Lucrare de diplomă

Instrument software pentru detecţia carenţelor de proiectare în sisteme orientate-obiect

Ciprian-Bogdan Chirilă

Universitatea *"Politehnica"* Timişoara
Facultatea de Automatică şi Calculatoare
Departamentul de Calculatoare şi Inginerie Software

Iunie, 2001

Conducători ştiinţifici:
Prof. Dr. Ing. Ioan Jurca
Asist. Ing. Radu Marinescu

Rezumat

Datorită numărului mare de sisteme orientate-obiect monolitice şi inflexibile şi al costurilor acestora este necesară o modalitate de a le reproiecta spre a fi întreţinute sau reutilizate. Pentru a putea fi reproiectate trebuiesc eliminate carenţele de proiectare. Acest lucru este realizabil cu ajutorul strategiilor de detecţie în mod sistematic, scalabil şi repetabil.

În acest articol vom prezenta un instrument software care modelează strategiile de detecţie prin intermediul metricilor şi al operatorilor statistici, aplică aceste strategii pe sisteme moştenite şi oferă introspecţie în codul sursă exact la entitatea suspectă.

Motto

Assiduus usus uni rei deditus et ingenium et artem saepe vincit.

Cuprins

1 Introducere **6**
- 1.1 Motivaţie 6
  - 1.1.1 Problemele strategiilor de detecţie 7
  - 1.1.2 Soluţii oferite sistemelor industriale 7
- 1.2 Contribuţie 8
- 1.3 Structură 8

2 Fundamente teoretice **10**
- 2.1 Entităţi 10
- 2.2 Metrici 11
- 2.3 Elemente de statistică 11
- 2.4 Tehnologia programării orientate-obiect 14

3 Stadiul detecţiei **15**
- 3.1 Sisteme moştenite 15
- 3.2 Soluţii oferite sistemelor moştenite 15
- 3.3 Carenţe de proiectare 15
- 3.4 Un design defectuos 16
  - 3.4.1 Rigiditate 16
  - 3.4.2 Fragilitate 16
  - 3.4.3 Imobilitate 16
  - 3.4.4 Vâscozitate 16
- 3.5 “Bad smells” în cod 17
- 3.6 Tipare de proiectare 20
  - 3.6.1 Ce sunt tiparele de proiectare 20
  - 3.6.2 Taxonomia tiparelor 21
- 3.7 Detector de carenţe de proiectare 21

4 Detecţia automată a carenţelor de proiectare **22**
- 4.1 Metamodel 23
- 4.2 Metrici 27

4.2.1 Greutatea unei clase 27
4.2.2 Numărul de atribute publice 27
4.2.3 Numărul de metode accesor 27
4.3 Operatori statistici - outliers 28
4.3.1 Problema zerourilor 28
4.4 Strategie de detecţie - limbajul SOD 28
4.4.1 Gramatica 28
4.4.2 Analizorul lexical 30
4.4.3 Analizorul sintactic 30
4.4.4 Generatorul de arbore 31
4.4.5 Arborele de detecţie 31
4.5 Exemplu de strategie - “Data classes” 31
4.5.1 Prezentarea strategiei 31

5 Instrumentul software - PRODEOOS **34**
5.1 Structură 34
5.2 Detectorul de carenţe 34
5.2.1 Structură 34
5.2.2 Metrică simplă 36
5.2.3 Metrică compusă 37
5.2.4 Operatori logici 37
5.2.5 Operatori statistici 38
5.2.6 Rezultate 38
5.2.7 Vizualizare 39
5.3 Managerul de conexiune 39
5.3.1 Structură 39
5.3.2 Vizualizare 40
5.4 Vizualizarea surselor 41
5.4.1 Vizualizare 42
5.5 Monitorul SQL 43
5.5.1 Structură 43
5.5.2 Vizualizare 43
5.6 Generatorul de rapoarte 44
5.6.1 Structură 44
5.7 Implementarea soluţiei problemei zerourilor 46
5.8 Gestionarea documentelor 46
5.9 Studiu de caz 46

6 Concluzii. Proiecte de viitor **55**
6.1 Concluzii 55
6.2 Proiecte de viitor - legate de instrumentul software 55

A Tabela de accese 58

B Tabela de apeluri 61

C Tabela de clase 65

D Tabela de declaraţii 67

E Tabela de funcţii 71

F Tabela de moşteniri 75

Listă de figuri

1.1 Soluţii oferite sistemelor industriale 8

2.1 Tehnica de analiză statistică **BoxPlots** 12
2.2 Aplicarea tehnicii **BoxPlots** pentru mărimea KLOC 13
2.3 Aplicarea tehnicii **BoxPlots** pentru mărimea MOD 14
2.4 Aplicarea tehnicii **BoxPlots** pentru mărimea FD 14

4.1 Abordare bazată pe metrici 22
4.2 Cele 3 dimensiuni ale arborelui de detecţie 32

5.1 Structura Prodeoos 35
5.2 Structura arborelui de detecţie 36
5.3 Structura metricii simple 36
5.4 Structura metricii compuse 37
5.5 Ierarhia operatorilor logici 37
5.6 Ierarhia operatorilor statistici 38
5.7 Modelul rezultatelor 39
5.8 Instrumentul software - Prodeoos 40
5.9 Structura managerului de conexiune 41
5.10 Structura vizualizatorului de surse 41
5.11 Vizualizatorul de surse 42
5.12 Monitorul SQL 44
5.13 Structura generatorului de rapoarte 45
5.14 Modelare entităţi nule 46

Listă de tabele

4.1 Tabela acceselor 23
4.2 Tabela apelurilor 24
4.3 Tabela claselor 24
4.4 Tabela declaraţiilor 25
4.5 Tabela funcţiilor 26
4.6 Tabela moştenirilor 26

A.1 Tabela detaliată a acceselor 58

B.1 Tabela detaliată a apelurilor 61

C.1 Tabela detaliată a claselor 65

D.1 Tabela detaliată a declaraţiilor 67

E.1 Tabela detaliată a funcţiilor 71

F.1 Tabela detaliată a moştenirilor 75

Capitolul 1

Introducere

1.1 Motivaţie

Tehnologiile de programare au o evoluţie foarte dinamică datorită *creşterii complexităţii programelor*[1, Schi97].

O dată cu creşterea dimensiunii proiectelor software a devenit tot mai clar că pentru a controla complexitatea, un rol substantial îl joacă *abstractizarea datelor* [2, Booc94] şi că în acest scop programarea structurată este ineficientă.

Motivaţia care stă la baza acestei lucrări este aceea de a analiza modul în care metricile software pot da naştere la strategii de detecţie şi modul în care aceste strategii pot fi automatizate. Motivul pentru care folosim metricile este acela că avem de a face cu milioane de linii de cod, care nu pot fi inspectate manual, context în care automatizarea se impune ca o necesitate. Scopul acestei teze nu diferă de scopul ingineriei software, care este acela de a creşte controlabilitatea procesului de dezvoltare a sistemelor software. Scopurile concrete al tezei se pot materializa prin:

- creşterea calităţii produselor software
- anticiparea şi reducerea costurilor de întreţinere.

Visele anilor 80 În anii 80 a avut loc o explozie puternică în arta tehnologiei obiectelor, ce a determinat începerea construirii unui număr mare de sisteme software bazate pe această tehnologie.

Avantajele se reflectau în:

- *creşterea calităţii software-ului* datorită noului nivel de abstractizare introdus de tehnologie
- *scăderea timpul de dezvoltare* prin reutilizări pe diferite nivele.

Realităţile anilor 90 Sistemele construite nu au fost deloc la nivelul aşteptărilor din punct de vedere al fiabilităţii. Ele s-au dovedit a fi sisteme pe scară largă:

- *inflexibile*: nu pot fi adăugate cu uşurinţă funcţionalităţi noi
- *monolitice*: nu există o structurare a funcţionalităţii sistemului bazată pe componente
- *greu de întreţinut*: încercarea de a aduce noi modificări se soldează cu un lanţ nesfârşit de ajustări în multiple locuri.

Aceste carenţe de proiectare, dacă ar putea fi remediate s-ar evita reproiectarea sistemelor economisind resurse de timp, resurse umane, resurse financiare. În acest punct intervine strategia de detecţie, care ce face: caută părţile din sistem afectate de carenţele de proiectare. Dacă facem o paralelă cu rezistenţa materialelor din domeniul construcţiilor civile atunci pentru a mai adauga un etaj la o vilă căutăm acei pereţi care sunt slabi, aceia care s-ar prabuşi în momentul în care am construi etajul. Motivul pentru care facem acest lucru este simplu: o dată partea sistemului localizată, va fi întărită sau reproiectată.

1.1.1 Problemele strategiilor de detecţie

Abordarea curentă a problemei detecţiei diferă foarte mult de la sistem la sistem deci este *dependentă de context*. La ora actuală ea se face *manual* scăzându-i potenţialul.

Dintre dificultăţile cu care se zbate problema detecţiei amintim:

- *cronofagă*: imaginaţi-vă cât ne-ar lua dacă am dori să analizăm posibilele probleme de proiectare într-un proiect de dimensiuni medii de 200 de clase
- *irepetabilă*: pentru că analiza depinde de la proiect la proiect
- *nescalabilă*: nu putem aplica raţionamentul folosit, pentru proiecte vaste.

1.1.2 Soluţii oferite sistemelor industriale

Vom comenta puţin această figură punând accent pe necesitatea de *reproiectare*. Se pune problema a ceeea ce se poate face cu un astfel de sistem software. Dacă avem de a face cu un sistem a cărui valoare economică este scăzută atunci în cazul în care el nu este flexibil se poate opta chiar pentru abandonarea lui. Problema este critică în cazul sistemelor cu valoare economică

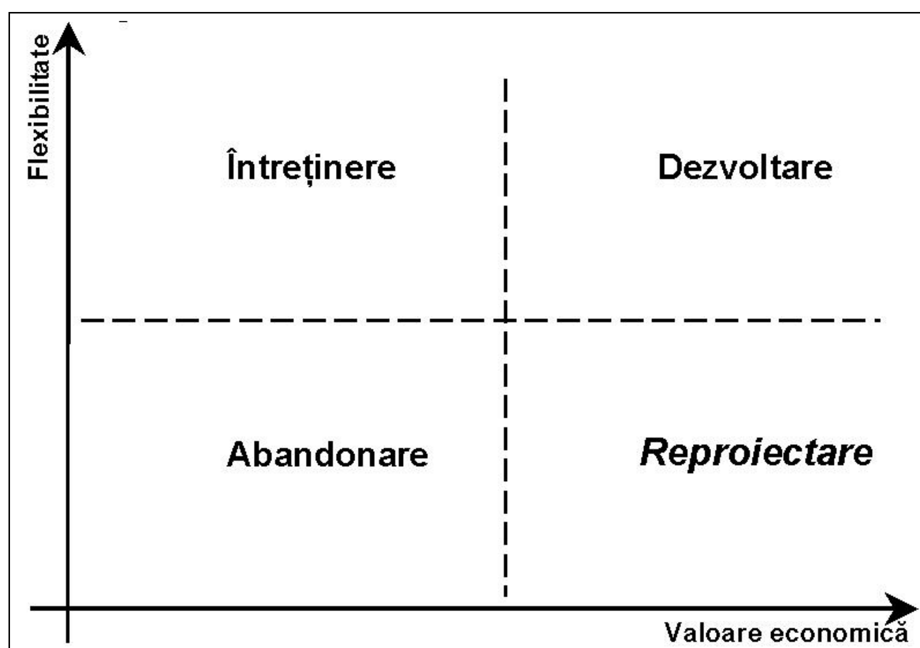

Figura 1.1: Soluţii oferite sistemelor industriale

crescută. Aici se pune problema dezvoltării, chiar în condiţiile unei inflexibilităţi cronice. Aici există o necesitate imperioasă pentru *detecţia carenţelor de proiectare*, în vederea eliminării lor şi a dezvoltării ulterioare.

1.2 Contribuţie

Contribuţia adusă este legată de automatizarea procesului de detecţie. Dacă pñă acum el se făcea manual şi era irepetabil, acum el se poate face îmod sistematic şi repetabil.

1.3 Structură

Lucrarea de faţă se împarte în şase capitole.

Primul capitol se referă la motivaţia necesităţii instrumentului software, contribuţia adusă şi structură tezei.

Capitolul 2 ne va introduce în lumea metricilor şi al operatorilor statistici. Tot acolo vom mai putea afla detalii despre design-ul sistemelor software şi de ce maladii suferă ele.

În capitolul 3 vă voi prezenta starea actuală ştiinţei din acest domeniu.

Capitolul 4 este cel mai vast şi aici se găseşte abordarea bazată pe metrici a detecţiei.

Capitolul 5 va discuta despre detaliile tehnice de realizare a instrumentului software. Sunt făcute şi studii de caz care să evidenţieze avantajul motorului de detecţie.

Capitolul 6 concluzionează teza punctând câştigurile obţinute.

Capitolul 2

Fundamente teoretice

Acest capitol prezintă câteva elemente de bază ale teoriei măsurării aplicată în domeniul software. Prima secţiune se referă la entităţile software cu care lucrăm, conturându-le mai clar rolul în abordarea noastră. A doua secţiune abordează elemente teoretice referitoare la metrici software. În secţiunea a treia sunt prezentate elemente de statistică care vor fi modelate ulterior în instrumentul software.

Măsurarea este definită ca fiind procesul prin care numere sau simboluri sunt asignate atributelor unor entităţi din lumea înconjurătoare, astfel încât să fie descrie conform unor reguli bine definite. Definiţia măsurării necesită câteva explicaţii şi câteva definiţii. Concepte folosite în definiţie ca: *entităţi*, *atribute* şi *reguli* vor fi explicate în continuare.

2.1 Entităţi

Entitatea se poate defini ca fiind subiectul procesului de măsurare. Prin entităţi înţeleg:

- metode;
- clase;
- subsisteme.

Un *atribut* este o trăsătură sau o proprietate a unei entităţi. Un atribut al unei metode poate fi complexitatea ciclomatică, iar al unei clase poate fi numărul de metode publice. Deoarece asignarea de numere şi simboluri trebuie să păstreze observaţiile intuitive şi empirice legate de atribute şi entităţi. În cele mai multe situaţii un atribut poate avea mai multe semnificaţii intuitive pentru persoane diferite. De accea se defineşte un model. Un *model*

este expresia unui punct de vedere privind entităţile măsurate. Din momentul în care un model a fost ales, se pot determina relaţiile dintre atribute care descriu entitatea măsurată. Necesitatea de a defini modele potrivite este relevantă în procesul de măsurare din ingineria software.

