UNIVERSITATEA „POLITEHNICA” DIN TIMISOARA
FACULTATEA DE AUTOMATICA SI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

Optimizarea detectiei automate a carentelor de proiectare în sistemele software orientate pe obiecte

Lucrare de diploma

Petru Florin Mihancea

Conducator stiintific:
dr. ing. Radu Marinescu

- Iunie 2003 -

Motto :
"Use numbers ... but never trust them!"

Cuprins

Introducere 9

1.1 Motivatia 9

1.1.1 Detectia carentelor de proiectare 9

1.1.2 Definirea problemei 11

1.2 Scopul 12

1.3 Organizarea lucrarii 12

Organizarea orientata pe obiecte a sistemelor software 15

2.1 Definitii 15

2.2 Concepte fundamentale 17

2.2.1 Abstractiunea 17

2.2.2 Ascunderea informatiei 18

2.2.3 Încapsularea 18

2.2.4 Modularizarea 19

2.2.5 Interfata 19

2.2.6 Clasa 20

2.2.7 Legarea si polimorfismul 20

2.2.8 Ierarhizarea 21

2.3 Proprietatile sistemelor software organizate orientate pe obiecte 22

2.4 Dezideratele proiectari obiectuale 22

2.4.1 Cuplaj redus 23

2.4.2 Coeziune ridicata 24

2.4.3 Complexitate gestionabila 24

2.4.4 Abstractizare corespunzatoare 25

2.5 Principii si tipare de proiectare 25

2.5.1 Proiectarea dirijata de schimbare 25

2.5.2 Principii de proiectare 26

2.5.3 Tipare de proiectare 29

Detectia carentelor de proiectare în sistemele software organizate orientat pe obiecte 33

3.1 Carente de proiectare 33

3.1.1 Definire 33

3.1.2 Cauzele aparitiei carentelor de proiectare 33

3.1.3 Simptoamele carentelor de proiectare 34

3.1.4 Clasificarea carentelor de proiectare 36

3.1.5 Exemple 37

3.2 Detectia carentelor de proiectare 37

3.2.1 Necesitatea 37

3.2.2 Metode de detectie informale 39

3.2.3 Dezavantajele metodelor informale 40

3.3 Elemente necesare detectiei sistematice 41

3.3.1 Simptoame vs. Semne 41

3.3.2 Metrici software 42

3.3.3 Metrici software în contextul orientarii pe obiecte 43

3.3.4 Cuantificarea principiilor de proiectare 46

3.3.5 Detectia carentelor de proiectare utilizând metrici software 47

3.4 Strategii de detectie 47

3.4.1 Definire 47
3.4.2 Mecanismul de filtrare 48
3.4.3 Mecanismul de compozitie 49
3.4.4 Exemple 50
3.4.5 Definirea si utilizarea strategiilor de detectie 51
3.4.6 Problema pragurilor 52

Conceptul de masina de tuning a strategiilor de detectie 55
4.1 Un proces de definire incrementala a strategiei de detectie 55
4.2 Atacarea problemei 58
4.3 Masina de tuning a strategiilor de detectie 59
4.3.1 Definire 59
4.3.2 Meta - arhitectura 59
4.3.2.1 Depozitul de exemple 60
4.3.2.2 Analizorul de descriptori 61
4.3.2.3 Componenta de masurare 62
4.3.2.4 Componenta de reglare 62
4.3.2.5 Depozitul de rezultate 63
4.3.2.6 Validatorul 64
4.3.2.7 Functionarea masinii 64
4.3.3 Considerente de implementare ale algoritmului de tuning 66
4.3.3.1 Algoritmi de îmbunatatire iterativa 66
4.3.3.2 Algoritmi genetici 67

Implementare si evaluare. Prototipul LRG – DSTM 71
5.1 Implementarea conceptului 71
5.1.1 Serverul de baze de date MySQL 72
5.1.2 Evaluatorul 73
5.1.3 Algoritmul de tuning 75
5.1.4 Interfata cu utilizatorul 81
5.2 Evaluare 82
5.2.1 Procesul de selectie a exemplelor 84
5.2.2 Strategia de detectie GodClasses 85
5.2.3 Strategia de detectie DataClasses 89

Concluzii si perspective 99
6.1 Concluzii 99
6.2 Sumarul contributiei 100
6.3 Perspective 100
6.3.1 Alte experimente de evaluare 101
6.3.2 Rafinarea implementarii 102
6.3.3 Alte idei 103

Bibliografie 105

Anexa A. Catalog cu descrieri informale ale carentelor de proiectare ce apar în sistemele software orientate pe obiecte 107

Anexa B. Solutiile de parametrizare pentru scheletul strategiei de detectie GodClasses 110

Anexa C. Solutiile de parametrizare pentru scheletul strategiei de detectie DataClasses...111

Capitolul I

Introducere

1.1 Motivatia

Organizarea orientata pe obiecte a unui sistem software poate furniza acestuia o serie de proprietati mult dorite de orice producator de programe: flexibilitate ridicata, extensibilitate sporita si ca urmare posibilitatea întretinerii usoare. Totusi, simpla utilizare a mecanismelor specifice tehnologiei orientata pe obiecte, cum ar fi polimorfismul sau încapsularea, nu conduce automat la obtinerea acestor deziderate. Toate aceste mecanisme trebuie utilizate în corelatie cu respectarea anumitor principii de proiectare obiectuala, acestea fiind cele care în final asigura proprietatile mai sus amintite. Încalcarea principiilor de proiectare va conduce la sisteme software monolitice, inflexibile si foarte greu de întretinut. Imaturitatea proiectantilor, care nu înteleg importanta respectarii acestor principii, este doar una dintre cauzele aparitiilor acestor situatii nedorite. Dupa cum arata F.P. Brooks Jr. în [3] necesitatea de modificare a software-ului este o proprietate de esenta a acestuia. Mai mult, aceste modificari trebuie executate într-un timp scurt si sunt realizate de alte persoane, care nu înteleg "filosofia" design-ului initial. Prin urmare, chiar daca la început design-ul sistemului software este foarte bun, în timp el se altereaza ajungându-se la aceeasi lipsa de flexibilitate si în final la imposibilitatea adaugarii de noi functionalitati. În consecinta, pentru ca un sistem software sa-si mentina valoarea economica, el trebuie supus unui proces de reorganizare în vederea recâstigarii, cel putin în parte, a proprietatilor de flexibilitate, extensibilitate si întretinere usoara. Acest proces de reorganizare este cunoscut în literatura de specialitate sub numele de proces de reengineering.

1.1.1 Detectia carentelor de proiectare

O etapa critica din cadrul procesului de reengineering consta în detectia problemelor sau carentelor de proiectare. În esenta, aceasta activitate consta în identificarea acelor entitati de design ce violeaza anumite principii de proiectare periclitând astfel evolutia sistemului software. Metoda traditionala de abordare a detectiei carentelor de proiectare sufera de un dezavantaj major: operatia este realizata într-o maniera manuala. Din acest motiv, detectia carentelor de proiectare este costisitoare ca timp, nu e repetabila si ce este mai grav, nu e scalabila. O metoda manuala de detectie a problemelor este aproape imposibil de aplicat în cazul sistemelor software de mari dimensiuni.

În lucrarea "Measurement and quality in object-oriented design" R. Marinescu propune o metoda de detectie a carentelor de proiectare bazata pe metrici software. Actualmente, exista definite metrici capabile sa cuantifice cele mai importante atribute asociate design-ului unui sistem software: cuplajul, coeziunea, complexitatea, etc. Pe baza modelelor de interpretare si a valorilor respectivelor metrici este posibil sa identificam deviatii de la un set de criterii ce caracterizeaza un design de calitate: cuplaj redus, coeziune ridicata, complexitate gestionabila. Totusi, utilizarea metricilor software în contextul detectiei carentelor de proiectare nu este usoara. Modelul de

interpretare asociat unei metrici ne poate ajuta sa identificam simptomul unei carente de proiectare reflectat în valorile anormale ale respectivei metrici, dar nu ne poate ajuta sa identificam "boala" ce cauzeaza respectivul simptom. Motivul principal al acestei deficiente consta în faptul ca un acelasi simptom, spre exemplu complexitate foarte mare, poate apare în contextul mai multor carente de proiectare distincte. Cu alte cuvinte, o singura metrica software are o granularitate prea mica pentru a putea identifica carenta de proiectare propriu-zisa ce afecteaza o anumita entitate de design. Concluzionând, exista o distanta prea mare între lucrurile pe care le masuram si lucrurile cu adevarat importante din punctul de vedere al proiectarii.

