul de cicluri dintr-o metodă, numărând câte while, do while şi for există. Este foarte greu să schimbi într-o clonă o astfel de metrică, înlocuirea ciclurilor repetitive putând fi făcută doar prin recursivitate. De aceea în acest caz nu admitem tolerante, numărul ciclurilor trebuie să fie exact la fel în două metode copiate.

```
whileFilter(MethodId1,MethodId2):-
    nrOfCycles1(MethodId1,Nr1),
    nrOfCycles2(MethodId2,Nr2),
    Nr1=Nr2.
nrOfCycles(MethodId,Nr):-findall(MethodId,whileT(_,_,MethodId,_,_),While),
    count(While,NrWhile),
    findall(MethodId,doWhileT(_,_,MethodId,_,_),DoWhile),
    count(DoWhile,NrDoWhile),
    findall(MethodId,forT(_,_,MethodId,_,_,_,_),For),
    count(For,NrFor),
    Nr is (NrWhile + NrDoWhile + NrFor).
```

Filtru operatorsFilter implementează metrică propusă de Halstead, care spune că o bucate de cod poate fi definită de numărul de operatori şi de tipul lor. Operatori propuşi sunt de adunare, scădere, înmulţire şi împărţire, operaţiile matematice elementare.

În urma analizei acestei metrici am observat că ea ajută la distingerea claselor, dar în cazul Java operatorul "+" nu poate fi analizat el fiind folosit de numeroase ori pentru concatenarea şişurilor. Astfel prin simplă schimbare a mesajelor afişate, şi ştergerea operatorului de concatenare s-ar putea împiedica detecţia clonei. De aceea noi am analizat doar ultimii trei operatori.

```
operatorsFilter(MethodId1,MethodId2):-
    nrOperators1(MethodId1,-,NrSub1),
    nrOperators2(MethodId2,-,NrSub2),
    NrSub1=NrSub2,

    nrOperators1(MethodId1,/,NrDiv1),
    nrOperators2(MethodId2,/,NrDiv2),
    NrDiv1=NrDiv2,

    nrOperators1(MethodId1,*,NrMul1),
    nrOperators2(MethodId2,*,NrMul2),
    NrMul1=NrMul2.
```

Se observă că se cere o egalitate exactă a operatorilor între două metode copiate. Toate tipurile de operatori sunt memoraţi în baza de cunoştinţe prin fapte de tipul *operationT*.

```
ifFilter(MethodId1,MethodId2):-
    profile(Profile),
    methodDelta(Profile,[MaxDifference]),
    nrOfIf1(MethodId1,Nr1),
    nrOfIf2(MethodId2,Nr2),
    delta(Nr1,Nr2,Delta),
    Delta =< MaxDifference.
```

Filtrul la nivel de condiţii compară câte if-uri sunt într-o metodă. Am admins o tolerantă în acest caz, pentru posibile adăugări sau ştergeri de condtii, prin tratarea unor cazuri speciale sau renunţarea le ele.

Filtrul *areStatic* este asemănător ca implementare cu filtrul de la nivel de clasă *areInterface*. Acesta va verifica dacă o metodă este sau nu statică şi va impune ca şi perechea ei sa fie.

Un filtru complex care diferenţiază destul de mult metodele este ultimul de la acest nivel, *compareSigniture/2* care compară semnătura a două metode. Deşi este un filtru puternic a fost lăsat la sfarşit pentru că el consumă mai mult timp de procesare, comparările făcute de el fiind similare cu cele de la grafurile de apeluri.

Filtrul compară ca numărul de parametrii a două metode să fie egal dar şi ca pentru fiecare parametru de un anume tip să existe unul de acelaşi tip în metoda copiată. Adică dacă am avea un parametru String în prima metodă să există unul de acelaşi tip şi în cea suspectă.

```
compareSigniture(MethodId1,MethodId2):-
        paramMet1(MethodId1,ListParam1),
        paramMet2(MethodId2,ListParam2),
        count(ListParam1,NrParam1),
        count(ListParam2,NrParam2),
        NrParam1 = NrParam2,
        findall(_,paramCompare(MethodId1,MethodId2),_),
        findall(Id,signMatch(Id,_),ResultList),
        count(ResultList,NrMatch),
        retractall(signMatch(_,_)),
        NrMatch = NrParam1 .
```

Se obţine la început listă parametrilor celor două metode şi se verifaca dacă numărul lor este egal. Incaz contrar se opreşte compararea. Dacă au acelsi număr de parametrii se vor compara pentru toţi tipuri şi se va încercă se se grupeze doi câte doi. Memorarea perechilor se va face dinamic prin fapte *signMatch* care au ca termeni id-urile celor doi parametrii. Se folosete predicatul predefinit *assert.*

```
paramCompare(MethodId1,MethodId2):-
        param1(MethodId1,ClsId1,IdParam1),
        param2(MethodId2,ClsId2,IdParam2),
        not(signMatch(IdParam1,_)),
        not(signMatch(_,IdParam2)),
        compareClassLevel(ClsId1,ClsId2),
        assert(signMatch(IdParam1,IdParam2)).
```

Dacă un parametru a fost deja împerecheat cu unul, el nu va mai fi grupat cu altul, lucru verificat la începutul predicatului paramCompare. Observăm că tipul parametrilor se verifică comparând cele două clase cu metricile definite la nivel de clasă. Această abordare este corectă atât pentru parametrii ce au tipuri elementare sau definite de noi, JTransformer generând interfeţele claselor şi pentru clasele din librării.

După ce s-a încercat să se grupeze toţi parametrii, se verifică ca numărul celor împerecheaţi să coincidă cu numărul total de parametrii. Pentru ca baza de cunoştinţe să rămână intactă pentru următoarea verificare se şterg toate clauzele alocate dinamic prin predicatul predefinit *retractall.*

5. *Grafuri de apel*

Dacă o pereche de două clase suspectate de copiat trece de primele două mari categorii de filtre, metrici la nivel de clasă şi la nivel de metodă, urmează să se analizeze la nivel de metodă toate apelurile pe care o metodă le face către o alta. După cum am explicat în capitolul de descriere a algoritmului, metodele dar şi clasele apelate trebuie să fie asemănătoare. Această asemănare este stabilită dacă clasele şi metodele analizate trec de metricile la nivel de clasă respectiv de metodă.