2.2 Metrici

Metricile sunt funcţii care caracterizează diferite entităţi ale unui sistem printr-un număr, fără a pune sistemul în execuţie. Ele oferă indicaţii referitor la complexitatea programului analizat: comprehensibilitate, claritate, care sunt posibilităţile de întreţinere, siguranţă. Metricile se aplică în domenii diferite şi acţiunile lor sunt foarte diversificate. Diversele domenii ale metricilor sunt:

- *textuale*, bazate pe numărarea operanzilor şi a operatorilor folosiţi de program. Face posibilă înţelegerea problemelor legate de înţelegerea codului spre exemplu prin faptul că există repetiţii de cod sau o funcţie, un bloc de instrucţiuni sunt prea vaste.

- *structurale* folosite spre a cunatifica complexitatea funţiei de control al fluxului pentru a evalua efortul necesar înţelegerii algoritmului sau spre a testa diferitele căi care îl traversează.

- *fluxuri de date* care numără parametri de intrare a unei funţii, sau numărul de variabile, sau macro-uri folosite de program va folosi estimării efortului înţelegerii şi modificării programului.

- *comentarii*, metricile nu vor putea măsura relevanţa comentariilor scrise în program, ele pot doar verifica dacă sunt prezente şi suficiente pentru înţelegerea codului.

2.3 Elemente de statistică

Din experimente şi studii de caz rezultă date care apoi trebuie investigate şi analizate foarte riguros.

Analiza datelor implică câteva activităţi şi presupuneri:

- avem un număr de măsurători a unuia sau mai multor atribute ale entităţilor software. Setul de măsuratori îl vom numi **set de date**

- ne aşteptăm ca entităţile software să fie comparabile. De exemplu putem compara module din acelaşi sistem software examinând diferenţele sau similarităţile datelor.

Design-ul *investigaţiilor* trebuie luat în seamă prin alegerea tehnicilor de analiză. De asemenea complexitatea analizei poate influenţa design-ul ales.

O tehnică simplă de analiză este *Box plots*. Deoarece datele obţinute din măsurători software nu sunt totdeauna normal distribuite şi valorile măsurăte nu se găsesc pe o scară liniară, este de preferat să se folosească valoarea mediană şi valorile din sferturile setului de date, pentru a putea putea defini locaţia centrală şi dispersia valorilor componente, în locul mediei şi al varianţei. Tehnica *Box plots* se bazează pe următoarele elemente de statistică: *mediana*, *sfertul inferior* şi *sfertul superior*. *Mediana* **m** este valoarea care are poziţia în mijlocul setului de date, jumătate din valori sunt mai mici decât mediana, iar cealaltă jumătate de valori este mai mare. *Sfertul superior* **u** este mediana valorilor mai mari decât **m**,şi *sfertul inferior* **l** este mediana valorilor mai mici decât **m**. Deci **l**, **m** şi **u** vor împărţi setul de date în patru părţi. Se defineşte *distanţa* **d** ca fiind lumgimea de la sfertul inferior la sfertul superior **d=u-l**. Se definesc apoi capetele distribuţiei:

- *inferior* $l - 1.5d$
- *superior* $u + 1.5d$.

Este necesar ca aceste valori să fie truncate spre valori existente în setul de date. Valorile aflate deasupra capătului superior şi dedesubtul capătului inferior sunt considerate ca fiind ieşite din spectrul normal.

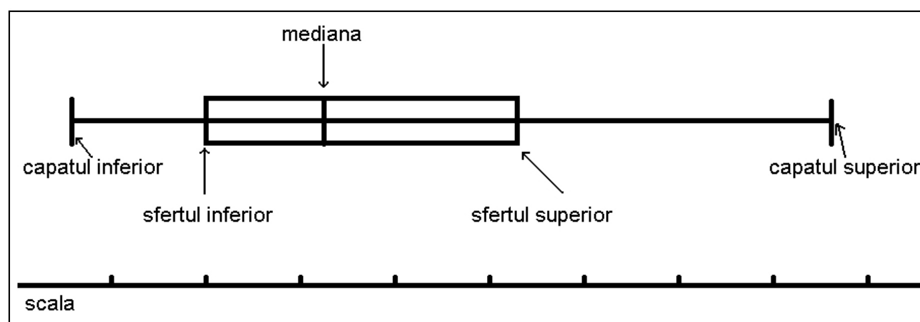

Figura 2.1: Tehnica de analiză statistică **BoxPlots**

Exemplu

System	KLOC	MOD	FD
A	10	15	36
B	23	43	22
C	26	61	15
D	31	10	33
E	31	43	15
F	40	57	13
G	47	58	22
H	52	65	16
I	54	50	15
J	67	60	18
K	70	50	10
L	75	96	34
M	83	51	16
N	83	61	18
P	100	32	12
Q	110	78	20
R	200	48	21

În tabel avem valorile obţinute prin măsurarea mai multor atribute a 17 sisteme software. Pentru fiecare sistem se fac trei măsurători: se măsoară miile de linii de cod (KLOC), mărimea medie a modulelor (MOD) şi numărul de erori găsite într-un bloc de 1000 de linii (densitatea erorilor).

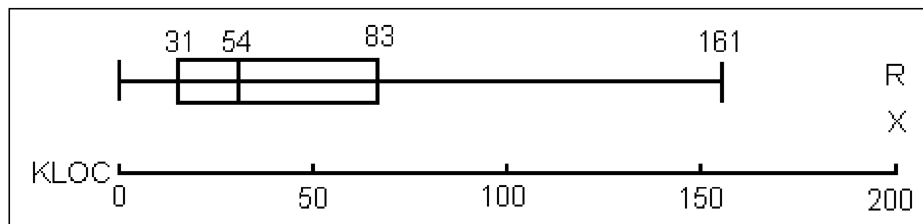

Figura 2.2: Aplicarea tehnicii **BoxPlots** pentru mărimea KLOC

Metricile MOD şi FD au aceleaşi valori extreme. Sistemele D,L şi A au o densitate anormală de erori, şi au de asemenea dimensiune anormală pentru module. D şi A au valoarea MOD foarte joasă, în timp ce L o are foarte înaltă. Investigaţiile ulterioare trebuie să identifice mai clar relaţia dintre MOD şi FD, dar datele inţiale par să confirme ideea că un sistem ar trebui să fie compus din module nici prea mici şi nici prea mari.

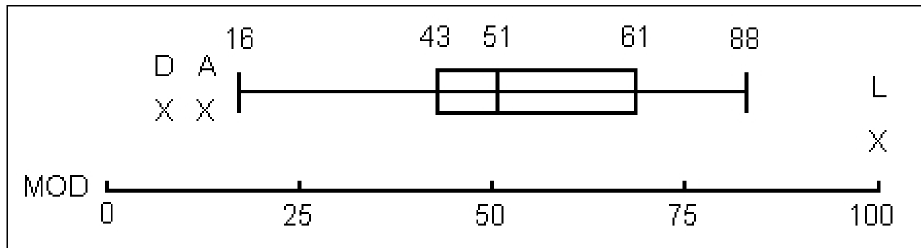

Figura 2.3: Aplicarea tehnicii **BoxPlots** pentru mărimea MOD

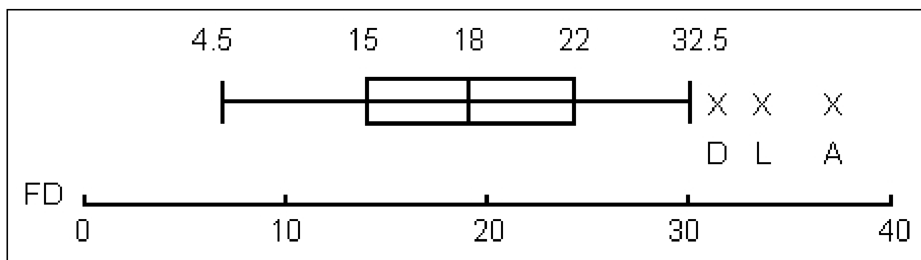

Figura 2.4: Aplicarea tehnicii **BoxPlots** pentru mărimea FD

2.4 Tehnologia programării orientate-obiect

Printre limbajele care implementează această tehnologie sunt: Java, C++, Eifell, SmallTalk. Java este un limbaj pentru dezvoltarea aplicaţiilor intranet şi Internet, simplu, *orientat pe obiecte*, interpretat, robust, securizat, independent de arhitectură, portabil, performant, multifir şi dinamic. Am ales acest limbaj pentru dezvoltarea instrumentului software deoarece are la bază toate conceptele menţioante mai sus şi pentru ca distribuţia produsă de firma Sun Microsystems este gratuită. Deoarece limbajul Java prezintă similarităţi cu limbajul C++, este simplu de utilizat de către programatori. Un avantaj major pentru care am optat pentru Java este mecanismul de alocare automată a memoriei. Altă facilitate este legată de programarea concurentă, care are ca şi suport firele de execuţie. Trăsătura de bază care-l face puternic este *orientarea pe obiecte*. Moştenirea multiplă nu este suportată deoarece beneficiile aduse de ea nu sunt justificate. Limbajul este unul distribuit, facilitate suportată de un pachet de clase bazat pe socluri, proiectat în acest scop.

Capitolul 3

Stadiul detecţiei

3.1 Sisteme moştenite

În industria mondială la ora actuală există sisteme software monolitice, inflexibile care sunt realizate în tehnologia programării orientate pe obiecte, pe care le voi numi *sisteme moştenite*. Costurile acestor sisteme s-au ridicat la sume înalte, ceea ce ne duce cu gândul la necesitatea imperioasă de a exista posibilitaţi spre a refolosi cât mai mult cu putinţă din structura aceste sisteme.

3.2 Soluţii oferite sistemelor moştenite

După cum arată graficul din figură se pot adopta patru direcţii de mers:

Vom comenta puţin acest grafic. În cazul unor sisteme flexibile, indiferent de costuri, ele se pot îmbunătăţi cu noi facilităţi. Problema critică se pune în cazul sistemelor rigide. Şi aici soluţiile se ramifică dicotomic: pe cele de costuri mici le putem abandona, deoarece este mult mai ieftin să le rescriem, iar pe cele de costuri mari nu avem decât să le *reproiectăm*.

3.3 Carenţe de proiectare

Pentru a putea realiza o exploatare a funcţionalităţilor sistemelor moştenite este necesar ca acestea să fie realizate strict conform principiilor programării orientate pe obiecte. În caz contrar sistemele vor fi confruntate cu o serie de *carenţe de proiectare*. În subcapitolul următor voi prezenta câteva dintre tumorile cu care se confruntă un sistem software în fazele de reproiectare.

3.4 Un design defectuos

3.4.1 Rigiditate

Rigiditatea [4, Mart00] este tendinţa software-ului de a se supune cu foarte mari dificultăţi modificărilor, chiar unor modificări simpliste. Fiecare modificare cauzează un lanţ de schimbări în modulele dependente. Nu există nici o modalitate de a estima timpul necesar efectuării modificărilor.

3.4.2 Fragilitate

Fragilitatea [4, Mart00] este tendinţa software-ului de a defecta în multe locuri de fiecare dată când este modificat. Modificând într-un anumit loc, apar dificultăţi în alte locuri care nu au legătură conceptuală cu primul. Proiectele care suferă de o astfel de tumoare angajează inginerii spre o muncă infernală careia nu i se întrezăreşte finalul. Uneori managerii de frica unui astfel de cerc vicios nu autorizează modificări dacât în zonele critice, cele necritice fiind lăsate nemodificate. La funcţionarea unui astfel de sistem pot apărea sitiaţii neprevăzute, astfel echipa scapă de sub control proiectul, afectând încrederea beneficiarului şi succesul proiectului.

3.4.3 Imobilitate

Imobilitatea [4, Mart00] reprezintă incapacitatea unui sistem software de a pune la dispoziţie componente pentru reutilizare, altor sisteme software sau chiar sieşi. Există situaţii în care pentru construirea unui nou sistem este nevoie de o componentă existentă în alt sistem, dar datorită dependenţelor puternice între componentele sistemului secund, este mai ieftină rescrierea componentei decât refolosirea ei.

3.4.4 Vâscozitate

Vâscozitatea [4, Mart00] este de două tipuri: vâscozitate de proiectare, şi vâscozitate de mediu. Prima se manifestă prin faptul că o modificare este mult mai usor de făcut în detrimentul filozofiei care guvernează ideile de bază ale sistemului construit. Cea de-a doua dimensiune pe care o cunoaşte vŝcozitatea este cea referitoare la mediu. Dacă se lucrează cu un mediu neperformant, care fură timp semnificativ de compilare, inginerii vor fi tentaţi să facă astfel de modificări care să nu ducă la recompilări masive.

3.5 “Bad smells” în cod

În cartea lui Martin Fowler [5] sunt prezentate, cauzele care conduc la refactorizări şi ce metode de refactorizare se impun a fi aplicate. Vom folosi aceste informaţii pentru a crea pe baza lor unele strategii de detecţie şi nu se va insista asupra soluţionării lor.

Duplicate Code Dacă există aceeaşi structură de cod în mai multe locuri cu siguranţă că programul va fi mai bun dacă găsim o metodă prin care să unificăm structurile. Cea mai simplă problemă a duplicării de cod este aceea când avem aceeaşi expresie în două metode ale aceleaşi clase. Soluţia este crearea unei metode noi care să conţină codul duplicat şi care să fie apelată de metodele anterioare. Duplicarea poate apărea în metodele a două clase ce sa găsesc în relaţie de moştenire. Aici soluţia este extragerea metodelor din ambele clase, deplasarea unor eventuale câmpuri spre superclasă. În cazul duplicării codului în clase între clase care nu există nici o relaţie, se procedează la crearea unei clase noi care să conţină codul o singură dată şi care să fie instanţiată pentru apel.

Long Method Programele orientate-obiect trainice sunt cele care au metode scurte. Cei care sunt neexperimentaţi cu tehnologia obiectelor consideră că în program nu are loc niciodată calculul propriu zis şi că totul se reduce la un lanţ lung de delegări. Metodele mici care implementează delegările sunt de fapt foarte valoroase peste câţiva ani când se doreşte adăugarea de noi facilităţi. Cu cât o procedură este mai lungă cu atât este mai greu de înţeles. Cheia spre a face procedurile mici să fie comprehensibile este aceea de a da un nume sugestiv, legat de scopul ei şi nu modalitatea în care operează. Dacă se aleg nume sugestive, nu mai trebuie să se inspecteze corpul metodei.