Pentru utilizarea eficienta a metricilor software în contextul detectiei carentelor de proiectare, R. Marinescu introduce un nou concept: strategia de detectie. La baza lui sta ideea utilizarii simultane a mai multor metrici care împreuna sa surprinda în totalitate aspectele relevante din contextul unei carente de proiectare. Prin posibilitatea de reprezentare într-o forma cuantificata a acestor aspecte, strategia de detectie face posibila detectia sistematica a carentelor de proiectare si face posibila executia într-o maniera automata a fazei de detectie a problemelor din cadrul procesului de reengineering.

Abstract, o strategie de detectie reprezinta o expresie cuantificata a unei reguli prin care fragmente de design conforme cu respectiva regula pot fi detectate în codul sursa. La baza unei strategii de detectie se gasesc doua mecanisme esentiale: mecanismul de filtrare si mecanismul de compozitie. Astfel, din punct de vedere anatomic, o strategie de detectie este constituita din urmatoarele elemente: metrici software, filtre de date si operatori de compozitie. Fiecare metrica software are asociat un filtru de date care permite identificarea unui set de entitati de design masurate de metrica asociata si care prezinta o valoare anormala pentru respectiva metrica. Dupa cum am spus la început, o strategie de detectie este o expresie cuantificata a unei reguli. Spre deosebire de metrici, o strategie de detectie trebuie sa cuantifice întreaga regula nu doar un aspect al ei. În consecinta, pe lânga mecanismul de filtrare care interpreteaza rezultatele masuratorilor individuale, mai este necesar un alt mecanism care sa suporte o interpretare corelata a mai multor seturi de rezultate. Acesta este mecanismul de compozitie. Operatorii de compozitie compun perechile metrica-filtru de date si pot fi priviti din doua perspective. Din punct de vedere logic, operatorii sunt o reflectie a "punctelor de articulatie" din cadrul regulii cuantificate de strategia de detectie, în care "operanzii" sunt descrieri ale simptoamelor carentelor de proiectare. Spre exemplu, operatorul AND sugereaza necesitatea prezentei atât a simptomului descris în partea dreapta a operatorului cât si a celui din partea stânga. Din punctul de vedere al operatiilor cu seturi, se întelege modul în care se construieste rezultatul final în urma aplicarii unei strategii de detectie. Operatorul AND corespunde unei operatii de intersectie dintre setul de entitati furnizat de perechea metrica-filtru din partea dreapta si setul de entitati furnizat de perechea din partea stânga. În urma aplicarii unei strategii de detectie se obtine ca rezultat un set de entitati de design suspectate a fi conforme cu regula cuantificata de respectiva strategie. Este necesara o inspectie manuala a acestor entitati pentru a vedea care anume sunt suspectate în mod corect si care sunt suspectate eronat.

Utilizarea conceptului de strategie de detectie în vederea detectiei problemelor de proiectare din sistemele software orientate pe obiecte s-a dovedit benefica. Tot mai multi cercetatori si ingineri din domeniul reingineriei software accepta acest mecanism ca un instrument util în detectia carentelor de proiectare dar si în vederea rezolvarii unor probleme de alta natura. În acelasi timp, se dezvolta tot mai multe instrumente software dedicate operatiei de detectie a problemelor din cadrul

procesului de reengineering si care utilizeaza conceptul de strategie de detectie. În paralel, se definesc noi strategii de detectie pentru identificarea diverselor carente de proiectare ce pot apare în sistemele software orientate pe obiecte. Din pacate, o carenta cunoscuta a strategiilor de detectie actuale consta în existenta unor situatii în care rezultatele furnizate nu au o acuratete multumitoare. Vom expune aceasta problema în sectiunea urmatoare.

1.1.2 Definirea problemei

M. Fowler apreciaza în [7] ca "nici un set de metrici nu poate rivaliza cu intuitia umana". Într-adevar, dupa cum am mai amintit, o strategie de detectie furnizeaza entitati de design suspectate de a fi afectate de o anumita carenta de proiectare. Între ele pot exista entitati de design care nu sunt afectate de carenta pe care strategia spune ca o identifica. Totusi, suspectarea eronata a unei entitati de design nu este foarte grava în conditiile în care numarul acestor situatii nu e foarte mare. La dimensiunile sistemelor software orientate pe obiecte din zilele noastre, detectia manuala a carentelor de proiectare este extrem de dificila si este mare consumatoare de timp. În aceste conditii, preferam sa utilizam strategiile de detectie si sa analizam manual setul entitatilor de design suspectate în vederea identificarii carentelor reale. Acesta reprezinta un subset considerabil mai mic decât setul tuturor entitatilor similare din sistem, necesitând astfel un timp de analiza mult mai mic. Situatia ar putea deveni neplacuta doar în momentul în care prea multe entitati de design sunt suspectate într-o maniera gresita.

Problema cu adevarat grava legata de utilizarea strategiilor de detectie în contextul detectie carentelor de proiectare este însa alta. Exista destule situatii în care entitati de design afectate de o carenta de proiectare nu sunt detectate de strategia de detectie care ar fi trebuit sa faca acest lucru. Aceste situatii sunt mult mai grave decât cele amintite în paragraful anterior deoarece pot perturba procesul de reorganizare a sistemului. Principial, motivul aparitiei acestor situatii ar consta în faptul ca metricile software urmarite de strategia de detectie nu surprind toate aspectele carentei de proiectare vizate. Prin urmare, ar fi necesara completarea strategiei cu alte metrici software ce sa captureze si aspectele neglijate pâna la momentul respectiv. Exista însa si un motiv mult mai complex ce ar putea fi responsabil de ratarea detectiei unor carente reale.

Dupa cum am vazut, fiecare metrica software utilizata de o strategie de detectie, are asociat un filtru de date ce permite detectia valorilor anormale ale respectivei metrici. Fiecare filtru este înzestrat cu un parametru numeric, un prag valoric, pe baza caruia se face distinctia între valorile normale si cele anormale ale respectivei metrici. În functie de tipul filtrului o valoare mai mica sau mai mare decât valoarea pragului este considerata anormala. Este clar ca stabilirea unui prag valoric necorespunzator pentru filtrele de date poate conduce atât la ratarea detectiei unor carente reale cât si la detectia unor entitati de design ce nu sunt afectate de respectiva carenta. Dupa cum am mai spus, pe noi ne deranjeaza în primul rând ratarea detectiei carentelor reale în timp ce a doua situatie trebuie mentinuta în limite adecvate. În mod natural se pune întrebarea: cum anume se stabilesc corect aceste praguri valorice? În [14] autorul indica o metodologie destinata acestei probleme. Din pacate, ea este vaga si neconvingatoare. Mai mult, autorul defineste un numar mare de strategii de detectie pentru identificarea diferitelor carente de proiectare, dar din lucrare nu reiese modul în care a fost aplicata aceasta metodologie. Acest lucru ar fi fost necesar pentru ca valorile anormale ale unei metrici software depind evident de modelul sau de

interpretare si de aspectul carentei de proiectare pe care trebuie sa-l captureze, dar pot depinde si de contextul respectivului aspect, adica de carenta propriu-zisa.

Prin urmare, putem spuna ca nu avem o metodologie clara si generala pe care sa ne bazam în momentul în care trebuie sa stabilim valoarea unui prag pentru un filtru de date. Datorita acestui fapt, este posibil ca ratarea detectiei unei carente reale de catre strategia ce o vizeaza sa se datoreze stabilirii inadecvate a parametrilor filtrelor de date utilizate de strategie.

1.2 Scopul

Scopul initial al lucrarii de fata consta în stabilirea unei modalitati concrete de determinare a valorilor pragurilor filtrelor de date. Cu alte cuvinte, trebuie sa stabilim o metodologie clara, care odata aplicata, sa permita identificarea acelor valori numerice pentru parametrii filtrelor de date utilizate în cadrul unei strategii de detectie care sa-i confere acesteia o acuratete de detectie maxima. În acelasi timp, modul de stabilire a acestor praguri trebuie sa fie unul general. Aceasta înseamna ca metodologia propusa nu trebuie sa depinda de o anumita strategie de detectie particulara sau de carenta de proiectare detectata de respectiva strategie. Ea trebuie sa functioneze indiferent de strategia de detectie pentru care este aplicata si indiferent de carenta de proiectare vizata de respectiva strategie.

În continuare va trebui sa stabilim eficacitatea metodologiei propuse. În acest sens vom selecta un numar de strategii de detectie utilizate actual în contextul detectiei problemelor. Vom aplica apoi metoda de stabilire a parametrilor pentru strategiile selectate si în final vom vedea daca noile valori permit o detectie mai precisa a carentelor de proiectare vizate de aceste strategii de detectie.