În baza de cunoştinţe apelurile sunt reprezentate sub forma de fapte *callT* iar id-ul metodei şi clasei apelate se obţine cu predicatul *callDep*. Acesta analizează doar apelurile între clasele proiectului, nu şi către librării.

```
callDep(MethodId,CalledClassId,CalledMetId):-
        callT(_,_,MethodId,_,_,_,CalledMetId),
        methodT(CalledMetId,CalledClassId,_,_,_,_,_),
        myClass(CalledClassId,_).
```

Predicatul *callDependencies* este cel din care se porneşte analiza şi cel care stabileşte condiţiile pentru a trece de acest filtru. Ţinând cond că ar fi complicat să se schimbe apelurile către metode în proiecte copiate şi este rareori întâlnit, nu am permis tolerante cerând o egalitate exactă. La început verificăm, înainte de a încerca să grupăm clasele, dacă au acelaşi număr de apeluri. După aceea generam grupurile de apeluri similare, le numărăm şi verificăm să fie eagale cu numărul total de apeluri.

```
callDependencies(MethodId1,MethodId2):-
        findall(CalledMet1,callT1(MethodId1,_,CalledMet1),ListMet1),
        uniqueList(ListMet1,UniqueList1),
        count(UniqueList1,NrCalledMet1),
        findall(CalledMet2,callT2(MethodId2,_,CalledMet2),ListMet2),
        uniqueList(ListMet2,UniqueList2),
        count(UniqueList2,NrCalledMet2),
        NrCalledMet1 = NrCalledMet2,
        findall(MethodId1,compareAllCallT(MethodId1,MethodId2),MatchingCalls),
        count(MatchingCalls,Nr),
        retractall(partialMatch(_,_)),
        retractall(partialMetMatch(_,_)),
        Nr =  NrCalledMet1.
```

Predicatul care generează şi testează grupările de apeluri posibile este compareAllCallT Se verifică înainte de a încerca să grupăm două apeluri, dacă unul dintre ele nu este deja grupat. Se observă că grupurile de metode sunt alocate dinamic prin fapte de tipul partialMetMatch iar clasele prin fapte de tipul partialMatch. Aceste fapte memorează identifiatorii celor două perechi.

```
compareAllCallT(MethodId1,MethodId2):-
        callT1(MethodId1,CalledClassId1,CalledMetId1),
        callT2(MethodId2,ClassId2,CalledMetId2),
        not(partialMetMatch(CalledMetId1,_)),
        not(partialMetMatch(_,CalledMetId2)),

        methodMetrics(CalledMetId1,CalledMetId2),
```

```
(
   (
   not(partialMatch(CalledClassId1,_)   ,
   not(partialMatch(_,ClassId2)),
   compareClassLevel(CalledClassId1,ClassId2),
   assert(partialMetMatch(CalledMetId1,CalledMetId2)),
   assert(partialMatch(CalledClassId1,ClassId2))
   );
  (
  partialMatch(CalledClassId1,ClassId2),
  not(partialMetMatch(CalledMetId1,CalledMetId2)),
  assert(partialMetMatch(CalledMetId1,CalledMetId2))
  )
).
```

Testând programul de-al lungul dezvoltării sale, am realizat că aceast filtru are o importantă deosebită ducând la o creştere semnificativă a corectitudinii grupării claselor si a verdictelor date. Aceast filtru se poate îmbunătăţii, surprinzător, odată cu dezvolaterea metricilor la nivel de clasă sau metode, deoarece ea foloseşte aceste metrici la compararea entităţilor apelate.

6. *Luarea deciziilor*

Până acum am prezentate în mare modul de implementare al algoritmului explicând partea de metrici la nivel de clasă şi metodă şi modul de verificare a apelurilor metodelor. Totuşi intercalat sau după aceste filtre sunt numeroase puncte de decizie în program care au un rol crucial. Toate rezultatele generate de filtre fiind interpretate în aceste puncte urmând să se enunţe unele verdicte. Putem să ne imaginăm această unealtă ca pe un proces juridic din viaţa reală, unde probele sunt datele generate de filtre, iar verdictele intermediare sau finale date de judecător sunt deciziile date de aceste puncte de control.

Vom prezenta modul de luare al deciziilor pentru clase. După cum am mai amintit există trei nivele de asemănare între clase *LOW, MEDIUM, HIGH* . Compararea claselor a două proiect începe în predicatul *generateAllMatchingClasses*. Se generează perechi de clase care nu au fost încă memorate în baza de date cu probabilitate ridicată de comparare. Acestea sunt comparate în predicatul *compare2Classes.* Dacă clasele trec de toate filtrele sunt declarate foarte probabil copiate şi memorate în baza de cunoştinţe.

```
generateAllMatchingClasses:-
    myClass1(Id1,Name1),
    myClass2(Id2,Name2),
    not(match(Id1,_,_,_,high)),
    not(match(_,_,Id2,_,high)),
    compare2Classes(Id1,Name1,Id2,Name2),
    retractall(match(Id1,_,_,_,_)),
    retractall(match(_,_,Id2,_,_)),
    assert(match(Id1,Name1,Id2,Name2,high)).

compare2Classes(Id1,Name1,Id2,Name2):-
    compareClassLevel(Id1,Id2),
    assertClassMatchLow(Id1,Name1,Id2,Name2),
    compareMethodLevel(Id1,Name1,Id2,Name2) .
```

Daca în predicatul *compare2Classes* se trece de compararea la nivel de clasa vom memora dinamic în baza de cunostinte perechea de clase ca avand probabilitate scazuta de asemanare. Urmeaza sa fie analizate la nivel de metoda, probabilitate putand fi schimbata.

```
assertClassMatchLow(Id1,Name1,Id2,Name2):-
        (
        not(match(Id1,_,_,_,medium)),
        not(match(_,_,Id2,_,medium)),
        not(match(Id1,_,_,_,low)),
        not(match(_,_,Id2,_,low)),
        assert(match(Id1,Name1,Id2,Name2,low)),!
        ) ;1=1.
```

Continuăm prin a prezenta cum sunt luate deciziile în continuare, în predicatul *compareMethodLevel*. Aici practic se analizează toate metodele dintr-o clasă. Clauza începe prin a genera toate grupurile de metode cu cele două tipuri de probabilităţi scăzute sau ridicate. Odată ce avem aceste perechi verificăm ca diferenţele între numărul lor şi numărul total de metode ale celor două clase să fie în limitele stabilite de profil, atât din punct de vedere al diferenţelor absolute câţ şi procentual. Predicatul e gândit că dacă nu sunt suficiente metode grupate să nu se schimbe probabilitatea de copiere a celor două clase şi să returneze fals. Dacă sunt suficiente perechi dar nu sunt cu probabilităţi mari de a fi copiate, atunci clasele devin medium şi predicatul returnează fals. Dacă se ajunge la finalul predicatului înseamnă că nu suntem în cele două cazuri, deci clasele sunt foarte probabil copiate şi returnăm true. Returnând true, clasele devin foarte probabil copiate în predicatul de un este apelat *compareMethodLevel.*

```
compareMethodLevel(ClassId1,Name1,ClassId2,Name2):-
        findall(_,generateAllMatchingMethods(ClassId1,ClassId2),_),
        findall(Id,methodMatch(Id,_,_,_,high),ResultHigh),
        count(ResultHigh,NrOfMatchesHigh),
        findall(Id,methodMatch(Id,_,_,_,low),ResultLow),
        count(ResultLow,NrOfMatchesLow),
        NrOfMatches is NrOfMatchesHigh + NrOfMatchesLow,
        calcNrMet1(ClassId1,NrMet1),
        calcNrMet2(ClassId2,NrMet2),
        retractall(methodMatch(_,_,_,_,_)),
        profile(Profile),
        classDelta(Profile,[_,_,_,_,_,MaxNrOfUnmatchingMethods,MinProcent]),
        delta(NrOfMatches,NrMet1,Delta1),
        Delta1=<MaxNrOfUnmatchingMethods,
        delta(NrOfMatches,NrMet2,Delta2),
        Delta2<MaxNrOfUnmatchingMethods,
        procent(NrMet1,NrMet2,Procent),
        Procent> MinProcent,
        procent(NrOfMatchesHigh,NrOfMatches,ProcentMatchHigh),
        not(
           assertClassMedium(ClassId1,Name1,ClassId2,Name2,ProcentMatchHigh)
          ).
```

assertClassMedium este predicatul în care decidem daca clasele ar trbui sa fie medium sau daca le vom memora ca perechi high.

```
assertClassMedium(ClassId1,Name1,ClassId2,Name2,ProcentMatch):-
        profile(Profile),
        classDelta(Profile,[_,_,_,_,ProcentMatchHigh,_,_]),
        ProcentMatch <ProcentMatchHigh
        retractall(match(ClassId1,_,_,_,_)),
        retractall(match(_,_,ClassId2,_,_)),
        assert(match(ClassId1,Name1,ClassId2,Name2,medium)).
```