Large Class Când o clasă are foarte multe funcţionalităţi, instanţele ei sunt foarte dese şi de aici la duplicarea de cod mai este doar un singur pas.

Long Parameter List În programarea procedurală am fost instruiţi să trimitem ca parametri toate datele de care rutina are nevoie. Astfel listele de parametri devin lungi, dar este de preferat această variantă este mult mai bună decât cea cu variabile globale. În cazul obiectelor situaţia se schimbă: dacă e nevoie de o dată poţi cere obiectului care se ocupă de ea să o obţină. Nu mai este nevoie să se transmită ca parametri liste lungi de date deoarece o parte importantă dintre acestea sunt conţinute în clasă sub formă de membri.

Listele de parametri sunt greu de înţeles şi se schimbă ori de câte ori mai este nevoie de o dată în plus.

Divergent Changes Software-ul pe care îl scriem trebuie să fie flexibil adică să fie "soft". Dorim să putem găsi ori de câte ori dorim puncte în care să putem face modificări prin care să se adauge noi funcţionlităţi. Dacă nu putem găsi aceste puncte atunci proiectul însemnă că este foarte rigid. Schimbările divergente apar când o numită clasă se modifică în direcţii diferite sub diverse motive. Soluţia care este se recomandă este accea de a reflecta modificările în clase diferite.

Shotgun Surgery Aceast "miros" de referă la schimbările care se pot declanşa în cascadă când facem modificări într-un anumit loc. Pericolul este acela de a omite schimbări importante printre cele mărunte şi numeroase. Schimbările divergente se referă la o clasă care suferă multe schimbări, pe când "Shotgun Surgery" reprezintă o schimbare care modifică clase multiple.

Feature Envy Un alt miros este acela care se degajă când avem de a face cu o metodă care este interesată de o clasă, alta decât cea în care este plasată. De regulă această metodă face apeluri prin intermediul unor instanţe ale claselor din care ar trebui să facă parte. Regula fundamentală este accea de a pune laolaltă lucrurile care se schimbă împreună.

Data Clumps Datele se pot compara cu copii care trăiesc şi se joacă împreună. Uneori datele apar răsfirate: apar ca şi câmpuri individuale sau ca şi parametri separaţi. Ar fi de dorit ca datele să vieţuiscă împreună într-un obiect nou. De aici încolo trebuie analizată posibilitatea mutării comportamentului în acelaşi obiect.

Primitive Obsession Mediile de programare au de regulă două tipuri de date unul primitiv şi altul structurat care se bazează pe primul. Mulţi programatori au reţineri în a folosi obiecte mici pentru sarcini simple. Toate datele primitive se pot înlocui cu obiecte.

Switch Statements Una dintre simptomele unui program orientat obiect este aceea a lipsei instrucţiunilor *switch*. Switch-urile introduc cod duplicat, deoarece ele apar prin codul sursă în mod repetat. Dacă se doreşte introducerea unei noi clauze în switch, vor trebui modificate toate celelalte switch-uri din program. Ca alternativă la această soluţie incomodă este *polimorfismul*.

Parallel Inheritance Hierarchies Există situaţii în care specializarea unei clase poate implica necesitatea specializării şi a altei clase. Eliminarea neajunsului se poate face prin asigurarea unei referinţe în ierarhia paralelă.

Lazy Class În urma refactorizărilor o clasă poate slăbi în funcţionalitate, de accea se procedează la eliminarea ei. Unele clase pot apărea în cod doar printr-un schelet declarativ, reprezentând o idee abandonată. Deoarece înţelegerea şi întreţinerea claselor este costisitoare se procedează la eliminarea lor.

Speculative Generality Abstractizarea exagerată în sisteme software care nu o utilizează, devine un mecanism greu de utilizat. Perspectiva utilizării ei în viitor nu aduce o motivaţie pentru prezent. Există mecanisme de eliminare a abstractizărilor. Acele părţi care sunt folosite doar în clasele de test trebuiesc aduse în domeniul concretului.

Temporary Field Uneori în clase apare câte un câmp rătăcit care nu este folosit decât ocazional. Acest câmp este orfan şi de regulă se introduce în cazul algoritmilor complicaţi. În loc de a fi trimis ca parametru el se introduce ca şi câmp în clasă.

Message Chains Un lanţ de mesaje apare în sisteme software când un obiect cere altui obiect un serviciu, care la rândul lui cere altui obiect. Se consideră că înlănţuirile sunt periculoase şi nu trebuie folosite sub nici o formă. Totuşi uneori utilizarea lor este recomandată, dar în manieră moderată.

Middle Man Eliminarea claselor intermediare şi utilizarea directă a obiectului ţintă este scopul în situaţii în care jumătate din interfaţa unei clase este realizată de altă clasă. Aici este vizată folosirea încapsulării în mod inadecvat.

Inappropriate Intimacy Clasele care sunt preocupate foarte intens de părţi intime ale altor clase, trebuiesc separate. Moştenirea poate duce la astfel de complicaţii. Subclasele pot explora superclasele mai mult decât acestea ar dori să fie cunoscute.

Alternative Classes with Different Interfaces Clasele care fac acelaşi lucru dar au semnături diferite trebuie să poarte acelaşi nume.

Incomplete Library Class Clasele de bibliotecă nu pot fi întotdeauna pregătite spre orice tip de reutilizare. Se pot introduce extensii locale de comportament ori de câte ori se simte nevoia.

Data Class Clasele de date sunt acele clase care separă datele de comportamentul lor natural. Ele sunt acceptate ca şi clase de pornire dar nu este corect să rămână în acest stadiu.

Refused Bequest Unele subclase moştensc de la părinţi funcţionalităţi de care nu au nevoie şi pe care nu le vor folosi niciodată. În astfel de situaţii s-a greşit la proiectarea ierarhiei. Subclasa refuză să implementeze interfaţa superclasei.

3.6 Tipare de proiectare

Următorul pas în evoluţia programării orientate pe obiecte pot fi liniştit considerate *tiparele de proiectare*.

Utilizarea *tiparelor de proiectare* facilitează şi chiar asigură capacităţile de reutilizabilitate a sistemelor moştenite. Voi prezenta cateva detalii legate de tiparele de proiectare.

3.6.1 Ce sunt tiparele de proiectare

Tiparele de proiectare pot fi definite în modul cel mai simplist astfel: tiparele de proiectare sunt moduri inteligente şi intrinseci de rezolvare a unei clase de probleme [9, Ecke00].

Ceea ce nu se înscrie în concepţia clasică referitoare la proiectare, prin care se înţelege:

- analiză
- proiectare
- implementare

este faptul că tiparele aduc cu ele de la bun început cod, adică implică parţial implementare.

Conceptul de bază a tiparelor se suprapune peste conceptul de bază al proiectării, care este: *adăugarea unui nivel de abstractizare* [9, Ecke00]. Scopul tiparelor este acela de a izola părţile care *se modifică* de cele care *nu se modifică*. Voi exemplifica prin două tipare ce sunt implemetate de compilatoarele de Java şi cel de C++:

- *moştenirea* oferă posibilitatea de a exprima diferenţe în comportament (care se modifică) al obiectelor cu aceeaşi interfaţă (care rămâne fixă).

- *compoziţia* permite schimbarea dinamică sau statică a obiectelor care determină comportamentul clasei.

3.6.2 Taxonomia tiparelor

- *idiom* este modalitatea de a realiza o operaţie particulară, dependentă de limbaj

- *proiectare specifică* soluţie care rezolvă un tip particular de problemă, care poate fi inteligent realizată, dar care nu tinde să se generalizeze,

- *proiectare standard* o modalitate de a rezolva un tip de problemă, tinde să se generalizeze prin reutilizare

- *tipar de proiectare* modalitate de a rezolva o clasă întreagă de probleme, se porneşte de la aplicarea de metode de proiectare standard în mai multe aplicaţii şi pe urmă se remarcă esenţa tiparului din fiecare aplicaţie.

3.7 Detector de carenţe de proiectare

Un detector de carenţe de proiectare nu este un *test checker* adică un instrument software care verifică dacă sunt implementate toate firele logice al unui program care se ramifică prin intermediul instrucţiunilor *if* sau *case*.

De asemenea nu este nici un *bound checker* care să se ocupe de detecţia scurgerilor de memorie. Nu detectează unde pot apare referinţe nule sau invalide în runtime.

Nu se fac nici verificări de reguli, nu este *rule checker*. Un rule checker verifică dacă sunt respectate reguli de genul: este interzisă folosirea instrucţiunilor *break* şi *continue*. Sau se pot imagina reguli de interzicere a declaraţiilor de funcţii C în surse C++. Pentru aceste ţinte există produse comerciale cum ar fi TELELOGIC TAU, LOGISCOPE produs al firmei TELELOGIC. Poate acest set de instrumente CASE (Computer Aided Software Engineering) vor conţine şi un modul care să uziteze de abordarea bazată pe metrici.

Capitolul 4

Detecţia automată a carenţelor de proiectare

După cum sună şi titlul, abordarea noastră se bazează pe metrici. În figura 4.1 voi creiona schema procesului de detecţie, automatizat de instrumentul software despre care vom vorbi puţin mai încolo.

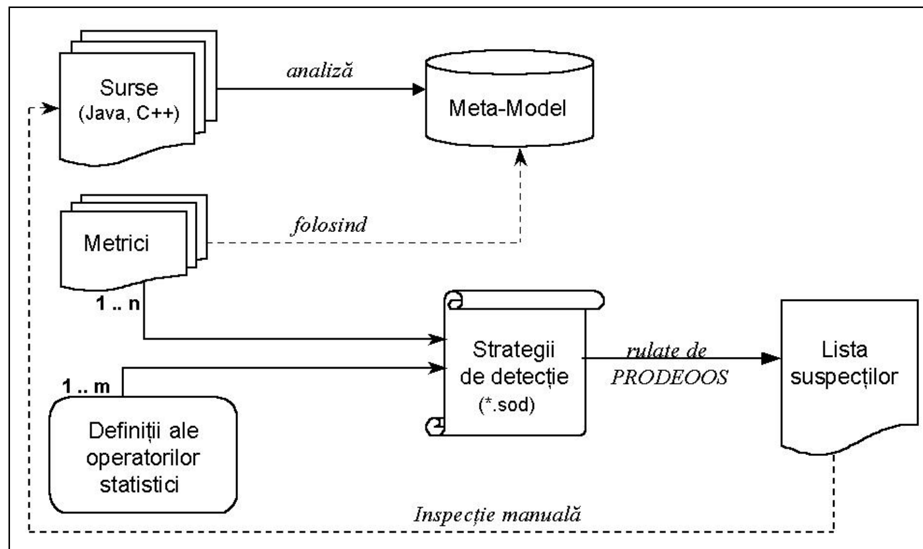

Figura 4.1: Abordare bazată pe metrici

Se porneşte de la codul sursă al unui sistem moştenit care se supune unor prelucrări pentru a obţine *metamodelul*. Vom intra puţin în detaliie metamodelului.

4.1 Metamodel

Acest metamodel conţine de fapt elementele care au valoare semantică în cadrul procesului de detecţie. Ne gândim la entităţi de tipul:

- claselor
- funţiilor
- subsistemelor.

Toate informaţiile legate de ele sunt stocate în tabele care au câmpuri a căror semnificaţie este comentată în cele ce urmeaža:

Tabela 4.1: Tabela acceselor

Câmp	Descriere
f_module	numele complet al fişierului unde are loc accesul variabilei incluzând calea până la acel fişier
f_start	linia unde începe accesul
f_start_char	poziţia caracterului unde începe accesul
f_stop	numărul liniei unde se termină accesul
f_stop_char	poziţia caracterului unde se termină accesul
f_class	numele clasei unde are loc accesul
f_function	numele funcţiei unde are loc accesul
f_signature	signatura funcţiei unde are loc accesul
f_name	numele variabilei accesate
f_type	tipul variabilei accesate
f_provider_class	numele clasei unde variabila
f_use	indică ce tip de variabilă a fost accesată (parametru, variabilă globală, atribut)
f_is_static	specifică dacă variabila este statică sau nu
f_is_complex	specifică dacă tipul variabilei este unul predefinit sau unul definit de utilizator
f_is_interface	diferenţiază accesul la o interfaţă de unul la o clasă
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.

Câmp	Descriere
f_provider_package	numele subsistemului unde variabilă este definită
f_how_many	numărul de accese la aceeaşi variabilă în acceeaşi funcţie

Tabela 4.2: Tabela apelurilor

Câmp	Descriere
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier
f_start	linia unde începe accesul
f_start_char	poziţia caracterului unde începe accesul
f_stop	numărul liniei unde se termină accesul
f_stop_char	poziţia caracterului unde se termină accesul
f_class	numele clasei unde are loc accesul
f_function	numele funcţiei unde are loc accesul
f_signature	signatura funcţiei unde are loc accesul
f_access_specifier	specifică tipul de funţie unde are loc accesul
f_called_class	numele clasei unde are loc accesul
f_called_function	numele funcţiei unde are loc accesul
f_called_signature	signatura funcţiei unde are loc accesul
f_called_access_specifier	specifică tipul de funcţie unde are loc accesul
f_how_many	numărul de accese la aceeaşi variabilă în acceeaşi funcţie
f_is_overloaded	indică dacă funcţia specificată este supraîncărcată sau nu. O funcţie este supraîncărcată dacă există mai multe implementări având acelaşi număr de parametri.
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.
f_called_package	numele pachetului unde funcţia apelată este definită

Tabela 4.3: Tabela claselor

Câmp	Descriere
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier

Câmp	Descriere
f_start	linia unde începe accesul
f_start_char	poziţia caracterului unde începe accesul
f_stop	numărul liniei unde se termină accesul
f_stop_char	poziţia caracterului unde se termină accesul
f_class	numele clasei unde are loc accesul
f_scope	pentru clase interne, acest câmp conţine numele clasei gazdă care conţine clasa internă
f_is_abstract	specifică dacă clasa este abstractă
f_is_template	specifică dacă clasa este generică
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.
f_namespace	numele spaţiului unde clasa este definită

Tabela 4.4: Tabela declaraţiilor

Câmp	Descriere
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier
f_start	linia unde începe accesul
f_start_char	poziţia caracterului unde începe accesul
f_stop	numărul liniei unde se termină accesul
f_stop_char	poziţia caracterului unde se termină accesul
f_namesepace	numele spaţiului unde are loc accesul
f_class	numele clasei unde are loc accesul
f_function	numele funcţiei unde are loc accesul
f_signature	signatura funcţiei unde are loc accesul
f_name	numele variabilei declarate
f_type	tipul de bază al variabilei
f_type_compl	tipul complet al variabilei
f_use (f_access_specifier)	specifică tipul de variabilă
f_is_complex	specifică dacă tipul variabilei e unul predefinit sau unul definit de utilizator
f_is_template	specifică dacă tipul variabilei este generic
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.