1.3 Organizarea lucrarii

În cadrul capitolului 2 se prezinta principalele aspecte legate de organizarea orientata pe obiecte a sistemelor software. Astfel, se vor descrie concepte ca abstractiunea, ascunderea informatiei, încapsularea, polimorfismul, legarea dinamica, etc., întâlnite în contextul programarii orientate pe obiecte. Dupa cum am mai spus, simpla cunoastere si utilizare a acestor concepte nu conduc la obtinerea unor proiectari de calitate. În cadrul capitolului 2 se prezinta si principalele atribute ale unei proiectari obiectuale bune. Capitolul se încheie printr-o discutie referitoare la principiile si tiparele de proiectare. Capitolul 3 se deschide prin definirea notiunii de carenta de proiectare. În continuare se identifica principalele cauze ale aparitiei lor, se prezinta simptoamele carentelor de proiectare si se indica o clasificare a acestora. Capitolul contine în acelasi timp o descriere a metodelor traditionale de detectie a carentelor de proiectare si prezinta dezavantajele acestora. În vederea introducerii conceptului de strategie de detectie, descris pe larg în acest capitol, se arata necesitatea utilizarii masuratorilor în contextul detectiei problemelor de proiectare. Capitolul contine si o descriere a mai multor metrici software.

Capitolul 4 prezinta viziunea noastra asupra procesului de definire a unei strategii de detectie. În acest context vom introduce conceptul de masina de tuning a strategiilor de detectie. În continuare vom prezenta principalele notiuni legate de acest concept si vom descrie, într-o maniera abstracta, modul de organizare si de functionare a unei astfel de masini. În finalul capitolului 4 vom purta o discutie referitoare la diferite alternative de implementare a conceptului, nu neaparat singurele

existente. Capitolul 5 se deschide printr-o prezentare a aplicatiei lucrarii noastre, a prototipului de masina de tuning realizat. Partea a doua a acestui capitol analizeaza rezultatele obtinute în urma utilizarii conceptului de masina de tuning pentru stabilirea valorilor pragurilor filtrelor de date pentru doua strategii de detectie: GodClasses si DataClasses. Pe scurt, se determina daca acest concept poate aduce îmbunatatiri în aceea ce priveste acuratetea de detectie a strategiilor mai sus amintite si, acolo unde este cazul, se prezinta carente ale acestor strategii care nu pot fi eliminate de acest concept. Aceste probleme sunt independente de masina de tuning deoarece nu sunt localizate la nivelul parametrilor filtrelor de date. Ultimul capitol contine concluziile rezultate în urma analizei de ansamblu a acestei lucrari si prezinta pe scurt modul în care vom continua munca de cercetare în perioada imediat urmatoare.

Capitolul II

Organizarea orientata pe obiecte a sistemelor software

La începutul erei object-oriented, multi producatori de software credeau ca utilizarea noii tehnologii va conduce de la sine la cresterea calitatii produselor software si la o scadere drastica a timpilor de dezvoltare si de mentenanta. S-a crezut, într-o maniera naiva, ca mecanismele specifice tehnologiei object-oriented, precum mostenirea, încapsularea sau polimorfismul, vor determina o crestere a flexibilitatii, extensibilitatii si întelegerii programelor. În zilele noastre, exista un numar mare de sisteme software utilizate în industrie, sisteme compuse din milioane de linii de cod si care prin dimensiune, complexitate si timp de dezvoltare au ajuns la o forma adecvata implementarii lor utilizând tehnologia object-oriented. Totusi, cele mai multe dintre sistemele mai sus amintite, nu prezinta caracteristicile asteptate: sunt monolitice, rigide si greu de extins.

F.P. Brooks sustine în articolul sau [3] ca complexitatea software-ului este o proprietate esentiala a acestuia, izvorâta din natura sa si nu una accidentala cauzata de metodele de producere utilizate în zilele noastre. Mai mult, el arata ca nu avem momentan nici o metoda de dezvoltare ori tehnologie care sa adreseze complexitatea esentiala a software-ului. În consecinta, cauza faptului ca tehnologia object-oriented nu a avut efectul scontat rezida în aceea ca ea rezolva o dificultate cu caracter accidental din cadrul procesului de dezvoltare a programelor. Atât ideea de tip de date abstract cât si cea de tip ierarhic, care stau la baza conceptului de programare orientata pe obiecte, elimina o dificultate accidentala. Ele permit designer-ului sa exprime mai bine esenta design-ului software-ului, eliminând toate dificultatile accidentale legate de reprezentarea lui. Complexitatea design-ului în sine este însa una de esenta si nu este adresata de aceste concepte.

Nu ne putem deci astepta de la tehnologia object-oriented sa rezolve de la sine toate problemelor legate de producerea software-ului. Totusi, organizarea orientata pe obiecte a sistemelor software poate sa ofere sistemelor anumite proprietati foarte atractive. În acest capitol vom defini organizarea orientata pe obiecte a sistemelor software, vom pune în evidenta proprietatile pe care le poate oferi acestora si vom aminti modalitatile prin care se pot obtine aceste proprietati.

2.1 Definitii

Odata cu cresterea complexitatii sistemelor software, design-ul si specificarea de ansamblu a întregii structuri a sistemului devine o problema mai importanta decât alegerea algoritmilor si a structurilor de date utilizate. Problema structurii include: organizarea sistemului sub forma unui ansamblu de componente, structuri globale de control, protocoale de comunicatie, atribuirea de functionalitati elementelor de design, compunerea elementelor de design, etc.

Definitie. Arhitectura software implica descrierea componentelor din care este construit sistemul, a interactiunilor dintre aceste elemente, a tiparelor ce ghideaza compozitia lor si a constrângerilor existente asupra acestor tipare [29].

În cadrul organizarii orientate pe obiecte sau mai general, bazate pe abstractiuni de date, reprezentarea datelor si operatiile primitive asociate lor sunt încapsulate într-un tip de date abstract sau obiect [29]. Componentele vor fi deci obiecte sau, mai exact, instante ale unui tip de date abstract si vor interactiona prin invocarea de proceduri si functii.

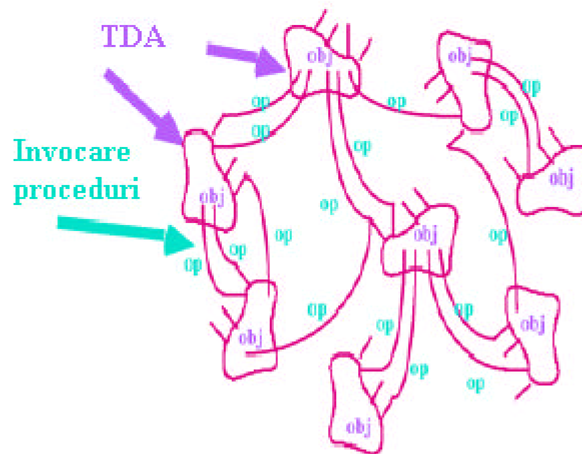

Figura 2.1 Organizarea orientata pe obiecte

Exista doua aspecte importante legate de aceasta modalitate de organizare [29]:

- un obiect este responsabil pentru pastrarea integritatii reprezentarii sale si
- o reprezentare asociata unui obiect este ascunsa în raport cu alte obiecte.

În contextul organizarii orientate pe obiecte vom defini conceptul de proiectare si programare obiectuala.

Definitie. Proiectarea orientata pe obiecte (Object-Oriented Design) este o strategie de proiectare în care proiectantii sistemului gândesc în termeni de "lucruri" si nu în termeni de operatii sau functii. Functionalitatea sistemului este obtinuta ca rezultat al interactiunii dintre obiecte care îsi mentin propria stare locala si furnizeaza operatiile asociate respectivei informatii de stare [30].

Definitie. Programarea orientata pe obiecte (Object-Oriented Programming) este o metoda de implementare prin care programele sunt organizate sub forma unor colectii de obiecte ce coopereaza între ele, fiecare obiect fiind o instanta a unei clase; fiecare clasa face parte dintr-o ierarhie de clase si între toate clasele se stabilesc anumite legaturi prin relatiile de mostenire existente între ele [2].

Analizând definitiile anterioare putem observa trei aspecte importante legate de orientarea pe obiecte:

- obiectele si nu al algoritmi sunt blocurile logice fundamentale
- fiecare obiect este o instanta a unei clase

- clasele sunt conectate între ele prin relatii de mostenire

Pentru a întelege cât mai bine cele doua definitii vom realiza în continuare o scurta comparatie între conceptele de proiectare si programare orientate pe obiecte[12].

Proiectarea orientata pe obiecte este o metoda de decompozitie a arhitecturii software. Ea se bazeaza pe obiectele pe care fiecare sistem ori subsistem le manipuleaza si este relativ independenta de limbajul de programare utilizat la implementarea propriu-zisa a programului.

Programarea orientata pe obiecte este preocupata de construirea efectiva a sistemului software sub forma unei colectii de tipuri de date abstracte, facând uz de mecanisme specifice cum sunt mostenirea si polimorfismul. Deci, programarea orientata pe obiecte este interesata de aspecte legate de limbajele de programare si de aspectele de implementare ale sistemului software organizat pe obiecte.