În această secţiune am prezentat doar deciziile luate la nivel de clasă, modul în care hotărâm dacă clasele sunt copiate cu probabilităţi ridicate, medii sau scăzute. Deciziile acestea se bazează pe modul în care metodele sunt grupate şi probabilităţile lor. Deci, evident, alte puncte de decizie vor fi şi la nivel de metodă. Acestea sunt relativ similare, şi nu vor mai fi prezentate în continuare. Se pot inspecta cu atenţie în program dacă se doreşte la predicatele *generateAllMatchingMethods* si *assertMetMatch.*

7. Compararea a doua proiecte

În această parte vom descrie modul în care se lansează în execuţie unealtă, şi operatiuniile pe care le executăm la început.Comparearea a două proiecte începe din clauza *run/3* care primeşte ca termeni numele cu calea către proiecte, şi profilul.

La început este golită baza de cunoştinţe de faptele memorate de o execuţie anterioară şi încărcăm în ea profilul ales, interfeţele către cele două module şi în cele două module încărcăm proiectele comparate.

Înainte de a lansa compararea tuturor claselor verificăm ca numărul claselor din cele două proiecte să nu difere foarte mult, altfel fiind inutilă compararea. *generateAllMatchingClasses* va genera toate perechile de clase copiate în baza de cunostine.

Ultimele funcţii îndeplinite de predicatul run fiind interpretarea rezultatului la nivel de proiect şi afişarea lui. Verificarea se face prin compararea diferenţelor absolute între numărul de clase împerecheate şi numărul de clase din cele două proiecte, ţinând cond de tolerante.

Codul de mai jos arată modul de implementare a predicatului run, de unde pornim execuţia.

```
run(Proj1,Proj2,Profile):-
        retractall(profile(_)),
        assert(profile(Profile)),
        profile(Profile),
        projectDelta(Profile,[MaxNumberDeltaClasses|_]),
        use_module('./IProject1.pl'),
        use_module('./IProject2.pl'),
        load1(Proj1),
        load2(Proj2),

        findall(Id1,myClass1(Id1,_),ClsPrj1),
        count(ClsPrj1,NrCls1),
        findall(Id2,myClass2(Id2,_),ClsPrj2),
        count(ClsPrj2,NrCls2),
        delta(NrCls1,NrCls2,DeltaClasses),
        DeltaClasses =< MaxNumberDeltaClasses,
        findall(_,generateAllMatchingClasses,_),
        write(Proj1 - Proj2),
```

```
        writef("\n"),
        listing(match),
        findall(Id,match(Id,_,_,_,high),HighMatchList),
        count(HighMatchList,NrHighMatches),
        findall(Id,match(Id,_,_,_,medium),MedMatchList),
        count(MedMatchList,NrMedMatches),
        findall(Id,match(Id,_,_,_,low),LowMatchList),
        count(LowMatchList,NrLowMatches),
        writef("\nHigh matching classes "),
        write(NrHighMatches),
        writef(" of "),
        write(NrCls1),
        writef("\n"),
        retractall(match(_,_,_,_,_)),
        TotalMatches is NrHighMatches+ NrMedMatches +NrLowMatches,
        delta(TotalMatches,NrCls1,NrUnmatched),
        TotalMatches> MinTotalMatches,
        NrHighMatches>MinHighMatch,
        NrUnmatched <MaxUnmatches,
        assert(projectMatch(Proj1,Proj2,high-NrHighMatches,medium-NrMedMatches,low-
        NrLowMatches,unmatched-NrUnmatched)).
```

Pentru celelalte două moduri de execuţie, când rulăm toate proiectele dintr-un director, ne folosim de predicatul *combine2* , care primeşte lista tuturor fişierelor obţinută de la directivă directory_files predefinită în prolog. Restul predicatelor ajutătoare le-am descris prin comentariul pus înaintea lor.

```
%generam calea catre doua proiecte din director.
runFromDir(Directory,Profile):-directory_files(Directory,ListFiles),
        combine2(ListFiles,Proj1,Proj2),
        atom_concat(Directory,Proj1,Project1),
        atom_concat(Directory,Proj2,Project2),
        run(Project1,Project2,Profile).

%din lista data ca parametru generez toate combinarile de cate doua proiecte posibile.
combine2(List,Proj1,Proj2):-
        member(Proj1,List),
        qlfExtension(Proj1),
        member(Proj2,List),
        qlfExtension(Proj2),
        nth1(Index1,List,Proj1),
        nth1(Index2,List,Proj2),
        Index1 <Index2

%verificam ca extensia fisierului sa fie qlf
qlfExtension(Name):-file_name_extension(_, ".qlf", Name).
```

VI. Manualul Utilizatorului

În acest capitol vom prezenta paşii ce trebuie executaţi pentru a rula aplicaţia, începând cu modul de exportare a bazei de cunoştinţe Prolog şi încheind cu apelurile posibile pentru lansarea analizei asupra proiectelor suspecte.

Având instalat plugin-ul JTransformer conform ghidului de instalare oferit de [9] se importă în Eclipse proiectele Java ce se doresc exportate. Pentru fiecare proiect va trebui parcurs următorii paşi pentru obţinerea bazei de cunoştinţe Prolog.

1. Crearea bazei de cunostinţe

Se asignează proiectului o bază de date Prolog, ce va fi calculate de JTransformer. Operaţia se realizează făcând click drepta pe proiect şi selectând din configure oprtiunea *Assign JTransformer Factbase* conform figurii 6.1. Va apărea o fereastră de dialog în care selectăm opţiunea OK. Inacest moment baza de cunoştinţe e generată în memoria de lucru.