Tabela 4.5: Tabela funcţiilor

Câmp	Descriere
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier
f_start	linia unde începe accesul
f_start_char	poziţia caracterului unde începe accesul
f_stop	numărul liniei unde se termină accesul
f_stop_char	poziţia caracterului unde se termină accesul
f_class	numele clasei unde are loc accesul
f_function	numele funcţiei unde are loc accesul
f_signature	signatura funcţiei unde are loc accesul
f_return	tipul obiectului returnat de funcţie
f_use (f_access_specifier)	specifică tipul de variabilă
f_storage (f_storage_specifier)	specificatorul de depozitare
f_ct_cyclo	valoarea numărului ciclomatic al unei funcţii
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.

Tabela 4.6: Tabela moştenirilor

Câmp	Descriere
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier
f_start	linia unde începe accesul
f_start_char	poziţia caracterului unde începe accesul
f_stop	numărul liniei unde se termină accesul
f_stop_char	poziţia caracterului unde se termină accesul
f_class	numele clasei unde are loc accesul
f_parent	numele clasei strămoş
f_attribute	specifică modul cum subclasa este derivată din superclasă. Acest atribut influenţează vizibilitatea membrilor din clasa părinte spre clasa derivată.

Câmp	Descriere
f_ct_dit	specifică distanţa în arborele de moştenire dintre clasa părinte şi cea derivată

În aceste tabele este parţial prezent codul sursă. Se regăsesc accese, apeluri, declaraţii de clase, declaraţii de funcţii, relaţii de moştenire. Ce nu vom regăsi în metamodel sunt structurile decizionale şi repetitive. Nu vom întâlni cicluri *for*, *while*, instruţiuni de genul *if*, *case*.

4.2 Metrici

Instrumentul de măsură a calităţii unui sistem software este *metrica*. Metrica surpinde calitativ şi cantitativ diferite caracteristici ale *entităţilor* din sistemele software orientate-obiect.

Scopul metricilor este acela de a ordona entităţile. În subparagrafele următoare exemplificăm câteva metrici şi carenţele care ar putea să le indice.

4.2.1 Greutatea unei clase

După cum îi spune şi numele, această metrică cântăreşte procentual câte metode non accesor are o clasă, raportat la numărul total de membri. Cu alte cuvinte o clasă este cu atât mai grea cu cât are mai multe metode funcţionale, adică acea clasă "ştie" să facă multe lucruri. În cadrul dezvoltării unui sistem software, se poate porni de la clase "uşoare", dar pe parcurs acestea trebuie să câştige greutate. Altfel ele riscă să fie clase de date monotone, de care vor depinde alte clase, creindu-se dependenţe nesănătoase.

4.2.2 Numărul de atribute publice

Metrica care numără atributele publice ale unei clase are rolul de a delimita acele clase, cu mulţi membri publici, care pot adesea să facă parte din colecţia de clase suspecte, posibile clase de date.

4.2.3 Numărul de metode accesor

Metrica numărătoare a metodelor accesor al unei clase poate demasca o clasă de date, care nu are membri publici, dar îşi ascunde membrii, sub funcţiile de acces, de citire şi de scriere. În cazul în care o clasă prezintă un număr mare de metode accesor, ea devine suspectă şi trebuie supusă inspecţiei manuale.

4.3 Operatori statistici - outliers

Cuvântul *outlier* înseamnă în limba engleza ceva ieşit din rând, cel care se deosebeşte de alţii, având o trăsătură anormală. Voi exemplifica câţiva dintre operatorii statistici care îi vom folosi în strategiile de detecţie:

- **Valorile mari - Top Values**
- **Valorile mici - Bottom Values**
- **Valorile mai mari decât - Higher Than**
- **Valorile mai mici decât - Lower Than**
- **Valorile extreme - BoxPlots**.

Pragurile impuse operatorilor statistici vor fi valori experimentale, care la început vor oscila în jurul unei valori, apoi se vor stabiliza. Alegerea acestor valori se poate face şi prin prisma rigurozităţii. Dacă dorim să fim foarte stricţi vom îngusta plaja de valori acceptate. Ajustarea se face prin calibrarea operatorilor statistici.

4.3.1 Problema zerourilor

Acestă problemă apărut în momentul în care am dezvoltat operatorul *Box Plots*. Problema pe care o pune acesta este una delicată. Deoarece el lucrează pe seturi de valori ce se înscriu pe scări neliniare, apare un hazard dacă analizăm doar entităţile returnate de metrică. Metrica va calcula valori numerice doar pentru entităţile pentru care are aceasta are valoare semantică. De accea soluţia care se impune este accea a colectării tuturor entităţilor. Entităţile rezultate în urma rulării metricii, vor avea valori calculate, pe când restul entităţilor vor avea valoarea 0 din oficiu. Astfel problema zerourilor are o rezolvare elegantă. Totul se rezolvă la implementare prin adăugarea la limbajul SOD, prezentat în secţiunea următoare, a unui parametru suplimentar la regula operatorilor statistici, care să permită optarea pentru analiză pe setul metricii sau pe toată colecţia de entităţi.

4.4 Strategie de detecţie - limbajul SOD

4.4.1 Gramatica

Strategia de detecţie a unei carenţe poate fi definită în felul următor: [6, Mari01a] expresie cuantificabilă a unei reguli prin care entităţile afectate de

carenţa respectivă să poată fi automat detectate. Pornim de la următoarea definiţie formală bazată pe metrici:

$$S := {M_1}^{O_1} * {M_2}^{O_2} * \cdots * {M_n}^{O_n}$$

$$* := \cup \quad | \quad \cap \quad | \quad \setminus$$

Pentru a opera asupra metamodelului vom avea nevoie de o structură arborescentă care să modeleze formula teoretică. De accea m-am gândit la următorul set (simplificat) de reguli:

```
01 DetectionStrategy     := StrategyDefinition SymbolsDefinition
02
03 # reguli pentru definirea unei strategii de detectie
04 StrategyDefinition    := StrategyName ":=" DetectionRule ";"
05 DetectionRule         := MetricWithOutliers |
06                          ComposedDetectionRule
07 MetricWithOutliers    := "(" MetricName "," OutlierName ")"
08 ComposedDetectionRule := DetectionRule CompositionOperator
09                          DetectionRule
10 StrategyName          := [A-z][A-z0-9_]
11 CompositionOperator   := "or" | "and" | "butnotin"
12
13 # simbolurile sunt definitii de metrici si operatori
14 SymbolsDefinition     := MetricDefinition |
15                          OutlierDefinition
16
17 # reguli pentru definirea metricilor
18 MetricDefinition   := MetricName  ":=" SqlQuery ";"
19 MetricName         := [A-z][A-z0-9_]
20 SqlQuery           := [.]   # SELECT <Entity> <Value>
21
22 # reguli pentru definirea operatorilor statistici
23 OutlierDefinition := OutlierName ":=" OutlierType
24                     "(" OutlierParameter ")" ";"
25 OutlierType       := "TopValues" | "BottomValues" |
26                      "HigherThan"| "LowerThan" | "BoxPlots"
27 OutlierName       := [A-z][A-z0-9_]
28 OutlierParameter  := [0-9][0-9,][%]
```

Instrumentul software modelează aceste reguli printr-un analizor lexical, un analizor sintactic şi un generator de arbore. Ideea de bază a gramaticii

este următoarea: o strategie poate fi o metrică simplă sau o metrică compusă dintr-un operator logic şi alte metrici. Astfel putem construi într-un mod relativ simplu strategii de detecţie de diferite complexităţi.

4.4.2 Analizorul lexical

Analizorul lexical este unul bazat pe o maşină de stări. Trecerea dintr-o stare în alta se face pe baza simbolului citit la intrare. Un aspect special îl constituie citirea script-urilor SQL, care nu vor fi verificate din punct de vedere lexical şi sintactic. Ele sunt citite până la întâlnirea unui separator.

4.4.3 Analizorul sintactic

Analizorul sintactic este de tip LR(1), implementat pe o gramatică relativ simplă. Singurul artificiu necesar pentru a putea modela corect strategia a fost eliminarea recursivităţii de stânga a regulii de detecţie.

Voi explica puţin această gramatică prin prisma analizorului sintactic cu care am modelat-o. Din linia 01 rezultă că o strategie de detecţie este alcătuită din două componente principale:

1. definiţia strategiei
2. definiţia simbolurilor.

Definiţia strategiei este compusă la rândul ei dintr-un nume şi o regulă de detecţie. Regula de detecţie se poate reduce la o metrică înzestrată cu un operator statistic, sau se poate expanda într-o regulă de deteţie compusă. Regula de detecţie compusă (liniile 08-09) este formată din două reguli de detecţie şi un operator de compunere. Operatorul de compunere la rândul lui se poate detalia în operator de reuniune, intersecţie sau diferenţă (linia 11).

Definiţia unui simbol este bivalentă: poate fi definiţie de metrică sau definiţie de operator statistic. O metrică se defineşte prin nume şi un script SQL (liniile 18-20). Un operator statistic are un nume, un tip şi eventual un parametru. Tipurile de operatori statistici cu care lucrăm în momentul de faţă sunt:

1. *BottomValues* şi *TopValues* operator care extrage valorile maxime şi minime dintr-un set de date
2. *HigherThan* şi *LowerThan* operator ce selectează valorile mai mari, respectiv mai mici decât un prag dat

3. *BoxPlots* operator ce selectează valorile extreme printr-o tehnică specială.

4.4.4 Generatorul de arbore

Generatorul de arbore este de fapt construit pe scheletul analizorului sintactic, dar în loc să se ocupe de verificări el de fapt se ocupă de construirea arborelui de detecţie. Arborele construit astfel poate executa în mod automat procesul de detecţie. Arborele de detecţie are dublă perspectivă: una structurală în care nodurile sunt *metrici, operatori statistici, operatori logici* şi una valorică în care nodurile structurale conţin *rezultate*, adică seturi de entităţi.

4.4.5 Arborele de detecţie

Arborele de detecţie este structura de date în jurul căruia se întâmplă detecţia. Acest arbore are trei nuanţě. Prima aromă este cea de arbore alcătuit din metrici simple, metrici compuse. Această abordare este una de abstractizare. A doua nuanţă se simte la implementarea concretă a abstracţiunilor. Metricile sunt implementate prin query-uri SQL, metricile compuse prin intermediul operatorilor logici. O altă dimensiune a arborelui, puţin diferită de cele două, prezentate anterior este aceea a datelor rezultate. Această perspectivă este cea interesantă pentru un utilizator obişnuit PRODEOOS. Un design-er de strategii lucrează pe nivelul al doilea. El va scrie script-urile SQL, va utiliza operatori logici, operatori statistici. Pe primul nivel operează geniul abordării detecţiei bazate pe metrici.

4.5 Exemplu de strategie - "Data classes"

4.5.1 Prezentarea strategiei

Am ales ca exemplu o strategie simplă care să detecteze clasele de date [7, Mari01b].

```
DataClassesStrategy:=
                (
                  (WOC, WOCBottom) and (WOC, WOCLower) and
                  ((NOPA, NOPAOutliers) or
                  (NOAM, NOAMOutliers))
                );
```

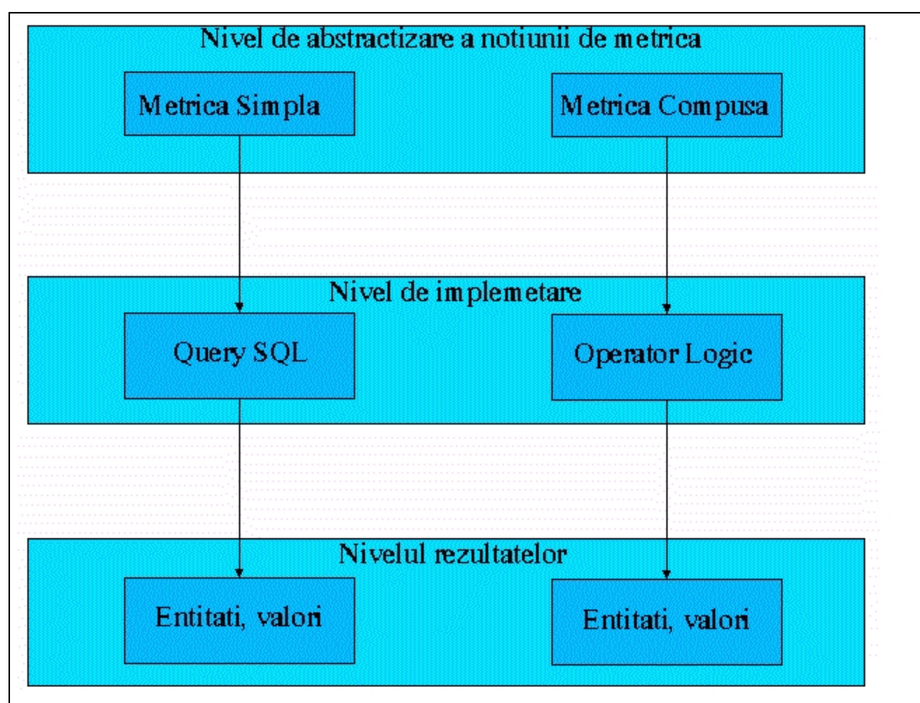

Figura 4.2: Cele 3 dimensiuni ale arborelui de detecţie

```
WOC:=
      SELECT all_pub.f_class AS f_class,
       (100*pub_mth.cnt / all_pub.cnt) AS woc FROM...
NOPA:=
      SELECT f_class, count(f_class) AS nopa
      FROM v_members...
NOAM:=
      SELECT f_class, count(F_class) AS noam
      FROM v_accessor_methods...

WOCBottom    := BottomValues(10);
WOCLower     := LowerThan(33);
NOPAOutliers := TopValues(7);
NOAMOutliers := TopValues(5);
```

Voi comenta pe scurt această strategie, care îsi propune să detecteze clasele de date. Prin clase de date înţelegem acele clase care separă datele de comportamentul lor. Bazându-ne pe această idee, construim strategia în

felul următor: ne interesează acele clase care au membri mulţi şi nu prea au metode funcţionale. Pentru acestă strategie putem folosi o metrică prin care se măsoară greutatea unei clase, pentru a le detecta pe cele cu procent mic de funcţionalitate, metrica care măsoară numărul de atribute publice şi metrica care numără metodele accesor. Operatorii statistici sunt calibraţi astfel: caut clase mai uşoare de 33 procente uitându-mă la ultimele 10, caut primele 7 clase cu cele mai mari numere de atribute publice, primele 5 clase cu cele mai multe metode accesor.