2.2 Concepte fundamentale

În sectiunile care urmeaza vom trece în revista principalele concepte si mecanisme pe care le întâlnim în contextul programarii orientate pe obiecte.

2.2.1 Abstractiunea

O modalitate fundamentala, utilizata de multe persoane în vederea întelegerii unei probleme complexe, o reprezinta utilizarea abstractiunilor. O buna abstractizare este aceea care retine toate aspectele relevante din perspectiva din care este analizata problema, în timp ce restul de caracteristici sunt ignorate. Atragem atentia asupra faptului ca abstractiunea trebuie privita ca o entitate logica care denota un model al unei entitati reale. Abstractizarea este un proces care se refera la operatia de identificare a caracteristicilor esentiale ale unei entitati reale si în acelasi timp a caracteristicilor ne-esentiale care urmeaza a fi ignorate [1]. În contextul nostru abstractiunea se poate defini astfel [2]:

Definitie. O abstractiune exprima toate caracteristicile esentiale care disting un obiect de un alt obiect. Abstractiunea defineste precis granitele conceptuale ale obiectului din punctul de vedere al perspectivei externe din care acesta este privit.

În procesul de creare a unei abstractiuni atentia noastra este concentrata doar asupra aspectului exterior al obiectului si asupra comportamentului sau în timp ce implementarea acestui comportament este ignorata. Cu alte cuvinte abstractiunea ne ajuta sa facem o distinctie clara între “ce face” un obiect si “cum face” respectivul obiect. Comportamentul obiectului este caracterizat printr-o suma de servicii sau resurse pe care el le ofera altor obiecte. Totalitatea serviciilor oferite de un obiect constituie contractul sau responsabilitatea sa fata de alte obiecte. Responsabilitatile obiectului sunt îndeplinite prin intermediul unor operatii, numite si metode sau functii membru. Fiecare operatie este caracterizata în mod unic de o semnatura compusa din: nume, o lista de parametrii formali si un tip de valoare returnata. Totalitatea operatiilor si regulilor corespunzatoare invocarii lor constituie asa numitul protocol al obiectului.

2.2.2 Ascunderea informatiei

În multe situatii, dupa cum se arata si în [1], conceptul de ascundere a informatiei si cel de încapsulare sunt confundate. Pentru a evita aceasta confuzie vom trata separat aceste concepte. Notiunea de ascundere a informatiei este definita de IEEE astfel:

Definitie. Ascunderea informatiei reprezinta tehnica de încapsulare a deciziilor de proiectare software în module într-o asemenea maniera încât interfata modulelor sa dezvaluie cât mai putin posibil din modul de desfasurare a activitatilor în interiorul lor. Fiecare modul devine astfel o cutie neagra din punctul de vedere al celorlalte module din sistem.

Abstractizarea este utilizata ca o tehnica de identificare a informatiilor care trebuie ascunse. De exemplu, în abstractizarea functionala am putea spune ca este important sa putem adauga elemente la o lista, dar detaliile asupra modului în care facem acest lucru nu sunt de interes si trebuie ascunse. Utilizând abstractiuni de date, vom spune ca o lista este un loc în care putem stoca informatii, dar modul în care lista este efectiv implementata este un aspect neimportant care trebuie ascuns. Atragem atentia ca abstractizarea si ascunderea informatiei nu sunt echivalente. Confuzia apare în momentul în care nu se face distinctie între ascunderea informatiei si tehnica abstractizarii utilizata la identificarea informatiei care trebuie ascunsa.

2.2.3 Încapsularea

Conceptul de încapsulare se gaseste în strânsa legatura cu notiunea de ascundere a informatiei, dar cele doua notiuni nu sunt echivalente. Poate cea mai buna definire a conceptului este data de Wirfs-Brock:

Definitie. Conceptul de încapsulare, asa cum este el utilizat într-un context object-oriented, nu este esential diferit de definitia lui din dictionar. Conceptul se refera la operatia de constructie a unei capsule, în cazul nostru o bariera conceptuala, în jurul unei colectii de lucruri.

Încapsularea, ca si proces, înseamna operatia de împachetare a unor elemente într-un container logic ori fizic. Ca si entitate, conceptul se refera la un pachet, un container ce contine unul sau mai multe elemente. Este foarte important de observat ca nu se spune nimic despre "peretii" pachetului. Ei ar putea fi transparenti, translucizi sau opaci.

Un exemplu de mecanism de încapsulare utilizat frecvent în limbajele de programare este structura sau articolul. Daca încapsularea si ascunderea informatiei ar fi una si acelasi lucru atunci orice am încapsula ar fi automat ascuns, desi informatia dintr-o structura nu este de obicei ascunsa (doar daca nu este ascunsa prin alte mecanisme).

Clasa reprezinta un alt mecanism de încapsulare care permite ca o parte din informatia continuta sa fie ascunsa, dar ea permite si ca o parte din informatie sa fie vizibila. În unele limbaje avem posibilitatea de a specifica diferite nivele de vizibilitate (ex. membrii publici, privati sau protejati din C++, Java). În discutia care urmeaza prin încapsulare vom întelege mecanismul de împachetare furnizat de clase.

Revenind la discutia despre programarea orientata pe obiecte, abstractiunile sunt utilizate pentru identificarea protocolului unui obiect, în timp ce încapsularea se ocupa cu selectia unei implementari si tratarea ei ca si un secret al respectivei abstractiuni. Procesul de încapsulare va fi vazut ca si actiunea de ascundere a implementarii obiectului fata de clientii sai (cei care apeleaza la serviciile sale) [14].

Definitie. Încapsularea este procesul de compartimentare a elementelor care formeaza structura si comportamentul unei abstractiuni; încapsularea este utilizata pentru separarea interfetei contractuale de implementarea acesteia.

Definitia de mai sus arata clar ca un obiect are doua componente: interfata obiectului si implementarea acestei interfete. Abstractizarea este procesul prin care se defineste interfata obiectului, iar încapsularea defineste reprezentarea obiectului, structura sa împreuna cu implementarea interfetei. O asemenea abordare prezinta urmatoarele avantaje:

- Separând interfata unui obiect de reprezentarea sa, putem modifica reprezentarea lui fara a afecta clientii sai pentru ca acestia depind de interfata obiectului nostru si nu de implementarea sa.

- Este posibila modificarea eficienta a programelor, limitând si localizând efortul necesar.

2.2.4 Modularizarea

Scopul descompunerii unui program în module este acela de a reduce costurile asociate operatiilor de re-proiectare si verificare permitându-ne sa realizam aceste operatii pentru fiecare modul în mod independent [25]. Clasele si obiectele obtinute dupa procesul de abstractizare si încapsulare trebuie sa fie grupate si depozitate într-un asa numit modul. Modulul poate fi privit ca un container fizic în care declaram clasele si obiectele rezultate în urma proiectarii logice a sistemului. Aceste module formeaza arhitectura fizica a programului. *Modularizarea* consta în divizarea programului într-un numar de module care vor putea fi dezvoltate si compilate separat, dar care sunt conectate, cuplate între ele. Gradul de cuplare dintre module trebuie sa fie mic pentru ca modificarile aduse unui modul sa afecteze cât mai putine alte module. Pe de alta parte, clasele care compun un modul trebuie sa aiba legaturi strânse între ele, care sa justifice gruparea lor la un loc. În acest sens, spunem ca un modul trebuie sa fie coeziv. Limbajele care suporta conceptul de modul fac în acelasi timp si distinctia între interfata modulului si implementarea sa.

2.2.5 Interfata

Asa cum am mai spus anterior, totalitatea semnaturilor definite de operatiile unui obiect se numeste *interfata* respectivului obiect. Interfata unui obiect caracterizeaza întregul set de cereri care pot fi trimise obiectului. Orice cerere care este conforma cu o semnatura din interfata obiectului poate fi trimisa acestuia. Un *tip* este un nume utilizat pentru identificarea în mod unic a unei interfete particulare [8]. Spunem despre un obiect ca este de un anumit tip daca el accepta toate cererile facute la operatiile definite în interfata asociata tipului respectiv. Un obiect poate fi de mai

multe tipuri, iar mai multe obiecte diferite pot partaja un tip. O parte din interfata unui obiect poate fi caracterizata de un tip în timp ce celelalte parti sunt caracterizate de alte tipuri. Doua obiecte de acelasi tip trebuie sa partajeze doar parti din interfetele lor.

Obiectele sunt cunoscute în interiorul sistemului numai prin interfetele lor. Singura cale prin care putem afla ceva despre un obiect sau prin care îi putem cere sa faca ceva consta în utilizarea interfetei lui. Interfata unui obiect nu spune nimic despre implementarea lui si mai mult obiecte diferite pot implementa aceeasi interfata în moduri diferite. Aceasta înseamna ca doua obiecte cu aceeasi interfata pot avea implementari total diferite.