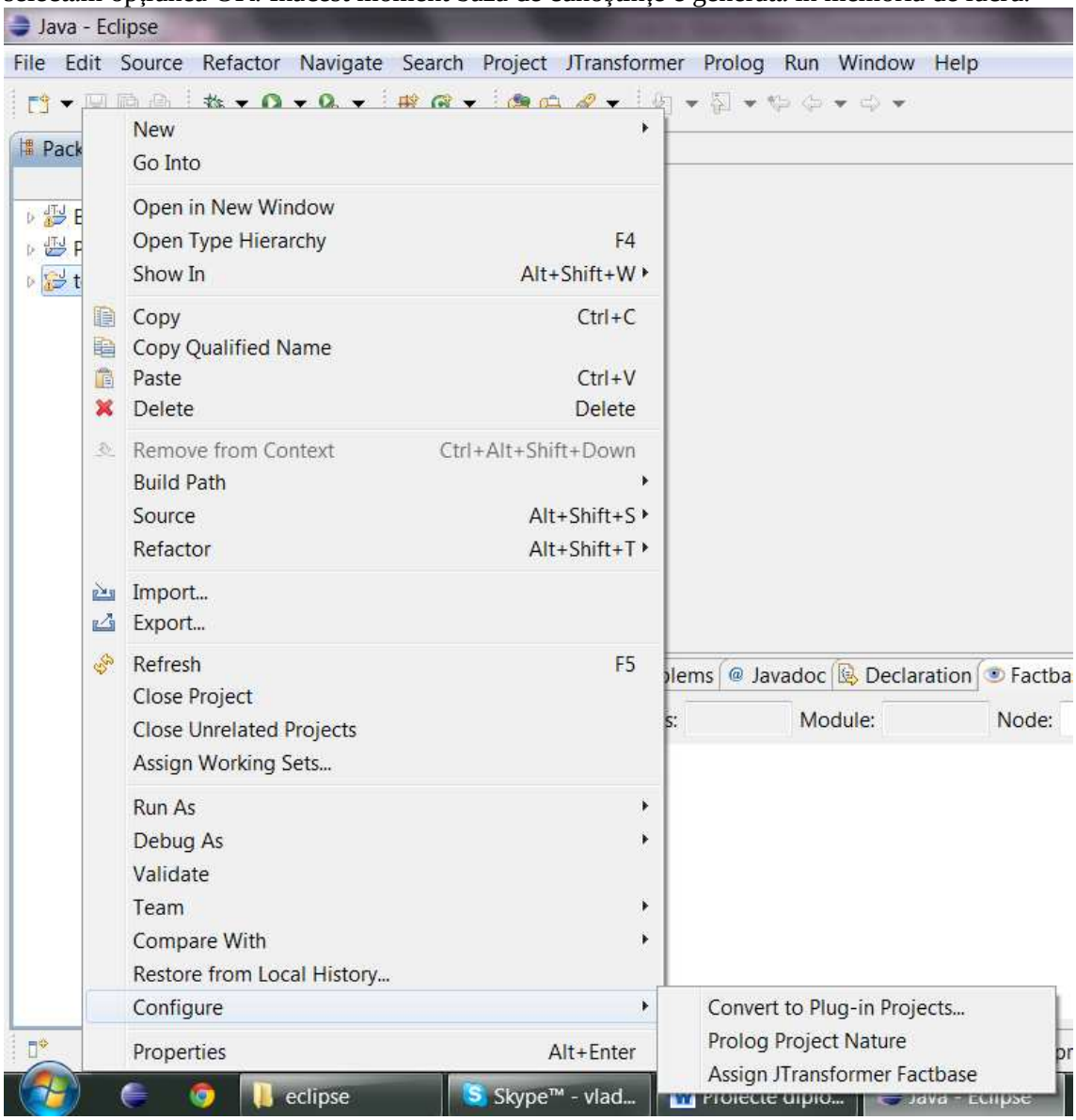


Fig. 7.1 Creerea bazei de cunostinte

2. Salvarea bazei de cunostinte

În următoarea etapă trebuie să exportăm baza de date creată în memoria de lucru sub formă unor fişiere. Asta se realizează selectând oprtiunea Export factbase din meniul contextual al proiectui, conform figurii 6.2. Va apărea fereastra de dialog din figura 6.3. Aici se dă numele bazei de cunoştinţe si locul unde va fi salvată. De asemenea, important este bifăm opţiune care ne creaza fişierele qlf, cea încercuită cu roşu în imaginea 6.3.

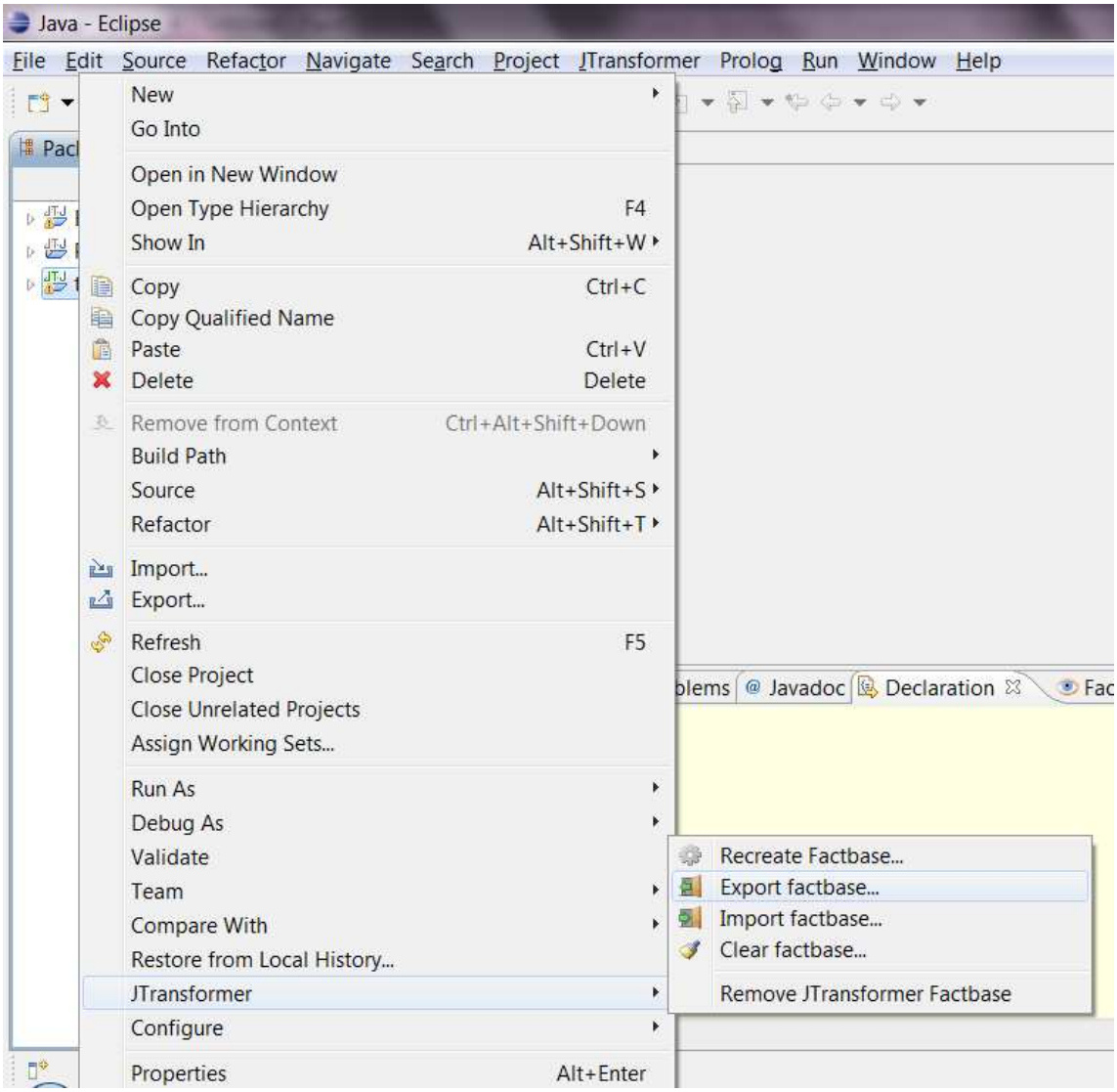


Fig. 6.2 Salvarea bazei de cunostinte în fisier

Baza de cunoştinţe va rămâne memorată şi în mediul de lucru eclipse unde poate fi interogată prinţ facilităţile oferite de JTransformer. Acestea ne-au ajutat mult în dezvoltarea tool-ului dar ca utilizator CloneDetector nu este nevoie să le stăpâniţi şi nu vor fi descrise în continuare. Dacă doriţi să toate facilităţile oferite de JTransformer puteţi urmări linkul oferit la [9].