Capitolul 5

Instrumentul software - Prodeoos

5.1 Structură

Instrumentul software are o structură modulară, fiind alcătuit din următoarele componente:

- *detectorul de carenţe* componenta care conţine principala funcţionaliate a instrumentului software
- *managerul de conexiune* care se ocupă de legătura cu motorul de baze de date
- *vizualizatorul de surse* care facilitează introspecţia în codul sursă
- *monitorul SQL* care permite executarea rapidă a unor query-uri
- *generatorul de rapoarte* care salvează rezultatele în format HTML.

Vom discuta pe scurt câteva dintre aspectele mai importante ce caracterizează componentele enumerate.

5.2 Detectorul de carenţe

5.2.1 Structură

Detectorul de carenţe operează pe o structură de arbore turnată într-un tipar compozit. În momentul în care i se dă semnalul, arborele de detecţie începe să rodească. Fructele lui vor fi găsite în orice nod al său sub forma rezultatelor

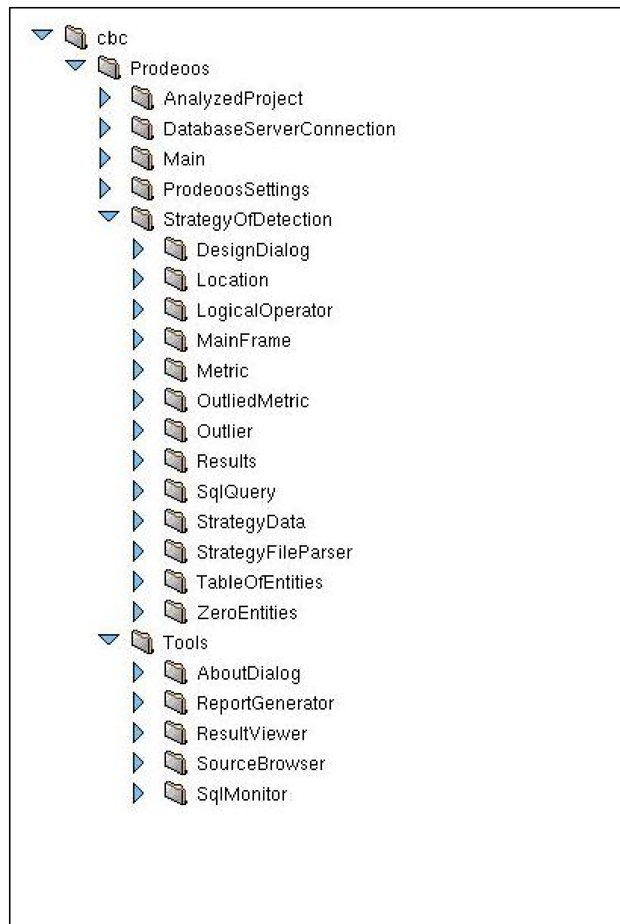

Figura 5.1: Structura PRODEOOS

pe care le voi prezenta imediat. Componentele detecţiei vor fi detaliate în capitolele ce urmează.

Ideea de bază a fost să stabilesc interfeţe pentru fiecare tip de element al detecţiei:

- pentru metrici interfaţa *AbstractOutliedMetric*
- pentru operatorii logici *AbstractLogicalOperator*
- pentru operatorii statistici *AbstractOutlier*.

Instanţele la clasele care implementează interfeţele pot fi referite doar prin intermediul acestora. Acest nivel de abstractizare ne oferă flexibilitate şi reprezintă partea stabilă a proiectului.

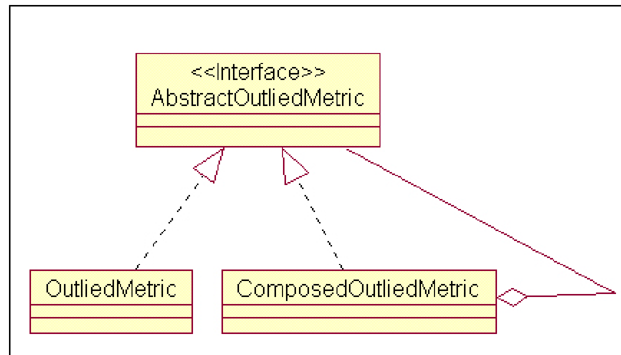

Figura 5.2: Structura arborelui de detecţie

5.2.2 Metrică simplă

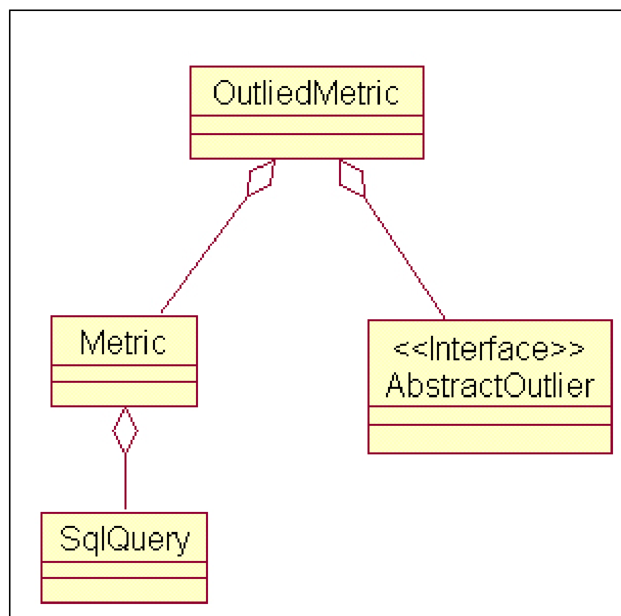

Figura 5.3: Structura metricii simple

O metrică simplă după cum se vede şi în figură este alcătuită dintr-o metrică propriu zisă şi un operator statistic. Se mai remarcă faptul că o metrică pură conţine un SqlQuery. Comportamentul unei metrici simple constă dintr-o metodă de execuţie a query-ului şi una de acces la vectorul de rezultate.

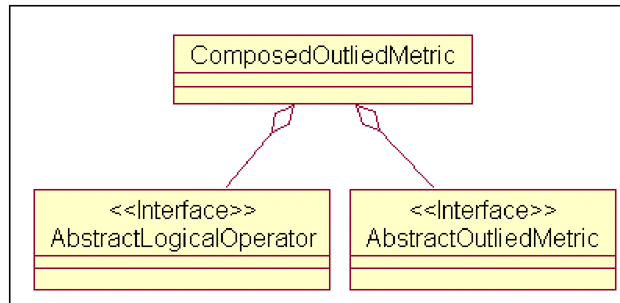

Figura 5.4: Structura metricii compuse

5.2.3 Metrică compusă

Metrica compusă e alcătuită la rândul ei dintr-un operator logic şi două metrici abstracte. În locul metricilor abstracte ne putem imagina fie o metrică simplă, fie una compusă. Metrica compusă va folosi intertaţa operatorului logic pentru a face operaţiile relaţionale necesare. Metrica compusă este elementul care permite crearea unei strategii oricât de complexă, pe oricâte nivele.

5.2.4 Operatori logici

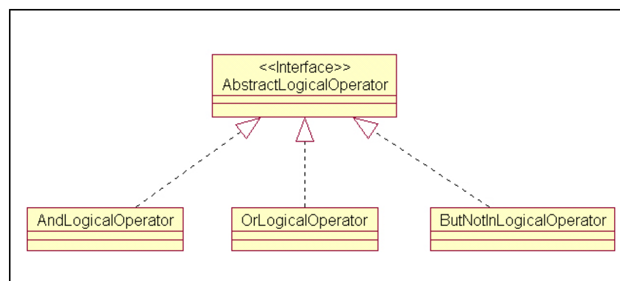

Figura 5.5: Ierarhia operatorilor logici

Operatorii logici sunt trei la număr:

- *AndLogicalOperator* care efectuează intersecţia a două seturi de rezultate
- *OrLogicalOperator* care realizează reunuiunea a două seturi de rezultate
- *ButNotInLogicalOperator* face diferenţa dintre cele două mulţimi de rezultate.

Rezultatul unui operator logic este de fapt rezultatul corespunzător unei metrici compuse. Se poate observa o corespondenţă între conceptul de metrică compusă şi implementarea acestuia prin intermediul operatorului logic.

5.2.5 Operatori statistici

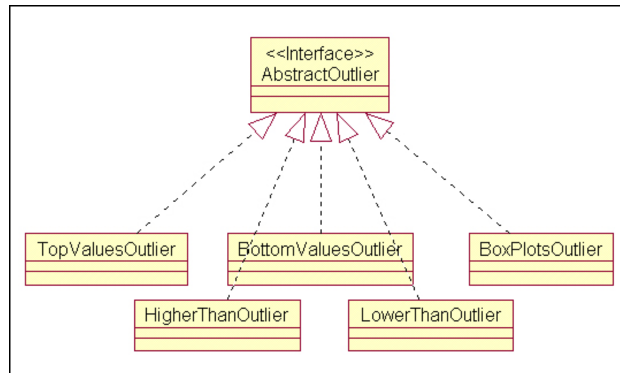

Figura 5.6: Ierarhia operatorilor statistici

Operatorii statistici implementaţi sunt cei prezentati în capitolul 2. Diagrama lor de clasă arată o interfaţă cu rol de abstractizare, şi cinci clase implementatoare. Ca şi comportament ei respectă aceeaşi filosofie: un operator statistic primeşte un set de rezultate de la metrica pe lângă care activează, şi este interogat de rezultate de un operator logic, care este un nod superior în arborele de detecţie. Poate că s-ar mai fi putut introduce un nivel de abstractizare legat de natura operatorilor: doi operatori folosesc praguri, alţi doi folosesc valori extreme. S-ar putea construi operatori statistici activi, cu capacitate de autocalibrare în cazul în care apar între seturile de date foarte multe valori egale.

5.2.6 Rezultate

Rezultatele sunt obiecte compuse din *entităţi măsurate*, care sunt la rândul lor alcătuite din *valori măsurate*. Ca şi soluţie de modelare a colecţiei de entităţi măsurate am ales vectorul oferit de pachetul de utilităţi al limbajului Java. Rezultatele se pot vizualiza foarte sugestiv sub formă de tabel. Pe coloane sunt trecute metricile, pe prima coloană apar numele entităţilor, iar apoi pe fiecare rând valoarea metricii pentru entitătea respectivă. Trebuie menţionat că un astfel de tabel există în fiecare nod al arborelui de detecţie după ce strategia a fost rulată.

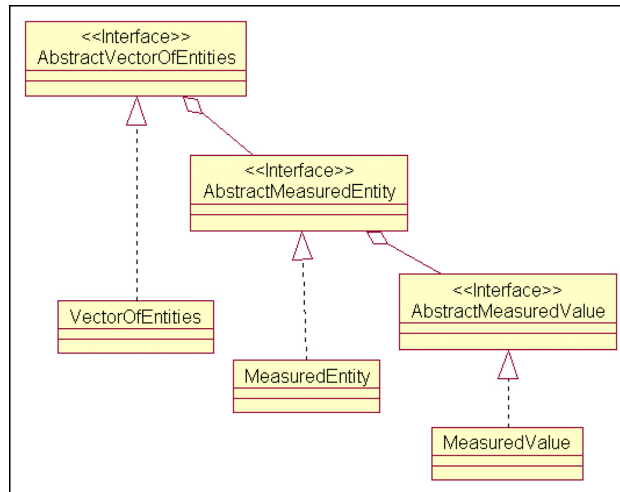

Figura 5.7: Modelul rezultatelor

5.2.7 Vizualizare

Fereastra este împărţită în trei zone. Zona din stânga conţine arborele de deteţie, cea din dreapta sus - tabelul cu rezultatele nodului curent selectat din arbore, iar cea din dreapta jos - semnătura nodului selectat din arbore. Butonul din partea de jos a ferestrei execută funcţia principală a programului. Nodurile arborelui din partea stângă determină conţinutul celorlalte două zone. Selecţia unui nod va determina un tabel de rezultate şi o semnătură specifică nodului. Tabelul va conţine datele din nodul arborelui, iar signatura este o afişare în inordine a nodurilor subarborelui determinat de nodul selectat. Astfel ne putem orienta mai bine şi putem să vedem cine a determinat rezultatul, prin aceasta înţeleg metricile, operatorii logici, operatorii statistici care au condus la obţinerea acestuia.

5.3 Managerul de conexiune

5.3.1 Structură

Din punct de vedere al ierarhiei de clase, managerul de conexiune este o componetă realizată printr-un tipar de intermediere. Comportamentul unei clase care implementează o asftel de conexiune este simplu. Ea are rolul de a furniza un obiect consistent care pe urmă să poată să facă operaţiile cu tabele. Tehnologia de acces la bazele de date prezentată din prisma aplicaţiilor client, cum e şi cea de faţă, este următoarea: instaţierea unui obiect driver în mod

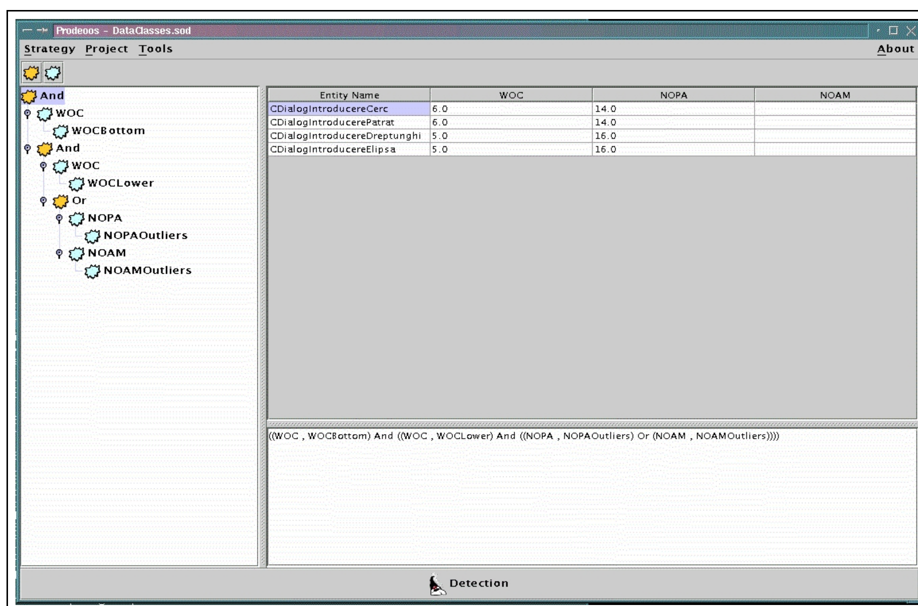

Figura 5.8: Instrumentul software - Prodeoos

dinamic dintr-o clasă compilată de producătorul driver-ului. Cu acest driver care implemetează o interfaţă comună tuturor producătorilor, se obţine un obiect cu rol de conexiune. Rolul clar al managerului de conexiune este acela de a obţine instanţa conexiunii, moment în care datoria lui s-a încheiat.