2.2.6 Clasa

Pâna acum nu am spus nimic despre modul de definire a unui obiect. Implementarea unui obiect este definita de catre *clasa* sa. Clasa specifica datele interne si reprezentarea obiectului si defineste operatiile pe care le poate efectua un obiect. Obiectele sunt obtinute prin instantierea unei clase. Spunem despre un obiect ca este o instanta a unei clase. Procesul de instantiere consta în alocarea spatiului de memorare pentru datele interne obiectului (formate din asa numitele variabile instanta) si asocierea acestora cu operatiile lor.

Atragem atentia asupra diferentelor care exista între clasa unui obiect si tipul sau. Clasa unui obiect defineste modul de implementare al obiectului. Clasa defineste starea interna a unui obiect si implementarea operatiilor sale. În contrast, tipul unui obiect se refera doar la interfata lui, la setul de cereri la care el poate raspunde. Un obiect poate fi de mai multe tipuri si obiecte de clase diferite pot avea acelasi tip. Evident, exista o strânsa legatura între conceptul de clasa si cel de tip. Pentru ca o clasa defineste operatiile pe care un obiect le poate realiza, ea defineste în acelasi tip si tipul obiectului. Când spunem ca un obiect este instanta unei clase, inducem ca obiectul prezinta interfata definita de clasa.

2.2.7 Legarea si polimorfismul

Când o anumita cerere este trimisa unui obiect, operatia ce se va executa depinde evident de cerere dar si de obiectul receptor. Acest lucru se întâmpla pentru ca poate exista mai mult de un obiect care sa raspunde unei cereri particulare. Asa cum am mai spus, obiecte diferite care suporta aceleasi cereri pot avea implementari diferite pentru operatiile asociate respectivelor cereri. Cu alte cuvinte, operatia invocata specifica serviciul dorit, iar obiectul concret reprezinta implementarea individuala a respectivului serviciu. Asocierea dintre o operatie invocata si obiectul care va furniza implementarea concreta pentru respectiva operatie se numeste *legare* (binding). Dupa momentul în care se realizeaza aceasta asociere avem:

- Legare statica (“static binding”): asocierea este realizata în momentul compilarii.

- Legarea dinamica (“dynamic binding”): asocierea nu este creata în momentul compilarii programului, ci are loc abia în timpul executiei sale.

În contextul legarii dinamice, invocarea unei operatii nu implica o corespondenta automata între respectiva operatie si o anumita implementare a sa. Corespondenta se realizeaza în timpul executiei. Principalul avantaj al acestei abordari o constituie posibilitatea substituirii obiectelor ce au interfete identice chiar în timpul executiei. Posibilitatea utilizarii unui obiect în locul unui alt obiect când ambele partajeaza o aceeasi interfata se numeste *polimorfism*. Acest concept sta la baza programarii orientate pe obiecte.

2.2.8 Ierarhizarea

Abstractiunile de date sunt un lucru bun, dar în majoritatea aplicatiilor mari vom descoperi mai multe abstractiuni decât putem cuprinde la un moment dat. Încapsularea ne ajuta sa tratam aceasta complexitate prin ascunderea interiorului abstractiunilor noastre. Modularizarea ne ajuta si ea, oferindu-ne o modalitate de a grupa abstractiunile legate logic între ele. Toate acestea, desi utile, nu sunt suficiente. Adesea, un grup de abstractiuni formeaza o ierarhie, iar prin identificarea acestor ierarhii putem simplifica substantial întelegerea problemei. Ierarhizarea poate fi definita astfel [2]:

Definitie. Ierarhizarea este o ordonare a abstractiunilor.

Exista doua feluri de ierarhii: ierarhia de clase si ierarhia de obiecte.

Ierarhia de clase

Mostenirea defineste o relatie între clase în care o clasa partajeaza structura si comportamentul sau cu una sau mai multe clase. Existenta mecanismului de mostenire este cea care face diferenta între programarea orientata pe obiecte si programarea bazata pe obiecte. Din punct de vedere semantic mostenirea indica o relatie "este un" ("is a"). De exemplu, având doua clase A si B, A mosteneste pe B daca putem spune "A este un fel de B". Daca A "nu este un fel de B" A nu ar trebui sa-l mosteneasca pe B. În concluzie, mostenirea implica o ierarhie de tipuri generalizate / specializate în care o clasa derivata specializeaza structura si comportamentul clasei mai generale din care ea a fost derivata.

În contextul mecanismului de mostenire atragem atentia asupra urmatorului aspect: exista o diferenta majora între mostenirea de clasa si mostenirea de tip [8]. Mostenirea de clasa defineste implementarea unui obiect în termenii implementarii unui alt obiect. Cu alte cuvinte, mostenirea este un mecanism pentru partajarea codului si a reprezentarii. Mostenirea de interfata arata când un obiect poate fi utilizat în locul altui obiect. Cele doua concepte pot fi usor confundate pentru ca cele mai multe limbajele de programare nu fac distinctia explicit: în C++ mostenirea înseamna atât mostenire de clasa cât si mostenire de interfata. Mostenirea pura de interfata are loc doar când se mosteneste de la o clasa pur abstracta. În Java, conceptul de interfata ajuta la perceperea acestei distinctii.

Ierarhia de obiecte

Agregarea este o relatie între obiecte în care un obiect este parte integranta a unui alt obiect. Din punct de vedere semantic, agregarea indica o relatie de forma

"parte din". Spre exemplificare, o relatie de agregare exista între un volan si un automobil pentru ca putem spune "volanul este o parte dintr-un automobil".

2.3 Proprietatile sistemelor software organizate orientate pe obiecte

Sistemele software orientate pe obiecte sunt caracterizate de un set de proprietati atractive. Deoarece un obiect ascunde reprezentarea sa fata de alte obiecte, este posibil sa schimbam implementarea lui fara sa afectam celelalte obiecte [29]. Aceasta proprietate poate avea efecte benefice asupra cerintelor de flexibilitate si extensibilitate. Pe de alta parte obiectele sunt de multe ori reprezentari ale unor entitati din lumea reala, deci structura programului este mai usor de înteles. Mai mult, deoarece entitatile lumii reale sunt utilizate în diverse sisteme, obiectele asociate lor pot fi reutilizate [30].

Exista însa si anumite dezavantaje ale acestei organizari. Pentru a putea apela la serviciile unui obiect un client trebuie sa cunoasca explicit interfata respectivului obiect. Daca o schimbare de interfata este necesara în vederea satisfacerii unor noi cerinte functionale ale sistemului, atunci trebuie analizat impactul respectivei modificari asupra tuturor clientilor obiectului schimbat [30].

Totusi, de la promisiunea de a oferi aceste proprietati pozitive si pâna la obtinerea lor efectiva exista o distanta uriasa. Asa cum am aratat la început, simpla aplicare a organizarii orientate pe obiecte nu conduce de la sine la cresterea flexibilitatii, extensibilitatii si întelegerii programelor. În aceasta situatie se pune întrebarea: cum anume trebuie sa procedam pentru a atinge aceste deziderate?

2.4 Dezideratele proiectari obiectuale

În sectiunea 2.2 am prezentat principalele mecanisme implicate în proiectarea orientata pe obiecte. Dupa cum arata R. Martin însa, cunoscând piesele de sah si regulile jocului nu înseamna ca si jucam bine. În aceasta sectiune vom arata ce însemna o proiectare obiectuala buna si care sunt diferentele dintre bine si rau la nivelul proiectarii. O proiectare corecta poate fi definita astfel [14][1]:

> O buna proiectare este aceea care balanseaza echitabil constrângerile minimizând costul total al sistemului de pe durata sale întregi vieti.

Prin cost total se întelege suma costurilor asociate realizarii design-ului, a transformarii lui într-o implementare corespunzatoare, a operatiilor de testare si depanare precum si a activitatii de mentenanta pentru sistemul respectiv. Cea mai importanta componenta a acestei sume este reprezentata de costul mentenantei. În concluzie "cea mai importanta caracteristica a unei proiectari bune este aceea ca ea conduce la o implementare usor de întretinut".

Pfleeger[2] ofera o alta caracterizare a unei proiectari orientate pe obiecte corecte:

[1] Definitia este preluata din: P.Coad si E.Yourdon. Object-Oriented Design. Prentice Hall, London, 2nd Edition, 1991.

[2] Discutia este preluata din: S.L. Pfleeger. Software Engineering – Theory and Practice. Prentice-Hall, NJ, 1998.

O proiectare de calitate trebuie sa prezinte acele caracteristici care conduc la produse de calitate: usor de înteles, implementat, testat si modificat precum si o corecta interpretare a cerintelor. Proprietatea de modificare usoara este foarte importanta pentru ca schimbarea cerintelor sau schimbarile necesare corectarii unei erori impun uneori realizarea unor modificari de design.