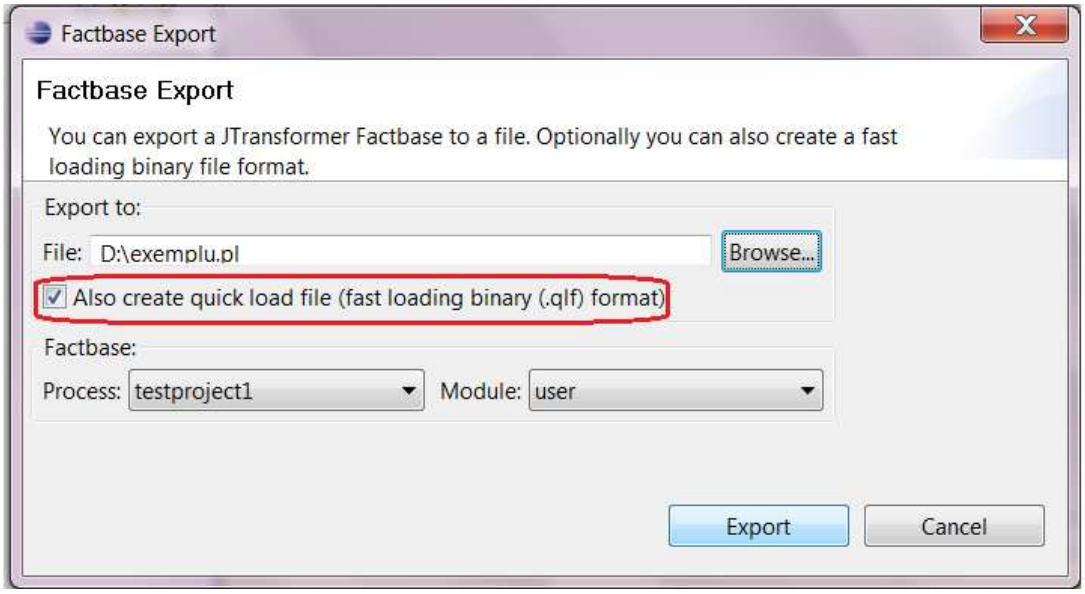


Fig. 6.3 Meniu de salvare

3. Lansarea programului.

Odată create reprezentările proiectelor Java sub forma unor fişiere Prolog, puten lansa în execuţie cloneDetector. Având Prolog-ul instalat aceasta se realizează simplu, cu duplu click pe fişierul cloneDetector.pl.

Pentru ca aplicaţia dezvoltată să fie cât mai utilă profesorilor s-au relizat mai multe moduri de a compara proiectele, apelând diferite predicate.

- Compararea a două proiecte
- Compararea proiectelor dintr-un director, interacţionând cu utilizatorul
- Compararea proiectelor dintr-un director, automată

Avem astfel varianta de compara doar două proiecte dând numele lor şi profilul. In urma acestei comparări afişându-se clasele ce sunt copiate cu probabilităţile lor. Se specifică cate clase au fost copiate din numărul total. Comandă de lansare în execuţie a acestei variante este de forma:

run(‘cale1/numeFisier1.qlf’, ‘cale2/numeFisier2.qlf’, profile).

Căile 1 respectiv 2 pot fi absolute sau relative la directorul din care se lansează în execuţie CloneDetector. Profilele create la început sunt loose şi tight. Se pot crea şi altele după cum vom explică la punctul 4. După execuţie se va afişa pe ecran ce clase seamănă între ele ca în figura 6.4 . Aceste informaţii vor fi afişate şi în celelalte două variante de execuţie, imaginea prezentată fiind pentru cazul doi

Varianta a doua de lansare în execuţie a uneltei, se face dând ca date de intrare directorul ce conţine un set de proiecte sub formă de AST-uri în Prolog şi profilul de analiza.

runFromDir(‘caleDirector’, profile).

Se vor genera grupuri de proiecte combinate două câte două şi se va începe analiza. Pentru fiecare două proiecte analizate se vor afişa clasele grupate cu probabilităţile lor şi ponderea claselor grupate din numărul total. Nu se trece mai departe la analiza următoarelor două proiecte până ce utilizatorul nu apra tasta “;”. După analiza tuturor proiectelor se va afişă lista proiectelor găsite suspecte cu o pondere mai mare. Practic această variantă este una

automatizată dar şi interactivă, programul aşteptând inputul utilizatorului pentru a trece mai departe. Afişarea rezultatelor va fi combinată ca în figura 6.4 pentru fiecare proiect iar la final se va afişă statisca ca în figura 6.5.

```
1 ?-  runFromDir('./factbase/',loose).
% ./IProject1.pl compiled into mod1 0.00 sec, 24,128 bytes
% ./IProject2.pl compiled into mod2 0.00 sec, 17,424 bytes
% ./factbase/georgescu.qlf loaded into mod1 0.02 sec, 4,770,152 bytes
% ./factbase/ionescu.qlf loaded into mod2 0.02 sec, 4,579,304 bytes
./factbase/georgescu.qlf- ./factbase/ionescu.qlf
:- dynamic match/5.

match(25691, 'Medic', 25693, 'Medic', low).
match(26114, 'Main', 26081, 'Main', high).
match(25701, 'Boala', 25704, 'Boala', high).
match(25725, 'BlackBoard', 25737, 'BlackBoard', high).


High matching classes 3 of 4
true
```

Fig. 6.4 Afisarea rezultatelor pentru compararea a doua proiecte sau a unui intreg director, cu interactiune din partea utilzatorului

Ultima variantă de a rula programul este una total automatizată. Parametrii de intrare vor fi directorul cu proiectele si profilul selectat.

runAll(‘caleDirector’, profile).

Programul va genera, asemănător variantei de mai sus, toate combinările posibile între proiecte şi va începe să le compare două câte două. Diferenţa între abordarea de sus este ca de această dată nu se va mai interacţiona cu utilizatorul ci se va continuă execuţia până se combină toate proiectele. La sfârşit se va afişa lista de proiecte suspecte cu numărul de clase grupate după probabilităţile de a fi copiate. Nu se va mai afişa numele claselor grupate, doar numărul total, de aceea pentru pentru proiectele găsite suspecte se mai poate executa încă odată programul pentru a vedea exact cum au fost asociate clase din cele două proiecte.

```
:- dynamic projectMatch/6.

projectMatch('./factbase/georgescu.qlf', './factbase/ionescu.qlf', high-3, medium-0, low-1, unmatched-0).
projectMatch('./factbase/webserver1.qlf', './factbase/webserver2.qlf', high-4, medium-0, low-0, unmatched-0).
projectMatch('./factbase/passc2.qlf', './factbase/passc2Copy.qlf', high-27, medium-0, low-0, unmatched-0).

true.

2 ?- runAll('./factbase/',loose).
```

Fig.6.5 Afisarea proiectelor suspecte

În Fig. 6.5 se prezintă modul de afişarea a proiectelor suspecte în cazul evaluării unui director întreg. Se observă că primii doi termeni sunt numele proiectelor urmate de numărul de clase asemănătoare pentru fiecare din cele 3 nivele de similitudine : high, medium, low. Ultimul termen este numărul de clase neimperecheate.