Este important de reţinut că PRODEOOS **nu depinde** sub nici o formă de vreun motor anume. Singura condiţie care se impune este aceea ca motorul să suporte SQL standard.

5.3.2 Vizualizare

Managerul de conexiune nu este o componentă foarte vizibilă atât timp cât nu apar erori. Când se nasc probleme legate de motorul de baze de date ea devine prezentă prin intermediul unor cutii cu mesaje referitoare la defectele apărute. Deoarece clasa care se instaţiază pentru obţinerea conexiunii este una externă sunt mari şanse ca aceasta să nu facă parte din *classpath* şi să nu poată fi găsită. O atenţie deosebită trebuie acordată sistemului de securitate al motorului cu care se lucrează. Conexiunea este parametrizată de nume de utilizator, parolă, referinţă la baza de date. Incorectitudinea vreunuia dintre parametri poate conduce de asemenea la eroare. În funcţie de producătorul driver-ului se poate discuta de tratarea şi documentarea erorilor. Driverul firmei SYBASE, JCONNECT-4.2 face toate aceste lucruri.

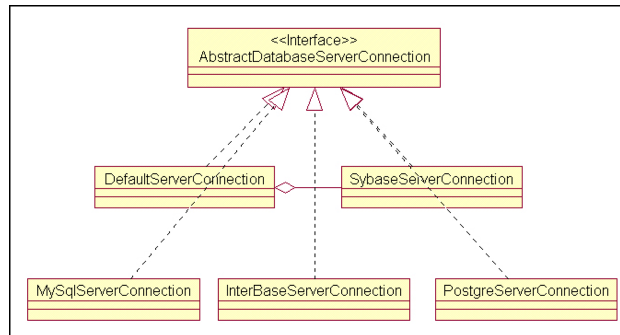

Figura 5.9: Structura managerului de conexiune

5.4 Vizualizarea surselor

Structură

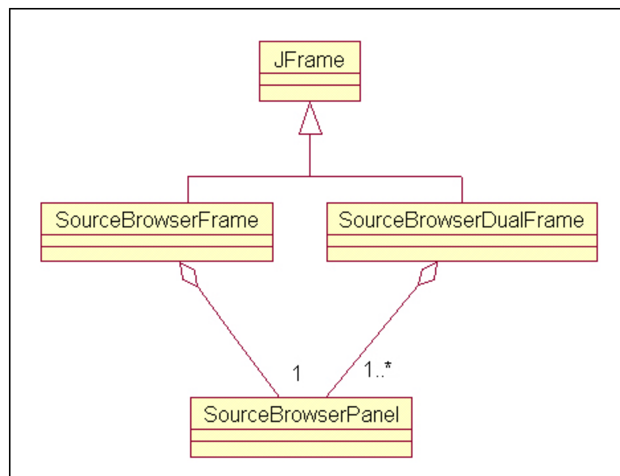

Figura 5.10: Structura vizualizatorului de surse

Vizualizatorul de surse este modelat pe un tipar de strategie. Deoarece există două limbaje de programare orientate pe obiecte a căror surse pot fi analizate prin intermediul metamodelului există necesitatea diferenţierii modului de vizualizare a codului sursă. Proiectele C++ vor avea nevoie de o vizualizare duală: declaraţia şi definiţia; declaraţia se găseşte în fişierul header, iar definiţia în fişierul cu extensia CPP. Pentru limbajul Java este suficientă vizualizarea unui singur fişier, deoarece este un limbaj organizat "inline".

5.4.1 Vizualizare

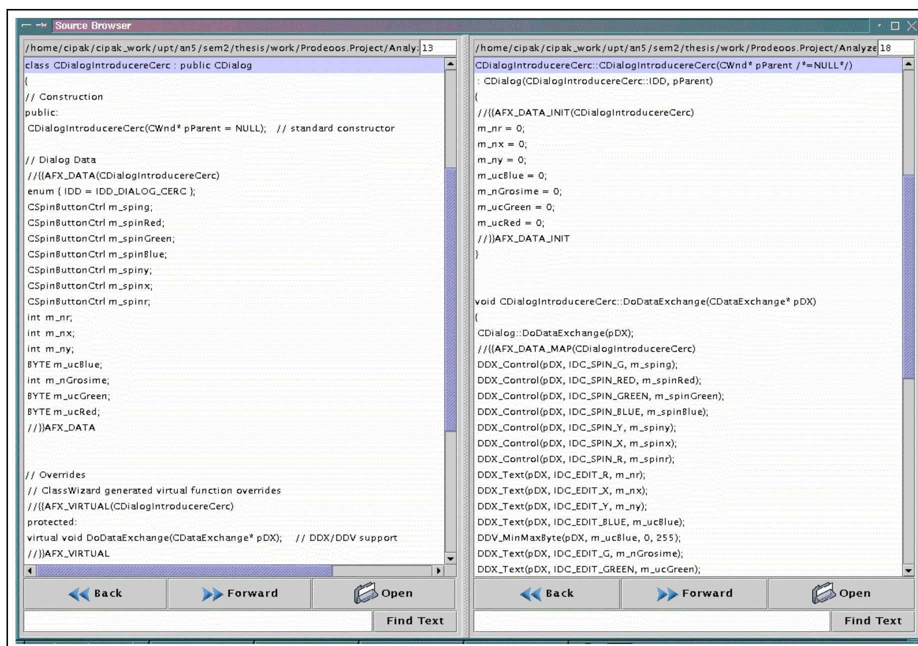

Figura 5.11: Vizualizatorul de surse

După cum am văzut şi în subcapitolul anterior, vizualizatorul de surse poate fi cu un panou sau cu două. Printre facilităţile lui amintim: căutarea de text, vizualizarea unei linii după număr, navigarea prin fişierele deschise recent, start cu marcarea unei entităţi anume. Rolul acestui navigator nu este acela de a corecta codul sursă şi apoi a relua testele. Refactorizările care se impun sunt de regulă complexe pentru că se fac la nivel de design. Scopul pentru care a fost făcut este acela de a permite utilizatorului confirmarea suspeţilor. Funţionalităţile lui enumerate mai sus confirmă scopul pentru care a fost proiectat. Codul sursă este organizat pe linii, în colţul din dreapta-sus existând un contor de linie pentru o orientare mai facilă. Contorul este unul activ, adică dacă i se impune o valoare, el va vizualiza linia cu numărul proaspăt introdus. Facilitatea de a naviga printre fişierele deschise anterior este una foarte des întâlnită la mediile de dezvoltare comerciale, ajutând la creşterea vitezei de inspectare. Ceea ce ar mai putea fi menţionat este accea că ar fi binevenită ideea unui control care să poată vizualiza codurile sursă cu evidenţierea sintaxei.

5.5 Monitorul SQL

5.5.1 Structură

Aici vom discuta despre modul în care aplicaţia interacţinează cu motorul de baze de date relaţionale. Menţionez că folosim un motor al firmei SYBASE, pus pe Internet spre evaluare. Când am prezentat abordarea nostră, am afirmat că se construieşte un metamodel ce constă în şase tabele ce vor fi manipulate de un motor. Tehnologia disponibilă la ora actuală care ne permite manipularea datelor prin comandarea motorului se numsşte JDBC (Java DataBase Conectivity), şi se realizează printr-un driver. Acest driver reprezintă un set de funţii şi clase Java care comunică cu motorul şi comandă efectuarea prelucrărilor de date. Trebuie specificat că pentru orice tip de motor există un driver separat scris de firma producătoare, dar care respectă interfaţa impusă de firma Sun Microsystems. Există patru tipuri de drivere:

- *tipul 1: punte JDBC-ODBC* care traduce apelurile JDBC în apeluri ODBC. Uneori ODBC are nevoie de o aplicaţie rezidentă pe partea de client

- *tipul 2: driver java nativ bazat pe API* care lucrează direct cu motorul de baze de date, prin funcţiile lui, API

- *tipul 3: driver java pentru reţea* ce lucrează prin intermediul unui server secundar, folosind un protocol independent de producătorul motorului. Server-ul secundar traduce apelurile în diferite dialecte de protocoale pentru o gamă largă de motoare

- *tipul 4: driver java nativ* care converteşte apelurile pe baza protocolului impus de producătorul motorului, facilitând o comunicaţie directă între client şi motor.

Driverul folosit de noi se numeşte jConnect-4.2 şi este:

- driver java pentru reţea într-un mediu cu trei nivele

- driver java nativ într-un mediu cu două nivele.

Protocolul folosit de jConnect este TDS-5.0 (Tabular Data Stream).

5.5.2 Vizualizare

Ca şi componentă de sine stătătoare a fost concepută această componentă. Ea este responsabilă de gestionarea script-urilor SQL. Monitorul execută

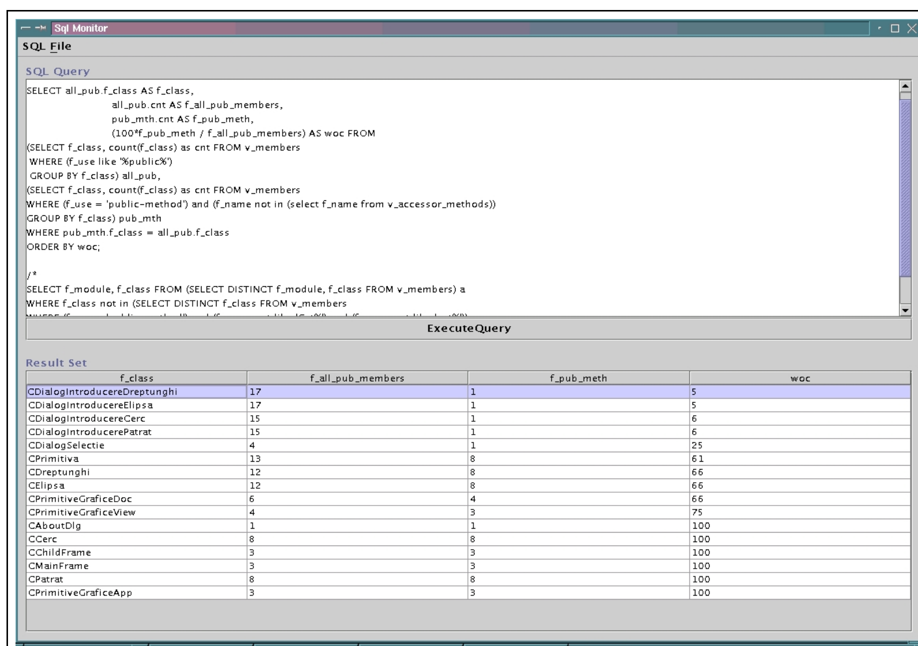

Figura 5.12: Monitorul SQL

operaţiile clasice pe care le efectuează un monitor SQL comercial. Oferă posibilităţi de deschidere de fişiere SQL, de salvare şi de execuţie a scriptului editat. Rezultatele sunt prezentate sub formă tabulară prin intermediul unui control al librăriei Swing. Rolul monitorului SQL este acela de a asista la creare unei noi strategii. El ajută utilizatorul să testeze metricile înainte de a le introduce într-o nouă strategie. Se poate justifica necesitatea unui astfel de monitor, faţă de cel oferit de producătorul motorului prin faptul că motorul şi aplicaţia pot rula distribuit. Mai mult pot rula chiar pe platforme diferite, caz în care monitorul SQL al producătorului este imposibil de folosit.

5.6 Generatorul de rapoarte

5.6.1 Structură

După cum se poate vedea şi în diagrama de clase există un nivel de abstractizare impus de interfaţa *AbstractReportGenerator* care este implementată de clasa *HtmlReportGenerator* şi posibil de clasa *TexReportGenerator*. Ideea de bază a fost introducerea unei modalităţi de a putea genera orice tip de rapoarte, lucru care se poate face prin definirea unei clase care are obligaţia să implementeze interfaţa *AbstractReportGenerator*. Clasele implementatoare

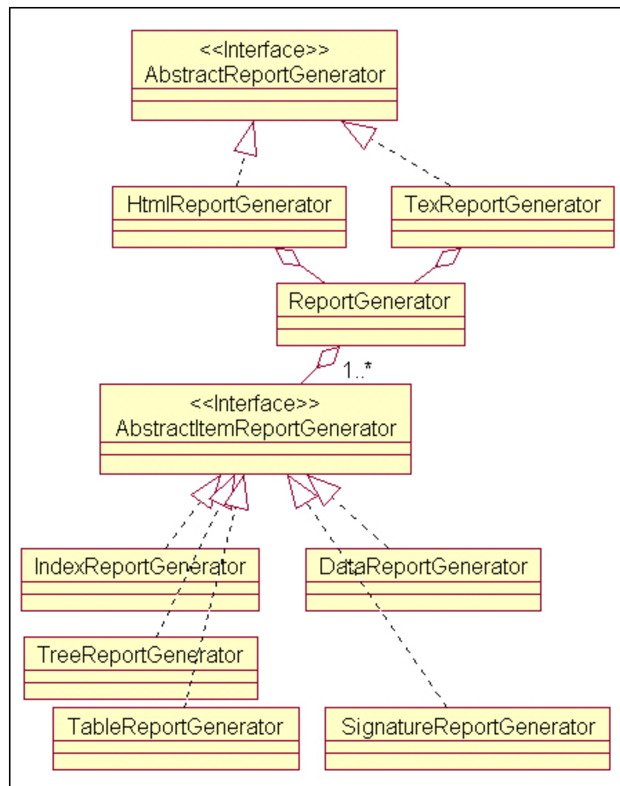

Figura 5.13: Structura generatorului de rapoarte

se pot folosi de clasa *ReportGenerator*, care "ştie" să lucreze cu obiecte de tip *AbstractItemReportGenerator*. Această divizare a generării rapoartelor este datorată faptului că rapoartele sunt formate din mai multe componente interdependente, în cazul nostru cu link-uri din una în cealaltă şi care se concretizează prin fişiere. Concret avem nevoie de:

- *generator de fişier index* din care se va putea naviga mai departe
- *generator de raport pentru arbore*, care conţine arborele de detecţie, sub forma unui tabel
- *generatorul de raport tabelă*, acest raport se va modifica la vizualizare în funţie de nodul selectat în arbore
- *generatorul de raport pentru semnături*, care specifică detalii referitoare la structura nodului arborelui, curent selectat.
- *generatorul de raport pentru date* care înglobează generatorul pentru tabelă şi cel pentru semnături.