Nu este posibil sa stabilim un set general de reguli care odata aplicate sa ne conduca automat la un design de calitate. În cele ce urmeaza vom prezenta cele mai importante proprietati care caracterizeaza o proiectare orientata pe obiecte corecta [14].

2.4.1 Cuplaj redus

Într-un design orientat pe obiecte, cuplajul reprezinta gradul de interconectare dintre entitatile sale. Cuplajul este un criteriu important de evaluare a design-ului deoarece el caracterizeaza o proprietate mult dorita: o modificare într-o parte a sistemului trebuie sa aiba un impact minim asupra celorlalte parti. În acelasi timp întelegerea unui modul trebuie sa necesite întelegerea unui numar cât mai mic de alte module. Un cuplaj excesiv între entitatile de design afecteaza în mod negativ diverse proprietati ale sistemului proiectat.

- **Reutilizabilitatea.** Reutilizabilitatea claselor si subsistemelor este redusa când acestea sunt puternic cuplate, datorita dependentei entitatii (clasa ori subsistem) de contextul în care este folosita. Aceasta dependenta face greu posibila reutilizarea entitatii într-un alt context.

- **Modularitatea.** În mod normal, un modul ori subsistem trebuie sa fie slab cuplat cu restul modulelor din sistem. Un cuplaj puternic între module afecteaza în mod negativ proprietatea de modularitate, indicând faptul ca nu sunt clar specificate responsabilitatile fiecarui modul din sistem.

- **Întelegerea si testabilitatea.** O autonomie scazuta a claselor afecteaza în mod negativ operatia de întelegere a sistemului. Când fuxul de control al unei clase depinde de un numar mare de alte clase este mult mai greu de urmarit logica clasei respective pentru ca întelegerea ei implica necesitatea întelegerii tuturor partilor de functionalitate externa pe care se bazeaza clasa noastra.

Principial exista doua categorii de cuplaj: cuplaj între obiecte aparut în urma invocarii de metode (cuplaj de interactiune) si cuplaj între clase datorat relatiilor de mostenire existente între ele (cuplaj de mostenire).

Prin cuplaj redus s-ar putea întelege necesitatea eliminarii sale în totalitate. Evident, acest lucru nu este posibil. Obiectele trebuie sa colaboreze între în vederea îndeplinirii cerintelor de functionalitate ale sistemului. Reducerea cuplajului trebuie înteleasa ca o operatie de eliminare a cuplajului ce nu este necesar.

2.4.2 Coeziune ridicata

Coeziunea este un aspect complementar cuplajului. Ea descrie gradul cu care entitatile unei portiuni de design contribuie la îndeplinirea unui singur si bine definit scop. Coeziunea ridicata la nivel de clasa indica o strânsa legatura între toate elementele respectivei clase. Dupa cum se poate observa o proiectare obiectuala buna trebuie sa balanseze echitabil constrângerile de coeziune ridicata si cele de cuplaj redus. Lipsa de coeziune afecteaza în mod negativ diverse proprietati ale sistemului proiectat.

- **Reutilizabilitatea.** Lipsa de coeziune se datoreaza în special încorporarii într-o singura clasa a mai multor functionalitati. Aceste clase sunt greu de reutilizat în alte contexte pentru ca anumite functionalitati ale sale nu sunt necesare în noul context.

- **Întelegerea.** Coeziunea scazuta la nivel de clasa datorata lipsei de concentrare a acesteia asupra unui singur scop cauzeaza un proces greu de întelegere a respectivei clase. Aceste clase includ un numar mare de metode care nu sunt corelate semantic. Este greu sa grupam metodele corelate pentru a întelege serviciile furnizate de o clasa necoeziva.

- **Modularitatea.** La nivel de modul, o coeziune scazuta arata ca sistemul nu este divizat corespunzator în subsisteme.

Coeziunea ridicata nu trebuie sa caracterizeze doar modulele si clasele. Criteriul se adreseaza atât metodelor cât si relatiilor de mostenire. O subclasa nu îsi are locul într-o anumita ierarhie de clase daca din punct de vedere semantic ea nu reprezinta o specializare a clasei pe care o extinde.

2.4.3 Complexitate gestionabila

Sistemele software devin pe zi ce trece tot mai complexe. Cresterea dimensiunii si a complexitatii afecteaza drastic multe proprietati ale sistemului proiectat.

- **Întelegerea.** Clasele mari si complexe sunt greu de înteles de catre oameni în special daca ele sunt si necoezive.

- **Mentenabilitatea.** Datorita greutatilor întâmpinate în întelegerea codului unei clase aceasta este greu de întretinut. Orice modificare adusa unei clase impune întelegerea ei în totalitate.

- **Siguranta si testabilitatea.** O clasa prea complexa nu numai ca nu este usor de întretinut si înteles dar este si greu de testat. Este deci mult mai probabila aparitia unor erori logice ceea ce are drept consecinta o scadere a sigurantei (reliability) în functionare.

Criteriul de complexitate redusa impune limitarea în dimensiune a metodelor claselor si pachetelor si simplificarea pe cât posibil a fluxului de control. Este foarte

benefica limitarea numarului de atribute si metode dintr-o clasa, a adâncimii ierarhiilor si a dimensiunii interfetelor.

2.4.4 Abstractizare corespunzatoare

Conceptul de abstractizare a datelor sta la baza organizarii orientata pe obiecte. O operatie de abstractizare corecta poate fi descrisa astfel: ”o anumita clasa trebuie sa reprezinte o singura si bine definita abstractiune si sa nu fie doar o adunatura de metode si definitii de variabile”[3]. Abstractizarea corecta influenteaza pozitiv proprietatea de întelegere usoara a unui sistem.

O clasa care reprezinta o abstractiune necorespunzatoare încearca probabil sa captureze aspectele corespunzatoare mai multor abstractiuni distincte sau încearca sa surprinda o abstractiune nerelevanta. Daca o clasa este prea complexa este foarte probabil ca ea sa încerce sa capteze mai multe abstractiuni. O astfel de clasa nu numai ca va fi excesiv de complexa dar va fi cu siguranta si necoeziva.

Abstractizarea necorespunzatoare poate fi si rezultatul utilizarii incorecte a mecanismului de mostenire. Ierarhiile de clase ar trebui sa fie adânci si înguste. Partea superioara trebuie sa fie abstracta iar subclasele trebuie sa fie specializari ale claselor extinse de acestea. Trebuie evitata utilizarea mostenirii ca mecanism de reutilizare a codului.

2.5 Principii si tipare de proiectare

Dupa cum explica R. Martin în [19] cunoasterea algoritmilor a structurilor de date si a limbajelor de programare împreuna cu mecanismele puse la dispozitie de acestea face posibila scrierea de programe de catre o persoana oarecare. Totusi, aceste programe nu vor fi bune din punctul de vedere al proiectarii lor. Cu timpul însa, persoana îsi da seama de importanta coeziunii, cuplajului si principiului de ascundere a informatiei. Se spune ca persoana a învatat principiile proiectarii orientate pe obiecte. Pentru a deveni însa un maestru un proiectant trebuie sa studieze design-urile realizate de alti maestri. În interiorul lor pot fi identificate tipare de proiectare care pot fi utilizate si în alte activitati de proiectare. Aceste tipare trebuie întelese, memorate si aplicate în mod repetat pâna devin o a doua natura.

Cunoasterea conceptelor fundamentale nu asigura obtinerea unui cuplaj redus, a coeziunii ridicate, a complexitatii reduse si a unor abstractiuni corespunzatoare. Din experienta acumulata de dezvoltatorii de software care au încercat sa obtina aceste deziderate, si-au facut aparitia o serie de euristici, reguli si principii de proiectare. Încalcarea lor conduce la aparitia carentelor de proiectare despre care vom discuta în capitolul urmator. Scopul acestui paragraf este de a prezenta câteva dintre principiile de proiectare importante si de a arata modul în care tiparele de proiectare pot ajuta în rezolvarea anumitor probleme de design.

2.5.1 Proiectarea dirijata de schimbare

Într-un articol [25] publicat în ani ’70, Parnas compara din perspectiva schimbarii doua modalitati de descompunere a sistemelor software în module. În cadrul primei variante s-a utilizat drept criteriu de descompunere transformarea

[3] R.E. Johnson, B. Foote. Designing reusable classes. Journal of Object-Oriented Programming, 1(2):22-35, June 1988.

fiecarui pas major de procesare într-un modul. S-ar putea spune ca aceasta descompunere se bazeaza pe diagrama de secventa a sistemului. Cea de-a doua varianta utilizeaza drept criteriu de descompunere conceptul de ascundere a informatiei. Modulele nu mai corespund pasilor majori de procesare. Fiecare modul este caracterizat de cunostintele legate de o decizie de proiectare pe care le ascunde fata de celelalte module. Cu alte cuvinte, se porneste de la o lista de decizii de proiectare dificile ori foarte probabil a fi schimbate (ex. modul în care se reprezinta o linie de text în memorie). Fiecare modul este astfel proiectat încât sa ascunda asemenea decizi fata de alte module. Mai exact interfata modulului este astfel definita încât sa ofere cât mai putine informatii despre modul în care se desfasoara activitatile în interiorul sau.