4. **Editarea sau crearea profilelor**. După cum am menţionat de numeroase ori în această lucrare pentru a putea face modificări în modul de analiză uşor în program, în funcţie de tipul proiectelor analizate am creat două filtre şi opţiunea de a adăuga uşor altele noi. Infigura 6.6 se presinta o parte a fişierului profiles.pl, locul unde sunt definite filtrele.

```
%loose profile
projectDelta(loose,
             [
              2 % MaxDeltaClasses- diferenta maxima intre nr de clase din primul si cel de-al doilea proiect

             ]).
classDelta(loose, % profil larg, care genereaza mai multe match-uri        [
           [
            5  % delta atributes
           ,0.6% procent atributes

           ,3  % delta methods

           ,1  % delta Interfaces
           ,0.5%ProcentMatchHigh-min procent of method matches for classe to be high
           ,1  %MaxNumberOfUnmathcingMethods  - dif. max intre match si nr clase din fiecare proiect
               % pentru a declara o clasa medium
           ,0.5%ProcentMinOfMatchingMethods - min procent pt a declara o clasa med.
           ]).

methodDelta(loose,
            [
              2 %delta if
             ]).

%----------------------------------------%

% tight profile. Projects have to be closer mathces
projectDelta(tight,
             [
              0 % MaxDeltaClasses- diferenta maxima intre nr de clase din primul si cel de-al doilea proiect
```

Fig. 6.6 Fisierul profiles.pl

După cum observăm fiecare profil este compus din trei părţi în care sunt definite tolerantele :projectDelta, classDelta şi methodDelta. Denumirile sunt sugestive, în zonele acesta descriindu-se tolerantele la nivel de proiect, clase şi metode. Primul termen din fiecare clauza este numele profilului, urmat de o listă de numere. Indreptul fiecărui nume există un comentări care explică ce semnificaţie are. Unele sunt procentele minine altele sunt diferenţele maxime pentru anumite tolerante.

Momentan sunt definite două profile, dar dacă se doreşte creerea altora se poate uşor copiind un profil, modificând numele lui într-un nume unic şi tolerantele în funcţie de necesitate.

5. **Excluderea claselor din analiza.**

Facilitatea de a exclude clasele din analiza a fost amintită şi anterior. Inacest paragraf vom explică cum se poate folosi. Pentru setarea claselor ce se doresc excluse din analiză trebuie să modificăm fişierul commonClasses.pl . Pentru a exclude o clasă din analiză trebuie să creăm o nouă axiomă în acest fişier care să conţină numele acesteia ca termen. Un exemplu este prezentat în figura 6.6.

```
:-dynamic commonClass/1.


commonClass('className1.java').
commonClass('className2.java').
commonClass('className3.java').
```

Fig.6.7 Fisierul commonClasses.pl

VII. Rezultate

De-a lungul dezvoltării am urmărit să generam perechile de clase suspecte cât mai corect, să nu pierdem perechi şi să nu grupăm clase care nu seamănă şi să grupăm.Pentru a verifică am rulat unealta pe o serie de proiecte. Incontinuare vom prezentă constatările, cazurile când CloneDetector merge perfect şi limitările.

Problema comparări claselor am observat că se poate împărţi în 3 cazuri :

1. Compararea claselor cu mai multe de 1-2 metod2. Sunt clase care conţin bucăţi din logică proiectului.
2. Comparerea claselor foarte mici cu 1 maxim 2 metode. Metodele nu conţin multă logică, nu apelează alte metode, fiind de obicei gettere sau settere.
3. Compararea interfeţelor.

Vom discuta primul caz pe exemplu din figura 8.1. Inacest test se compară două proiect cu un număr mic de clase, dar fiecare clasă implementează funcţionalitate, având 4-5 metode în medie. Practic clasele acestor proiecte se încadrează în cazul 1. Clasele au suferit modificări de tipul :

- redenumire – majoritatea atributelor şi metodelor au fost redenumite. Am redenumit chiar şi clasele.
- Schimbarea ordinii. Ordinea metodelor a fost schimbată.
- Ştergerea unor porţiuni. Au fost şterge porţiuni de cod: atribute, mesaje, condtii.

```
match(25696, 'WebServer', 25950, 'AServer', high).
match(26015, 'ConnectionHandler', 25741, 'ManagerConex', high).
match(25700, 'PropertyHolder', 25691, 'ProprietateConex', high).
match(26488, 'Main', 26083, 'Main', high).


High matching classes 4 of 4
true.

2 ?-
```

Fig. 7.1 Rezultate dupa compararea unor clase mari.

În ciuda schimbărilor produse asupra codului, CloneDetector a grupat corect toate clasele. Acest lucra s-a întâmplat şi în alte cazuri. Deci putem spune că pentru clase de tipul 1 platforma se comportă exemplar.

Pentru a testa celelalte două cazuri am folosit un proiect mai mare, de 27 de clase. Am creat o copie simplă a lui, de tipul 2, în care am modificat doar ordinea claselor în pachete, obţinând astfel o baza de date diferită pentru cele două proiecte. Am rulat CloneDetector pe cele două proiecte, în prima fază folosind profilul loose. Toate cele 27 de clase au fost împerecheate cu probabilitate High ,deci verdictul dat de tool este corect, proiectele fiind copiate.

Uitându-ne totuşi mai atent am observat că 8 clase din cele 27 nu au fost grupate cu copiile lor din al doilea proiect, ci cu alte clase. Aceast lucrua a avut loc pentru cazurile 2,3. Clasele erau foarte mici, având 1 metodă sau două, cu semănătura la fel, primeau array de bytes şi returnau mesajul. Acest lucru era de aşteptat, deoarece pentru porţiuni foarte mici de cod, setul de metrici nu le poate diferenţia foarte mult iar profilul loose înlătură diferenţele între metrici prin tolerantele sale. Aceste clase nu au apeluri către alte clase deci nici compararea apelurilor nu se aplică. De asemenea pentru interfeţe folosite doar ca marcare (fără metode) sau care au puţine metode cu aceeaşi semănătură, vom avea aceeaşi problema şi este imposibil chiar în acest caz să diferenţiem intefetele pe baza metricilor. Totuşi aceste