5.7 Implementarea soluţiei problemei zerourilor

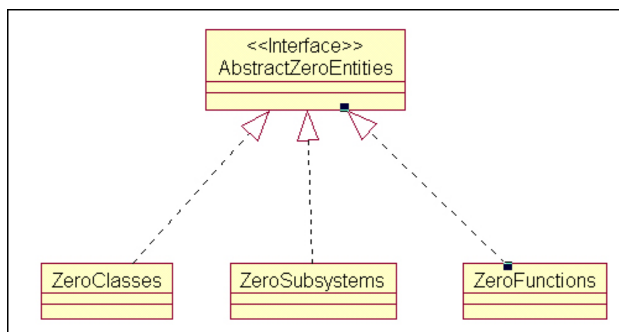

Figura 5.14: Modelare entităţi nule

5.8 Gestionarea documentelor

Prodeoos gestionează souă tipuri de documente: strategii de detecţie şi informaţii referitoare la proiectele analizate. Pentru strategii este creată o clasă *StrategyData* care conţine arborele de detecţie, metode pentru vizualizarea datelor la inspecţia arborelui, metode de serializare. Informaţiile referitoare la proiecte sunt stocate în clasa *ProjectData*. Structura de directoare referitoare la un proiect analizat este simplă şi necesită un director pentru surse, unul pentru rezultate, unul pentru metamodel, altul pentru baza de date şi fişierul de jurnal.

5.9 Studiu de caz

Propun să vedem cum se comportă Prodeoos pe proiecte reale de diferite dimensiuni şi care nu sunt de test. Nu sunt proiecte care vor “simula” în mod artificial carenţe de proiectare, care să fie apoi detectate. Vom realiza un studiu de caz concentrându-ne spre mecanismul de detecţie şi cele două trăsături ale sale *accelerarea detecţiei* şi *automatizarea ei*. Primul studiu va fi un proiect realizat în Microsoft Visual C++, intitulat *Primitive Grafice*, care manipulează în mod interactiv primitive grafice. El este dotat cu clase corespunzatoare unor dialoguri pentru setarea parametrilor primitivelor. La strategia *Data Classes* se vor evidenţia ca fiind suspecte clasele:

Entity Name	WOC	NOPA	NOAM
CDialogIntroducereCerc	6	14	
CDialogIntroducerePatrat	6	14	
CDialogIntroducereDreptunghi	5	16	
CDialogIntroducereElipsa	5	16	

```
// DialogIntroducereCerc.h : header file
//

////////////////////////////////
// CDialogIntroducereCerc dialog

class CDialogIntroducereCerc : public CDialog
{
// Construction
public:
CDialogIntroducereCerc(CWnd* pParent = NULL);
        // standard constructor

// Dialog Data
//{{AFX_DATA(CDialogIntroducereCerc)
enum { IDD = IDD_DIALOG_CERC };
CSpinButtonCtrl m_sping;
CSpinButtonCtrl m_spinRed;
CSpinButtonCtrl m_spinGreen;
CSpinButtonCtrl m_spinBlue;
CSpinButtonCtrl m_spiny;
CSpinButtonCtrl m_spinx;
CSpinButtonCtrl m_spinr;
int m_nr;
int m_nx;
int m_ny;
BYTE m_ucBlue;
int m_nGrosime;
BYTE m_ucGreen;
BYTE m_ucRed;
//}}AFX_DATA


// Overrides
```

```
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDialogIntroducereCerc)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
        // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
//{{AFX_MSG(CDialogIntroducereCerc)
virtual BOOL OnInitDialog();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};



// DialogIntroducereDreptunghi.h : header file
//

//////////////////////////////////////
// CDialogIntroducereDreptunghi dialog

class CDialogIntroducereDreptunghi : public CDialog
{
// Construction
public:
CDialogIntroducereDreptunghi(CWnd* pParent = NULL);
        // standard constructor

// Dialog Data
//{{AFX_DATA(CDialogIntroducereDreptunghi)
enum { IDD = IDD_DIALOG_DREPTUNGHI };
CSpinButtonCtrl m_spinRed;
CSpinButtonCtrl m_spinGrosime;
CSpinButtonCtrl m_spinGreen;
CSpinButtonCtrl m_spinBlue;
CSpinButtonCtrl m_spinL;
CSpinButtonCtrl m_spinl;
```

```
CSpinButtonCtrl m_spiny;
CSpinButtonCtrl m_spinx;
int m_nl;
int m_nL;
int m_nx;
int m_ny;
BYTE m_ucBlue;
BYTE m_ucGreen;
int m_nGrosime;
BYTE m_ucRed;
//}}AFX_DATA


// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDialogIntroducereDreptunghi)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
        // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
//{{AFX_MSG(CDialogIntroducereDreptunghi)
virtual BOOL OnInitDialog();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};


// DialogIntroducereElipsa.h : header file
//

//////////////////////////////////
// CDialogIntroducereElipsa dialog

class CDialogIntroducereElipsa : public CDialog
{
// Construction
```

```
public:
CDialogIntroducereElipsa(CWnd* pParent = NULL);
        // standard constructor

// Dialog Data
//{{AFX_DATA(CDialogIntroducereElipsa)
enum { IDD = IDD_DIALOG_ELIPSA };
CSpinButtonCtrl m_sping;
CSpinButtonCtrl m_spinry;
CSpinButtonCtrl m_spinrx;
CSpinButtonCtrl m_spiny;
CSpinButtonCtrl m_spinx;
CSpinButtonCtrl m_spinRed;
CSpinButtonCtrl m_spinGreen;
CSpinButtonCtrl m_spinBlue;
BYTE m_ucBlue;
BYTE m_ucGreen;
int m_nGrosime;
int m_nrx;
int m_nry;
BYTE m_ucRed;
int m_nx;
int m_ny;
//}}AFX_DATA


// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDialogIntroducereElipsa)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
        // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
//{{AFX_MSG(CDialogIntroducereElipsa)
virtual BOOL OnInitDialog();
//}}AFX_MSG
```

```
DECLARE_MESSAGE_MAP()
};


// DialogIntroducerePatrat.h : header file
//

/////////////////////////////////
// CDialogIntroducerePatrat dialog

class CDialogIntroducerePatrat : public CDialog
{
// Construction
public:
CDialogIntroducerePatrat(CWnd* pParent = NULL);
        // standard constructor

// Dialog Data
//{{AFX_DATA(CDialogIntroducerePatrat)
enum { IDD = IDD_DIALOG_PATRAT };
CSpinButtonCtrl m_spinRed;
CSpinButtonCtrl m_spinGrosime;
CSpinButtonCtrl m_spinGreen;
CSpinButtonCtrl m_spinBlue;
CSpinButtonCtrl m_spiny;
CSpinButtonCtrl m_spinx;
CSpinButtonCtrl m_spinl;
int m_nl;
int m_nx;
int m_ny;
BYTE m_ucBlue;
BYTE m_ucGreen;
int m_nGrosime;
BYTE m_ucRed;
//}}AFX_DATA


// Overrides
// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CDialogIntroducerePatrat)
protected:
```

```
virtual void DoDataExchange(CDataExchange* pDX);
        // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:

// Generated message map functions
//{{AFX_MSG(CDialogIntroducerePatrat)
virtual BOOL OnInitDialog();
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
```

Într-adevăr dacă ne uităm în codul sursă aceste clase sunt clase de date. Ele au rolul de a copia datele din documentul gestionat de aplicaţie, îl supun modificărilor prin intermediul cutiilor de dialog şi dacă sunt confirmate, sunt copiate înapoi în document. Posibilităţi de ameliorare a carenţei ar putea fi folosirea unui tipar de stare, sau cel puţin adăugarea de metode accesor membrilor supuşi modificărilor. Se mai poate comenta faptul că aceste clase detectate nu au fost suspecte din punctul de vedere al metricii NOAM. Acest lucru resultă din ultima coloană a tabelului. Metrica care a determinat căderea acestor clase în grupul de suspecţi a fost numărul de atribute publice. De accea este necesar să vedem şi care alte clase sunt la limită.

Dacă testăm proiectul cu strategia *Shotgun Surgery*, care detectează clasele cele mai dependabile dintr-un proiect, obţinem:

Entity Name	WCM	CC
CPrimitiva	64	6

```
class CPrimitiva:public CObject
{
public:
int m_nTip;
int m_nGrosime;
unsigned char m_ucRed;
unsigned char m_ucGreen;
unsigned char m_ucBlue;
CPrimitiva();
```

```
virtual ~CPrimitiva();
virtual void Serialize(CArchive&);
virtual void Deseneaza(CDC*);
virtual void DeseneazaColorat(CDC*);
virtual bool PtInInterior(CPoint);
virtual void Selecteaza(CDC*);
virtual void Deplaseaza(int,int);
};
```

Detecţia realizată este corectă din punctul de vedere al motorului de detecţie. Dacă modificăm clasa *CPrimitiva* se vor modifica foarte multe clase cum ar fi: *CDreptunghi*, *CElipsa*, *CPatrat*, *CCerc*. Dar clasa CPrimitiva a fost introdusă ca parte stabilă a proiectului, ca abstractizare în modelarea primitivelor grafice. De fapt este o clasă abstractă, cu rol de interfaţă. Nu este o clasă pură, dar nu se vor instanţia obiecte din ea.

Legat de strategia de detecţie a claselor de date, se poate defini o strategie de detecţie a claselor pur funţionale. Ele se găsesc exact la polul opus faţă de clasele de date, conţinând comportamentul datelor din clasele de date. Strategia *God Classes* nu a detectat pe proiectul analizat nici o astfel de clasă.

Altă analiză putem face pe un proiect *AplicatieCompilator*, realizat in C++ dar da data asta se foloseşte librăria grafica *Qt* a firmei *Troll*. Voi încerca să pun accent pe complexitatea unui aspect care este greu de observat cu ochiul liber. Aplicaţie este un compilator cu 56 de clase şi 12000 linii. Clasele sunt destul de aglomerate, conţin metode care verifică regulile lexicale şi sintactice ale unui compilator Pascal. Carenţa de proiectare care s-a făcut este omiterea unui nivel de abstractizare în modelarea unui dispozitiv de afişare a informaţiilor rezultate în urma analizei fişierului sursă. Sunt interesat de un rezultat al strategiei *Shotgun Surgery*. Strategia scoate următoarele clase:

Entity Name	WCM	CC
Atom	185	20
DispozitivAfisare	126	27

Clasa *Atom* este o clasă abstractă, aici nu sunt de făcut comentarii. Clasa *DispozitivAfisare* este cea care merită comentată. În proiect sunt două instanţieri ale acestei clase dar este foarte frecvent utilizată de celelalte clase.

```
void AplicatieCompilatorDoc::
analizorLexical(QMultiLineEdit* multiLineEdit)
{
        save(multiLineEdit);
        QDialogAfisareRezultat *dlg=
new QDialogAfisareRezultat(0,"Analizor Lexical");

        DispozitivAfisare *pDispozitivAfisare=
         new DispozitivAfisare(dlg->getMultiLineEdit());
        pDispozitivAfisare->adaugaContinut("Incepem...");
        Compilator* pCompilator=new Compilator(szFileName,
                                               pDispozitivAfisare);
        pCompilator->analizeazaLexical();
        pDispozitivAfisare->adaugaContinut("Oprim...");
        delete pDispozitivAfisare;

        dlg->exec();
        delete dlg;
}
```

Fără PRODEOOS ar dificil de găsit astfel de clase pentru că operaţia are necesita timp imens. Imaginaţi-vă un grup de ingineri care analizează cod de 12000 de linii de cod şi observă această carenţă. Oare cât ar dura această căutare. Cu *Prodeoos* acest proces durează sub 5 minute. Chiar dacă am presupune că acel grup de ingineri ar avea strategia gata definită, prelucrarea datelor este o muncă deprimantă. Chiar dacă instrumentul software nu poate înţelege filosofia proiectului, prin amprenta pe care o lasă programatorul, poate stabili unde este necesară o refactorizare.

Capitolul 6

Concluzii. Proiecte de viitor

6.1 Concluzii

Utilizând instrumentul software câştigăm:

- *abordare sistematică* prin descrierea strategiilor în limbajul SOD
- *repetabilitate* bazată pe faptul că strategiile de detecţie pot fi reutilizate, parametrii pot varia, dar strategie rămâne aceeaşi
- *scalabilitate* asigurată de instrumentul software - PRODEOOS cât şi de rezultatele obţinute pe sisteme industriale.

6.2 Proiecte de viitor - legate de instrumentul software

Traducerea în format XML a strategiilor de deteţie XML (eXtensible Markup Language) este un subset al SGML care are scopul să permită documentelor scrise în format SGML să fie trensmise, recepţionate şi procesate pe Web asą cum este posibil cu HTML-ul. Scopul pentru care a fost proiectat este uşurinţa implementării şi interoperabilitate între SGML şi HTML. În forma în care sunt acum strategiile de deteţie pot fi traduse în format XML până la nivelul metricilor. Putem să gândim mai departe: dacă metamodelul ar fi şi el în format XML (bazele de date pot fi foarte uşor reprezentate în XML) atunci practic toată tehnica de deteţie se reduce la un format XML. Acest format nu depinde de platformă, şi poate fi folosit şi de alte aplicaţii interesate de acest domeniu.

Modelarea metricilor prin intermediul unor clase java Deoarece metricile sunt modelate cu ajutorul limbajului SQL, putem spune că acest lucru reduce uşor din flexibilitate dacă facem o comparaţie cu Java, din acest punct de vedere. Pentru a crea o strategie de deteţie nouă utilizatorul trebuie să cunoască structura metamodelului (lucru relativ simplu) şi să ştie să scrie script SQL pe baza lui. Până să gasim un motor de baze de date convenabil, care să suporte SQL standard, am încercat la rând MySql, Postgres, InterBase, Sybase. Primele trei s-au dovedit a fi incomplete în ceea ce priveşte sintaxa unor comenzi (SELECT FROM SELECT). Cu toate că strategiile de deteţie nu depind de motorul ales, poate ar fi mai convenabil crearea unui metamodel bazat pe obiecte Java. Spre deosebire de înregistrări care sunt uneori redundante, obiectele permit accesul la date într-un mod mai direct, de exemplu dacă ne interesează informaţiile despre clasa părinte a unei clase date nu mai trebuie să localizăm clasa dorită printr-o comanda SELECT şi apoi clasa părinte prin altă comandă SELECT. Accesul s-ar putea realiza cu ajutorul a două referinţe: una în lista de clase şi alta în clasa dată. Este adevărat că astfel ar putea să crească dimensiunea metamodelului dar câştigăm viteză şi simplificăm complexitatea.