Parnas constata superioritatea celei de-a doua modalitati de descompunere în special din punctul de vedere al schimbarii. O schimbare asupra deciziilor de proiectare, despre care se stia de la bun început ca sunt foarte probabil a fi modificate, afecteaza în prima varianta de descompunere aproape toate modulele sistemului. Nu acelasi lucru se întâmpla în cazul celei de-a doua modalitati de descompunere. Modificarile ce trebuiesc realizate sunt concentrate la nivelul unui numar foarte mic de module. Este evident faptul ca în cel de-al doilea caz flexibilitatea sistemului este mai mare. În acelasi timp un modul este mai usor de înteles necesitând întelegerea unui numar restrâns de alte module. În cazul primei variante de descompunere sistemul nu poate fi înteles decât ca un întreg.

În contextul lucrarii [3] în care se atrage atentia asupra necesitatii de continua schimbare ce planeaza asupra software-ului, observatiile lui Parnas sunt extrem de importante. Necesitatea de schimbare sta la baza fenomenului de îmbatrânire a software-ului descris tot de Parnas în [26] si despre care vom mai vorbi în capitolul urmator. La fel ca si în cazul îmbatrânirii oamenilor, principala arma pe care o avem în lupta cu acest fenomen este întârzierea si limitarea lui. Sloganul "design for change" surprinde în mod elegant aceasta lupta. Principiul de proiectare care trebuie utilizat întotdeauna este cel descris în prima parte a acestei sectiuni. Se porneste prin identificarea schimbarilor cele mai probabile de a aparea pe durata vietii produsului software. Evident, nu vor putea fi identificate toate aceste schimbari, dar vor putea fi detectate clase de posibile schimbari: schimbari la nivelul interfetei cu utilizatorul, migrarea pe o alta platforma, etc. În continuare sistemul este astfel organizat încât elementele cel mai probabil a fi modificate sunt "concentrate" într-o cantitate de cod mica astfel încât modificarea lor sa afecteze doar o mica parte a sistemului.

Încalcarea acestui principiu are în timp repercusiuni catastrofale: sistemul este pur si simplu imposibil de întretinut pentru ca este imposibil de înteles. Toate entitatile de design par total necoezive, total cuplate. Exista multe cauze ale încalcarii paradigmei "design for change" dar poate cea mai importanta este aceea ca design-ul obtinut în urma utilizarii acestei abordari difera mult de design-ul "natural" vazut intuitiv de programatori. Intuitia îi spune acestuia sa gândeasca în termeni de pasi de executie si nu în termeni de schimbari.

2.5.2 Principii de proiectare

Într-o suita de articole [16,17,18,20,21,22] R. Martin sintetizeaza un numar de principii de proiectare orientata pe obiecte. Principiile sunt organizate pe doua niveluri de abstractizare: principii la nivelul proiectarii claselor si principii la nivelul arhitecturii pachetelor. În aceasta sectiune ne propunem sa trecem în revista câteva dintre aceste principii insistând doar asupra celor mai importante aspecte.

Open-Closed Principle

> Entitatile software (clase, module, functii, etc.) trebuie sa fie deschise pentru extindere dar închise pentru modificari.

Prin deschiderea unui modul înspre extindere întelegem ca activitatile sau mai exact comportamentul sau poate fi extins. Putem sa facem ca respectivul modul sa se comporte altfel pe masura ce cerintele aplicatiei se schimba. Prin închiderea modulului înspre modificari întelegem inviolabilitatea codului sursa asociat lui. Nimeni nu are dreptul sa modifice codul sursa.

Abstractizarea corespunzatoare este cheia ce deschide drumul pentru obtinerea acestui deziderat. Putem crea abstractiuni care sa fie fixe, dar care sa prezinte un grup nelimitat de comportamente. Abstractiunile vor fi implementate sub forma de clase abstracte a caror scop principal este de a defini o interfata comuna pentru toate subclasele sale. Grupul nelimitat de comportamente asociat unei abstractiuni este reprezentat de totalitatea subclaselor ce extind clasa abstracta. Este deci posibil ca un modul sa captureze aspectele legate de o anumita abstractiune si sa fie închis modificarilor din moment ce el depinde de o abstractiune care este fixa. Comportamentul modulului poate fi extins prin noi derivari ale abstractiunii.

Multe euristici de proiectare sunt derivate din acest principiu. "Make all member variables private" sau "No global variables" nu sugereaza altceva decât riscul de a deschide un modul înspre modificari. Utilizând variabile globale sau câmpuri publice riscam sa fim nevoiti sa efectuam multe modificari în locuri total neasteptate. Acelasi lucru este exprimat si de euristica "Run time type identification is ugly". Un modul care identifica tipul efectiv al unei referinte catre o clasa de baza dintr-o ierarhie, se va modifica ori de câte ori apare o noua subclasa în respectiva ierarhie. Totusi, operatiile de identificare a tipului efectiv nu sunt întotdeauna daunatoare.

Liskov Substitution Principle

> Functiile care utilizeaza referinte catre o clasa de baza dintr-o ierarhie trebuie sa fie capabila sa utilizeze instante (obiecte) ale oricarei subclase corespunzatoare clasei de baza fara sa stie efectiv acest lucru.

O functie care încalca acest principiu detine informatii despre toate subclasele clasei la care functia considerata detine o referinta. De fiecare data când o noua subclasa este creata functia trebuie modificata. Apare imediat o încalcare a lui open-closed principle. Importanta fundamentala a acestui principiu consta în faptul ca ofera o modalitate simpla de verificare a corectitudinii unei ierarhii de clase. Relatia de tip "is a" existenta între o subclasa si superclasa sa trebuie sa se refere la comportamentul extern, cel care este perceput de un client. Nu sunt importante aspectele de comportament intrinseci ci cele extrinseci.

O euristica care adreseaza aceste probleme spune ca nu este permis ca o subclasa sa suprascrie o metoda a superclasei sale, fara ca aceasta noua metoda sa faca ceva concret (metode de tip NOP). Este evident ca o astfel de metoda introduce falsitati în modul în care un client percepe comportamentul unui obiect daca acesta nu cunoaste tipul efectiv al obiectului.

Dependency Inversion Principle

> Abstractiunile nu trebuie sa depinda de detalii. Detaliile trebuie sa depinda de abstractiuni.
>
> Modulele de nivel înalt nu trebuie sa depinda de module de nivel coborât. Ambele trebuie sa depinda de abstractiuni.

Acest principiu este cel care face diferenta între programarea si proiectarea cu caracter procedural si programarea si proiectarea orientata pe obiecte. Metodele traditionale de dezvoltare software ca analiza si proiectarea structurata tind sa creeze structuri software în care modulele de nivel înalt depind de module de nivel inferior si în care abstractiunile depind de detalii. Scopul acestor metode este de a defini o ierarhie de subprograme care descrie modul în care modulele de nivel înalt apeleaza serviciile puse la dispozitie de modulele de nivel inferior. O astfel de structura este intrinsec slaba. Modulele de nivel înalt trateaza politicile de nivel înalt ale aplicatiei, captureaza modelul decizional al domeniului în care functioneaza sistemul software. În general aceste politici nu sunt interesate de detaliile care le implementeaza. Totusi, ele depind de modulele de nivel inferior si în consecinta o schimbare la nivelul acestor module poate sa le afecteze în mod direct facând necesara modificarea lor. În acelasi timp, modulele de nivel înalt nu vor putea fi reutilizate în alte contexte. Întrebarea care se pune în mod natural este: de ce trebuie sa depinda aceste politici de modulele de nivel inferior daca ele nu sunt interesate de detaliile de implementare?

Este clar ca lucrurile ar trebui sa fie exact invers. Modulele de nivel înalt ar trebui sa forteze o modificare în modulele de nivel inferior. Mai mult, modulele de nivel înalt care captureaza politica aplicatiei trebuie sa fie usor de reutilizat. Aceasta inversiune a dependentei reprezinta esenta acestui principiu. Într-o arhitectura orientata pe obiecte dependintele trebuie sa fie dirijate înspre abstractiuni. Cu alte cuvinte, modulele de nivel înalt nu mai depind de modulele ce contin detalii de implementare ci ambele trebuie sa depinda de abstractiuni.