probleme nu afectează analiza pe ansamblu aşa de mult. Pericolul ar putea fi să raportăm prea mulţi suspecţi pentru un set de proiecte, dar într-un proiect şcolar numărul de clase foarte mici şi de interfeţe este neglijabi în comparaţie cu numărul total de clase, neinfluentand verdictul. Acest lucru nu se va întâmplă, după cum vom observă şi în analiza practică.

```
match(25694, testMars, 26081, testMars, high).
match(25762, 'UnsubscribeMessage', 25768, 'ReplyPriceMessage', high).
match(25793, 'SubscribeMessage', 26173, 'SubscribeMessage', high).
match(25866, 'ReplyAddressMessage', 25904, 'ReplyAddressMessage', high).
match(25723, 'GetAddressMessage', 26144, 'UnsubscribeMessage', high).
match(25741, 'DNSUnmarshaller', 25903, 'ClientUnmarshaller', high).
match(26067, 'StockMarketInterface', 25761, 'StockMarketInterface', high).
match(26074, 'ReplyPriceMessage', 26106, 'GetAddressMessage', high).
match(26111, 'LocateServer', 25805, 'LocateServer', high).
match(26181, 'GetPriceMessage', 25877, 'GetPriceMessage', high).
match(26207, 'ClientUnmarshaller', 25848, 'ClientTransformer', high).
match(26152, 'ClientTransformer', 26123, 'DNSUnmarshaller', high).
match(26353, 'ClientStockMarket', 26051, 'ClientStockMarket', high).
match(26385, 'ServerTransformer', 26448, 'ServerTransformer', high).
match(26386, 'MessageServer', 25695, 'Marshaller', high).
match(26387, 'ServerWithRR', 26450, 'ServerWithRR', high).
match(26113, 'Requestor', 25808, 'Requestor', high).
match(26459, 'Replyer', 26522, 'Replyer', high).
match(26112, 'ByteStreamTransformer', 25807, 'ByteStreamTransformer', high).
match(26453, 'Registry', 26516, 'Registry', high).
match(25697, 'Entry', 25806, 'Entry', high).
match(26862, 'NamingServiceTransformer', 26975, 'NamingServiceTransformer', high).
match(26906, 'NamingServiceServer', 27019, 'NamingServiceServer', high).
match(26442, 'Configuration', 26505, 'Configuration', high).
match(25696, 'Message', 25691, 'Message', high).
match(25695, 'Marshaller', 26449, 'MessageServer', high).
match(25763, 'Address', 25749, 'Address', high).


High matching classes 27 of 27
true.
```

Fig. 7.2. Rezultate cu profilul loose pe un proiect mediu

O soluţie pentru problema desrisa mai sus a fost schimbare profilului. Prin schimbare pe profilul tight am redus numărul de clase grupate greşit de la 8 la 4, după cum putem vedea şi în figura 7.3. Totuşi schimbare profilului trebuie să fie făcută în cunoştinţă de cauză. Prin schimbare lui, tolerantele vor fi mai mici sau chiar zero, depistându-se copiile exacte. Se poate pierde astfel suspecţi care au schimbări în cod. O altă posibilă soluţie ar fi adăugarea unor metrici caracterisitce pentru metodele mici.

```
match(25694, testMars, 26081, testMars, high).
match(25762, 'UnsubscribeMessage', 26144, 'UnsubscribeMessage', high).
match(25793, 'SubscribeMessage', 26173, 'SubscribeMessage', high).
match(25866, 'ReplyAddressMessage', 25904, 'ReplyAddressMessage', high).
match(25723, 'GetAddressMessage', 25768, 'ReplyPriceMessage', high).
match(25741, 'DNSUnmarshaller', 25903, 'ClientUnmarshaller', high).
match(26067, 'StockMarketInterface', 25761, 'StockMarketInterface', high).
match(26074, 'ReplyPriceMessage', 26106, 'GetAddressMessage', high).
match(26111, 'LocateServer', 25805, 'LocateServer', high).
match(26181, 'GetPriceMessage', 25877, 'GetPriceMessage', high).
match(26207, 'ClientUnmarshaller', 26123, 'DNSUnmarshaller', high).
match(26152, 'ClientTransformer', 25848, 'ClientTransformer', high).
match(26353, 'ClientStockMarket', 26051, 'ClientStockMarket', high).
match(26385, 'ServerTransformer', 26448, 'ServerTransformer', high).
match(26386, 'MessageServer', 26449, 'MessageServer', high).
match(26387, 'ServerWithRR', 26450, 'ServerWithRR', high).
match(26113, 'Requestor', 25808, 'Requestor', high).
match(26459, 'Replyer', 26522, 'Replyer', high).
match(26112, 'ByteStreamTransformer', 25807, 'ByteStreamTransformer', high).
match(26453, 'Registry', 26516, 'Registry', high).
match(25697, 'Entry', 25806, 'Entry', high).
match(26862, 'NamingServiceTransformer', 26975, 'NamingServiceTransformer', high).
match(26906, 'NamingServiceServer', 27019, 'NamingServiceServer', high).
match(26442, 'Configuration', 26505, 'Configuration', high).
match(25696, 'Message', 25691, 'Message', high).
match(25695, 'Marshaller', 25695, 'Marshaller', high).
match(25763, 'Address', 25749, 'Address', high).


High matching classes 27 of 27
true.
```

Fig. 7.3. Rezultate cu profilul tight pe un proiect mediu

Proiectul CloneDetector a fost testat în mai multe circumstate şi în comparaţie cu DUDE[10]. DUDE este o unealtă realizată folosind algoritmi de comparare la nivel de şiruri de caracter.

În primă instanţă am testat cele două unelte pe teste concepute de noi iar rezultate obţinute sunt afişate în tabel.

Caz de test	CloneDetector	Dude
Clona tipul 1	Detectată	Detectată
Clona tipul 2 cu porţiuni redenumite	Detectată	Detectată
Clona tipul 2 total redenumită	Detectată	Nedetectată
Clona tipul 3 partial redenumită	Detectată	Detectată
Clona tipul 3 total redenumită	Detectată	Neraportată
Proiecte diferite	Neraportată	Neraportată
Proiecte mari diferiţe	Neraportată + timp mare de executie	Neraportată

Analizând aceste proiecte am observat că platforma DUDE depistează repede orice tip de clonă, dacă există bucăţi de cod neredenumite. Dacă clonă a fost total redenumită, DUDE nu va mai depistă proiectul plagiat.

CloneDetector a reuşit să depisteze clonele pentru testele create, indiferent de numărul redenumirilor.

Ultimele două teste au fost create pentru a vedea cum interpretează unelte proiectele total diferite. S-a observat că ambele unelete nu au raportat proiectele că fiind copii, lucru corect. În ultimul caz au fost date sper comparare proiecte mari, nu şcolare, pentru a testa timpii de execuţie. În acest caz, pentru un proiect de câteva mii de clase , DUDE a reuşit să îl analizeze în aproximativ 30 de secunde timp mai bun faţa de platforma noastră, care a terminat inspecţia în 3 minute. Lucrul acesta era de aşteptat, DUDE fiind integrat în alte unelte de analiză software de mari dimensiuni, în vreme ce programul nostru a fost gândit pentru proiecte şcolare.

În cea de-a doua etapă am testat cele două proiecte pe un set de teme de laboratorul de Proiectarea Arhitecturală a Software Complexe. Au fost analizate 16 proiecte cu cele două unelte. Analiza proiectelor a durat mai puţin de 30 de secunde cu CloneDetector şi aproximativ 10 cu DUDE, timpi foarte buni pentru ambele.

DUDE a raportă 1 pereche de proiecte copiate, depistând trei porţiuni de cod identice de lungime 7,8 şi 9 rânduri. În urma analizării proiectelor s-a confirmat că acestea au fost copiate.