Modul de lucru în loturi Deoarece unele sisteme pe care rulează PRODEOOS sunt mai lente, s-ar putea introduce o facilitate de lucru în loturi prin care utilizatorul alege la început strategiile cu care doreşte să testeze sistemul software, şi apoi lasă PRODEOOS să lucreze şi să salveze rezultatele. Mai mult, raţionamentul se poate duce mai departe: s-ar putea executa pe fiecare proiect care se doreşte a fi analizat o listă specială de strategii. Pe fiecare proiect din lista de proiecte s-ar putea executa un set de strategii în funţie de specificul său.

Adaptarea instrumentului software la plug-in-uri Primul lucru la care m-am gândit când am redactat acest subcapitol sunt: *operatorii statistici*. De aici mi-au mai venit şi alte idei pe care le voi prezenta în continuare. Operatorii statistici operează pe o structură de date liniară calculând diferite praguri şi limite pentru a putea extrage elementele cu valori extreme. Deci ele nu depind decât de structura datelor pe care operează. Această structură a fost fixată de o interfaţă care oferă clienţilor săi acces la valorile elementelor.

Integrarea instrumentului software într-un mediu de dezvoltare Un instrument CASE (Computer Aided Software Enginnering) poate fi binevenit într-un mediu de dezvoltare. Ne putem gândi la un mediu open-source cum ar fi JEdit care se poate descărca de pe Internet sau la unul comercial

ca al firmei Intelij, Idea.

Integrarea instrumentului software într-un server de aplicatii Web Cea mai în vogă abordare a ofertei serviciilor informaţionale este serverul de aplicaţii. Am imaginat o parte de client care să construiască metamodelul din sursele beneficiarului şi să se conecteze la serverul de aplicaţii care să se ocupe de deteţii, după care rezultatele să fie transmise la beneficiar. Construcţia metamodelului se face pe partea de client pentru a nu periclita intimitatea codului. Deoarece PRODEOOS rulează pe platformă Java, serverul de aplicaţii ar putea fi unul j2ee: cel oferit de firma Sun Microsystems sau JBoss, care se găsesc gratis pe Internet.

Salvarea rezultatelor în format XML Salvarea rezultatelor în diferite formate poate fi făcută cu foarte mare uşurinţă datorită interfeţelor care abstractizează:

- *arborele de detecţie* prin intermediul interfeţei nodurilor
- *seturile de rezultate* prin intermediul interfeţei vectorilor de entităţi.

Salvarea în formatul XML are ca scop facilitarea interpretării rezulatelor de altăplicaţie, de pe altă platformă, în absenţa instrumentului software.

Evidenţierea sintaxei (Highlight) Vizualizatorul de surse s-ar putea îmbunătăţi cu o astfel de facilitate. Codul s-ar vedea mult mai clar, şi localizarea entităţilor s-ar realiza mult mai uşor. Monitorul SQL s-ar mai putea bucura de o asemenea facilitate, la editarea script-urilor, oferind un minim de control asupra corectitudinii.

Anexa A

Tabela de accese

Tabela A.1: Tabela detaliată a acceselor

Câmp	Descriere	Comentarii
f_module	numele complet al fişierului unde are loc accesul variabilei incluzând calea până la acel fişier	
f_start	linia unde începe accesul	
f_start_char	poziţia caracterului unde începe accesul	
f_stop	numărul liniei unde se termină accesul	
f_stop_char	poziţia caracterului unde se termină accesul	
f_class	numele clasei unde are loc accesul	
f_function	numele funcţiei unde are loc accesul	Pentru constructori numele clasei prefixeazaă numele funcţiei
f_signature	signatura funcţiei unde are loc accesul	Acest câmp este vid dacă funcţia nu are parametri
f_name	numele variabilei accesate	
f_type	tipul variabilei accesate	
f_provider_class	numele clasei unde variabila accesată a fost definită	

Câmp	Descriere	Comentarii
f_use	indică ce tip de variabilă a fost accesată (parametru, variabilă globală, atribut)	Această coloană trebuie să aibă una dintre următoarele valori: • *global* pentru variabile globale • *param* pentru parametri • *local* pentru variabile locale • *attr-public* pentru atribute publice • *attr-private* pentru atribute private • *attr-protected* pentru atribute protejate
f_is_static	specifică dacă variabila este statică sau nu	• *1* dacă este statică • *0* dacă nu este statică
f_is_complex	specifică dacă tipul variabilei este unul predefinit sau unul definit de utilizator	• *1* dacă tipul este definit de utilizator, o clasă de exemplu • *0* dacă tipul este predefinit int, char

Câmp	Descriere	Comentarii
f_is_interface	diferenţiază accesul la o interfaţă de unul la o clasă	
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.	Câmpul este folositor pentru măsurarea gradului de cuplare dintre subsisteme
f_provider_package	numele subsistemului unde variabilă este definită	Câmpul este folositor pentru măsurarea gradului de cuplare dintre subsisteme
f_how_many	numărul de accese la aceeaşi variabilă în aceeaşi funcţie	

Anexa B

Tabela de apeluri

Tabela B.1: Tabela detaliată a apelurilor

Câmp	Descriere	Comentarii
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier	
f_start	linia unde începe accesul	
f_start_char	poziţia caracterului unde începe accesul	
f_stop	numărul liniei unde se termină accesul	
f_stop_char	poziţia caracterului unde se termină accesul	
f_class	numele clasei unde are loc accesul	Dacă spaţiul de nume al clasei nu este global atunci acest câmp conţine *numespatiu::numeclasa*
f_function	numele funcţiei unde are loc accesul	
f_signature	signatura funcţiei unde are loc accesul	Acest câmp este vid dacă funcţia nu are parametri

Câmp	Descriere	Comentarii
f_access_specifier	specifică tipul de funţie unde are loc accesul	Coloana trebuie săibăuna dintre valorile următoare: • *single-function* pentru funcţii globale • *public-method* pentru metode publice • *private-method* pentru metode private • *attr-protected* pentru metode protejate
f_called_class	numele clasei unde are loc accesul	Dacă spaţiul de nume al clasei nu este global atunci acest câmp conţine *numespatiu::numeclasa*
f_called_function	numele funcţiei unde are loc accesul	
f_called_signature	signatura funcţiei unde are loc accesul	Acest câmp este vid dacă funcţia nu are parametri

Câmp	Descriere	Comentarii
f_called_access_specifier	specifică tipul de funcţie unde are loc accesul	Coloana trebuie să aibă una dintre valorile: • *single-function* pentru funcţii globale • *public-method* pentru metode publice • *private-method* pentru metode private • *attr-protected* pentru atribute protejate • *library-function* pentru funcţii folosite dar nedefinite în proiect
f_how_many	numărul de accese la aceeaşi variabilă în aceeaşi funcţie	
f_is_overloaded	indică dacă funcţia specificată este supraîncărcată sau nu. O funcţie este supraîncărcată dacă există mai multe implementări având acelaşi număr de parametri.	

Câmp	Descriere	Comentarii
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.	
f_called_package	numele pachetului unde funcţia apelată este definită	pentru funcţii de librărie acest câmp este vid

Anexa C

Tabela de clase

Tabela C.1: Tabela detaliată a claselor

Câmp	Descriere	Comentarii
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier	
f_start	linia unde începe accesul	
f_start_char	poziţia caracterului unde începe accesul	
f_stop	numărul liniei unde se termină accesul	
f_stop_char	poziţia caracterului unde se termină accesul	
f_class	numele clasei unde are loc accesul	
f_scope	pentru clase interne, acest câmp conţine numele clasei gazdă care conţine clasa internă	
f_is_abstract	specifică dacă clasa este abstractă	
f_is_template	specifică dacă clasa este generică	

Câmp	Descriere	Comentarii
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.	
f_namespace	numele spaţiului unde clasa este definită	

Anexa D

Tabela de declaraţii

Tabela D.1: Tabela detaliată a declaraţiilor

Câmp	Descriere	Comentarii
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier	
f_start	linia unde începe accesul	
f_start_char	poziţia caracterului unde începe accesul	
f_stop	numărul liniei unde se termină accesul	
f_stop_char	poziţia caracterului unde se termină accesul	
f_namesepace	numele spaţiului unde are loc accesul	Dacă spaţiul de nume este cel implicit atunci câmpul este vid
f_class	numele clasei unde are loc accesul	Acest câmp este vid pentru variabilele globale în C++

Câmp	Descriere	Comentarii
f_function	numele funcţiei unde are loc accesul	Acest câmp este vid pentru variabilele globale în C++. Pentru constructori numele clasei prefixează numele funţiei.
f_signature	signatura funcţiei unde are loc accesul	Acest câmp este vid dacă funcţia nu are parametri. Acest câmp este vid pentru variabilele globale în C++.
f_name	numele variabilei declarate	
f_type	tipul de bază al variabilei	
f_type_compl	tipul complet al variabilei	În acest câmp tipul este reprezentat exact aşa cum apare el în codul sursă.

Câmp	Descriere	Comentarii
f_use (f_access_specifier)	specifică tipul de variabilă	Coloana trebuie să aibă una dintre valorile: • *single-function* pentru funcţii globale • *public-method* pentru metode publice • *private-method* pentru metode private • *attr-protected* pentru atribute protejate • *library-function* pentru funcţii folosite dar nedefinite în proiect
f_is_complex	specifică dacă tipul variabilei e unul predefinit sau unul definit de utilizator	Această coloană poate avea valorile • *1* dacă tipul este definit de utlizator • *0* dacă tipul este predefinit

Câmp	Descriere	Comentarii
f_is_template	specifică dacă tipul variabilei este generic	Această coloană poate avea valorile • *1* dacă tipul variabilei este generic • *0* dacă tipul variabilei este predefinit
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.	

Anexa E

Tabela de funcţii

Tabela E.1: Tabela detaliată a funcţiilor

Câmp	Descriere	Comentarii
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier	
f_start	linia unde începe accesul	
f_start_char	poziţia caracterului unde începe accesul	
f_stop	numărul liniei unde se termină accesul	
f_stop_char	poziţia caracterului unde se termină accesul	
f_class	numele clasei unde are loc accesul	Acest câmp este vid pentru variabilele globale în C++. Pentru constructori numele clasei prefixează numele funţiei.

Câmp	Descriere	Comentarii
f_function	numele funcţiei unde are loc accesul	Acest câmp este vid pentru variabilele globale în C++
f_signature	signatura funcţiei unde are loc accesul	Acest câmp este vid dacă funcţia nu are parametri
f_return	tipul obiectului returnat de funcţie	Tipul returnat este prefixat cu specificatorul 'const' dacă funcţia a fost definită astfel.

Câmp	Descriere	Comentarii
f_use (f_access_specifier)	specifică tipul de variabilă	Coloana trebuie să aibă una dintre valorile: • *single-function* pentru funcţii globale • *public-method* pentru metode publice • *private-method* pentru metode private • *attr-protected* pentru atribute protejate • *library-function* pentru funcţii folosite dar nedefinite în proiect

Câmp	Descriere	Comentarii
f_storage (f_storage_specifier)	specificatorul de depozitare	Coloana trebuie să ia una dintre valorile: • *vid* pentru funcţiile uzuale • *virtual* pentru funcţii virtuale • *static* pentru funcţii statice • *const* dacă funcţia este statică
f_ct_cyclo	valoarea numărului ciclomatic al unei funcţii	Aceasta este o metricăprocedurală definită de McCabe, cunoscută sub numele de *complexitate ciclomatică*.
f_package	pentru proiectele C++ acest câmp va conţine calea relativă la fişierul sursă. Pentru proiectele Java acesta va avea semantica obişnuită.	

Anexa F

Tabela de moştenire

Tabela F.1: Tabela detaliată a moştenirilor

Câmp	Descriere	Comentarii
f_module	numele complet al fişierului unde are loc accesul variabilei încluzând calea până la acel fişier	
f_start	linia unde începe accesul	
f_start_char	poziţia caracterului unde începe accesul	
f_stop	numărul liniei unde se termină accesul	
f_stop_char	poziţia caracterului unde se termină accesul	
f_class	numele clasei unde are loc accesul	
f_parent	numele clasei strămoş	Tabela conţine închiderea tranzitivă a relaţiei de moştenire.
f_attribute	specifică modul cum subclasa este derivată din superclasă. Acest atribut influenţează vizibilitatea membrilor din clasa părinte spre clasa derivată.	

Câmp	Descriere	Comentarii
f_ct_dit	specifică distanţa î arborele de moştenire dintre clasa părinte şi cea derivată	

Bibliografie

[1] H. Schildt: *C++, Manual complet*, Editura TEORA, 1997.

[2] G. Booch: *Object-Oriented Analysis and Design with Applications*, Second Edition, Addison-Wesley, 1994.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1997.

[4] Robert Martin: *Design Principles and Patterns.* http://www.objectmentor.com, 2000.

[5] Fowler Martin: *Refactoring.* Second Edition, Addison-Wesley, 1999.

[6] Radu Marinescu: *Detecting Design Flaws via Metrics in Object-Oriented Systems.* Proceedings of the TOOLS USA 2001, IEEE Computer Society, 2001 (to be published).

[7] Radu Marinescu: *Design Flaws and Detection Strategies.* PhD Presentation, Karlsruhe, April 2001.

[8] Brad Appleton: *Patterns and Software: Essential Concepts and Terminology*, 1997.

[9] Bruce Eckel: *Thinking in Patterns with Java*, Revision 0.5a, 2000.

[10] Ioan Jurca: *Programarea reţelelor de calculatoare*, Editura de Vest, Timişoara 2000.

[11] Ciprian-Bogdan Chirilă: *Automatizarea procesului de detecţie a carenţelor de proiectare în sisteme orientate obiect*, Sesiunea de comunicări ştiinţifice studenţeşti, Timişoara 2001.

[12] Mark Watson: *Aplicaţii* JAVA *inteligente pentru Internet şi intraneturi*, Editura ALL EDUCATIONAL, Bucureşti 1999.

[13] Tim Evans: *10 minute HTML*, Editura Teora 1996.

[14] Paul A. Blaga, Horia F. Pop: LaTeX 2_ε, Editura Tehnică, Bucureşti,1999