Folosind CloneDetector cu profilul mai larg (acceptă diferenţe mai mari între proiecte) au fost depistate 2 perechi de proiecte. Prima a fost cea depistată şi de DUDE. A două pereche a fost analizată şi deşi structural proiectele erau asemănătoare ele nu au fost copiate.

Schimbând profilul pe cel strict, platforma a raportat doar prima pereche de clone, cea de-a două fiind exclusă.

VIII. Concluzii

CloneDetector a fost conceput pentru a depista clone în rândul programelor şcolare. Pentru a satisface aceste cerinţe cât mai bine şi a uşura munca utilizatorilor, au fost dezvoltate diverse facilităţi, fără a afecta scalabilitatea algoritmului.

O facilitate menţionată şi anterior este aceea de a exclude unele clase, pachete sau foldere din analiză. Aceasta se dovedeşte foarte utilă în cazul în care toate proiectele au porţiuni de cod date de la început.

După cum am menţionat, setarea procentelor sau variaţiilor minime/maxime pentru a declara două clase copiate sunt esenţiale. Variaţii mici determină raportarea doar a proiectelor foarte asemănătoare, în vreme ce variaţiile mari lărgesc aria de raportare. în funcţie de proiectele propuse şi direcţiile de dezvoltare date de profesori unele proiecte pot varia mai mult sau mai puţin. Pentru a putea analiză diferit în funcţie de tema dată, am creat două profile de analiză unul larg şi al doilea mai exact. Aceste profile pot fi selectate la rulare sau noi profile pot fi uşor definite în fişierul profiles.pl.

Varianta actuală a platformei poate fi uşor modificată şi îmbunătăţită în funcţie de nevoile apărute sau a constatărilor utilizatorilor, prin adăugarea unor noi filtre. Aceasta se poate face relativ uşor şi cu modificări puţine.

Un avantaj dobândit prin folosirea platformei SWI -Prologul este facilitatea acesteia de a lucra multi-threading, sporind viteza de analiză a proiectelor.

Plugin-ul JTransformer are un rol semnificativ dar momentan nu este integrat cu platforma creată. Acest impediment se poate rezolva în viitor integrând CloneDetector cu Eclipe, ca plugin. Până atunci soluţia comodă pentru un profesor ar fi de a primi de la studenţi odată cu tema şi fişierul qlf generat.

Pe viitor CloneDetector, poate fi integrat cu serviciile web, dupa modelul marilor universitati Standford, MIT s.a. Astfel profesorii ar avea acees la unealta onlin daca isi fac un cont. Pentru evaluare ar putea sa trimita setul de proiecte primind raspunsul pe email.

CloneDetector se doreşte a fi un tool uşor de folosit şi eficient în depistarea proiectelor software copiate. În urma testelor realizate asupra unor proiecte concrete, suntem încrezători în utilitatea acestei platforme.

S-a observat că programele ce folosesc algoritmi de comparare a şirurilor de caractere, deşi foarte rapide, pot depista doar clonele de tipul 1, eventual 2 şi 3 dacă redenumirile nu sunt complete. Instrumentele ce se bazează pe grafuri de apeluri pot semnala toate tipurile de clone dar datorită algoritmilor complexi, au o durată de procesare mai mare. CloneDetector poate depista toate clonele de tipul 1 şi 2 indiferent de numărul de redenumiri efectuate, semnalând şi clone de tipul 3 cu probabilitatea lor de a fi copiate.

Bibliografie

[1] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, Ettore Merlo, “Comparison and Evaluation of Clone Detection Tools”, vol. 33, no.9, IEEE Transaction on Software Engineering, 2007, pp.1–3.
[2] S. Ducasse, M. Rieger, and S. Demeyer, “A Language Independent Approach for Detecting Duplicated Code,” Proc. Int’l Conf. Software Maintenance (ICSM ’99), 1999.
[3] B.S. Baker, “On Finding Duplication and Near-Duplication în Large Software Systems,” Proc. Second Working Conf. Reverse Eng.,L. Wills, P. Newcomb, and E. Chikofsky, eds., pp. 86-95, July 1995.
[4] K. Kontogiannis, R.D. Mori, E. Merlo, M. Galler, and M. Bernstein, “Pattern Matching for Clone and Concept Detection,” Automated Software Eng., vol. 3, nos. 1-2, pp. 79-108, June 1996.
[5] Nghi Troung, Paul Roe, Peter Bancroft, “Static Analysis of Student’s Java Programs”, Queensland University of Technology, Australia,pp.4-5
[6] J. Krinke, “Identifying Similar Code with Program Dependence Graphs,” Proc. Eighth Working Conf. Reverse Eng. (WCRE’ 01), 2001.
[7] R. Komondoor and S. Horwitz, “Using Slicing to Identify Duplication în Source Code,” Proc. Int’l Symp. Static Analysis, pp. 40-56, July 2001.
[8] Peter Bulychev, Marius Minea, “Duplicate code detection using anti-unification”, pp. 2-3
[9] Proiectul JTransformer. [Online].Tutorial 2007. Se gaseste la adresa de internet: http://roots.iai.uni-bonn.de/research/jtransformer/
[10] Radu Marinescu, [Online] http://intooitus.com
[11] Paul Clough ”Plagiarism în natural and programming languages: an overview of current tools and technologies”, University of Sheffield, Iulie 2000
[12] Balazinska, M., Merlo, E., Dagenais, M., Lague, B., Kontogiannis, K. : “Measuring clone based reengineering oportunities”, IEEE Symposium on Software Metrics, 1999, pp. 292 – 303.
[13] Kapser, C., Godfrey, M.W.: “Toward a taxonomy of clones în source code: A case study. In: Evolution of Large Scale Industrial Software Architectures.” (2003)
[14] Kapser, C., Godfrey, M.: “A taxonomy of clones în source code: The reengineers most wanted list. In: Working Conference on Reverse Engineering “, IEEE Computer Society Press (2003)
[15] Tudor Girba, Uni. Bern, www.tudorgirba.com
[16] Xinran Wang, Yoon0Chan Jhi, Sencun Zhu, Peng Liu: “Detecting Software Theft via System Call Based Birthmarks”, Computer Security Applications , 2009
[17] M. H. Halstead, “Elements of software science”, North Holland, New York, 1977.
[18] T. J. McCabe, A complexity measure, IEEE Transactions on Software Engineering, SE-2 (4), pp(308-320), 1976.
[19] Xin Chen, Brent Francia, Ming Li, Brian Mckinnon, Amit Seker,” Shared Information and Program Plagiarism Detection”, Universitatea din California 2003.
[20] L. M. Ottenstein, “Quantitative estimates of debugging requirements”, IEEE Transactions of Software Engineering, Vol. SE-5, pp(504-514), 1979.
[21] G. H. Gonnet and R. Baeza-Yates, An Analysis of Karp-Rabin String Matching Algorithm, Information Processing Letters 34, pp(271-274), 1990.
[22] Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, Ettore Merlo “ Comparison and Evaluation of Clone Detection Tools”, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, Nr. 9,Septembrie 2007
[23] Paul Clough, Plagiarism în natural and programming languages: an overview of current tools and technologies, University of Sheffield, Iulie 2000