În concluzie orice entitate de design trebuie sa depinda de interfete sau clase abstracte, nu de clase concrete. Motivul este ca lucrurile abstracte se modifica mai rar decât lucrurile concrete. Mai mult, abstractiunile reprezinta puncte în design unde acesta poate fi extins (Open-Closed Principle). Acest principiu sta la baza dezvoltarii framework-urilor.

Interface Segregation Principle

> Clientii nu trebuie sa fie fortati sa depinda de interfete pe care ei nu le utilizeaza.

O clasa poate avea mai multi clienti interesati de servicii diferite furnizate de ea. Daca o singura interfata înglobeaza toate aceste servicii atunci vor exista clienti care depind de parti din respectiva interfata neinteresante din punctul lor de vedere. Desi nu pare o problema majora, o schimbare într-o parte a interfetei de care este interesata doar o anumita categorie de clienti va putea afecta toti clienti indiferent de care parte din interfata sunt ei interesati. În cel mai fericit caz, toti clientii vor trebui recompilati. Aceasta operatie este însa consumatoare de timp în cazul sistemelor mari. Întrebarea care se pune însa este: de ce sa fiu obligat sa recompilez clienti care nici macar nu sunt interesati de serviciul care a fost schimbat? Este cel putin ciudat sa

trebuiasca sa recompilezi parti din sistem care nu au nimic în comun cu partea modificata de cineva. Aceasta situatie apare pentru ca s-a încalcat principiul segregarii interfetei.

O metoda corecta de tratare a situatiei este sa distribuim metodele interfetei initiale în mai multe interfete specifice fiecarei categorii de clienti. Aceste interfete vor putea fi ulterior implementate în cadrul aceleiasi clase. Ca urmare, daca trebuie sa modificam interfata de care depinde o anumita categorie de clienti numai acestia vor fi afectati.

La finalul discutiei despre principiile de proiectare nu putem sa nu facem urmatoarea observatie: toate principiile de proiectare prezentate (si nu numai) rezolva anumite probleme care apar în contextul efectuarii de schimbari la nivelul sistemului software. Acest lucru demonstreaza veridicitatea afirmatiilor lui F.P.Brooks Jr. din [3] referitoare la necesitatea de continua schimbare ce planeaza asupra sistemelor software. În acelasi timp, se arata cât de importanta este afirmatia lui Parnas din [26] conform careia software-ul trebuie proiectat având în vedere posibilele schimbari ce vor fi necesare pe durata de viata a programului.

2.5.3 Tipare de proiectare

În timp, daca proiectam arhitecturi orientate obiect având grija sa respectam principiile mai sus amintite (si cele care mai exista), vom observa ca anumite structuri de design se repeta iar si iar. Aceste structuri repetitive se numesc tipare de proiectare.

Definitie. Un tipar de proiectare descrie o problema de proiectare repetata ce apare într-un context specific de proiectare si prezinta o schema generica dovedita utila pentru solutionarea ei. Schema solutiei descrie componentele constituente, responsabilitatile lor, relatiile dintre ele si modul în care acestea colaboreaza [4].

Nu ne propunem în aceasta sectiune sa purtam o discutie fundamentala despre tiparele de proiectare. Ne vom limita doar la discutarea anumitor proprietati ale acestora si a modului în care ele rezolva o anumita problema de proiectare particulara.

Tiparele de proiectare prezinta urmatoarele proprietati [4]:

1. Un tipar adreseaza o problema de proiectare ce apare în mod repetat în anumite situatii prezentând o modalitate de solutionare a ei. Spre exemplu, un efect colateral al partitionarii unui sistem într-un set de obiecte cooperante este necesitatea de a mentine consistenta datelor între anumite obiecte. Evident, nu vrem sa mentinem aceasta consistenta printr-un cuplaj strâns între clase. Motivele au fost mentionate în sectiunile anterioare. Tiparul Observer descrie modul în care aceasta problema poate fi rezolvata. Obiectele implicate în acest tipar sunt de doua categorii: subiect si observator. Când un obiect observator trebuie sa fie sincronizat cu starea unui alt obiect subiect, el se înregistreaza la acesta. Când subiectul îsi schimba starea tot el anunta toti observatorii înregistrati la el. Ca raspuns, fiecare observator va interoga subiectul în vederea sincronizarii starii sale cu starea subiectului.

2. Tiparele documenteaza o solutie de proiectare existenta si dovedita utila. Ele nu sunt inventate sau create artificial. Ele captureaza si furnizeaza o modalitate de reutilizare a cunostintelor de design rezultate din experienta altor proiectanti.

3. Tiparele identifica si specifica abstractiuni la un nivel superior unei singure clase, instante ori componenta. Tiparele descriu un numar de componente, clase ori obiecte, detaliaza responsabilitatile fiecareia si relatiile dintre ele precum si modul în care ele coopereaza. Toate componentele rezolva împreuna problema adresata de un tipar.

4. Tiparele furnizeaza un vocabular adecvat si o modalitate eleganta de întelegere a principiilor de proiectare. Numele tiparelor devin de cele mai multe ori parte constituenta a limbajului proiectantilor facilitând astfel discutiile referitoare la diverse probleme de proiectare si la solutiile adecvate lor. Nu mai este necesar sa se descrie în totalitate solutia referitoare la o anumita problema particulara. De cele mai multe ori aceasta descriere este lunga si complicata. În locul ei se specifica doar numele tiparului utilizat si se arata corespondenta dintre o anumita parte a solutiei si o anumita componenta a tiparului.

5. Tiparele de proiectare sunt utile în documentarea arhitecturii unui sistem software. Ele sunt capabile sa descrie viziunea proiectantului asupra design-ului. Acest lucru îi va ajuta pe altii sa nu încalce viziunea initiala când extind sau modifica structura originala sau codul sistemului.

6. Tiparele ajuta la construirea unui sistem caracterizat de anumite proprietati necesare. Ele furnizeaza un schelet comportamental si ajuta la implementarea functionalitatii aplicatiei. Mai mult, ele adreseaza si anumite cerinte nefunctionale legate de sistem: flexibilitate, siguranta în functionare, reutilizabilitate, etc.

7. Tiparele ajuta la construirea unor arhitecturi complexe si eterogene. Fiecare tipar defineste un set de componente, roluri si relatii între ele. Ele pot fi utilizate pentru specificarea anumitor aspecte particulare ale unei structuri software concrete. Tiparele actioneaza ca blocuri "prefabricate" pentru construirea unui design mai complex. Acest lucru afecteaza pozitiv viteza de executie a activitatii de proiectare si calitatea finala a acesteia. Întelegerea si corecta aplicare a unui tipar elimina timpul necesar gasirii unei solutii originale.

8. Un tipar furnizeaza structura de baza a solutiei corespunzatoare unei anumite probleme de proiectare si nu solutia integrala. Cu alte cuvinte, se furnizeaza o schema corespunzatoare unei solutii generice pentru o familie de probleme si nu un modul prefabricat care poate fi utilizat asa cum este el. Aceasta schema trebuie implementata tinând cont de particularitatile contextuale ale problemei.

9. Tiparele ajuta la gestiunea complexitatii software. Fiecare tipar descrie un mod adecvat de abordare a unei probleme: componentele necesare, rolurile lor, detaliile ce trebuie ascunse, abstractiunile ce trebuie sa fie vizibile si modul de functionare de ansamblu. În contextul unei anumite probleme rezolvabile prin utilizarea unui tipar nu este necesar sa pierdem timpul pentru realizarea unei noi solutii. Daca tiparul este aplicat corect putem avea încredere în solutia furnizata de el.

O proprietate importanta a tiparelor de proiectare este aceea ca ele adreseaza probleme de proiectare întâlnite la diverse nivele de abstractizare a unui sistem. Anumite tipare ajuta la structurarea sistemului în subsisteme, altele la rafinarea subsistemelor, iar altele la implementarea anumitor aspecte particulare ale proiectarii într-un anumit limbaj de programare. În concluzie ele pot fi [4]:

- **Tipare arhitecturale.** Un tipar arhitectural exprima o schema de organizare structurala fundamentala a unui sistem. El prezinta un set predefinit de subsisteme, specifica responsabilitatile fiecaruia si include reguli si linii directoare pentru organizarea relatiilor dintre ele. Un astfel de tipar este Model-View-Controller.

- **Tipare de proiectare propriu-zise.** Ele furnizeaza o schema de rafinare a subsistemelor ori componentelor sistemelor software sau a relatiilor dintre ele. Un tipar descrie o structura de componente intercomunicante des întâlnita care rezolva o problema generala de design într-un anumit context. Un exemplu de astfel de tipar este Observer.

- **Idiom.** Un idiom este un tipar de nivel coborât specific unui anumit limbaj de programare. El descrie cum sa implementam anumite aspecte particulare ale componentelor sau a relatiilor dintre ele utilizând facilitatile unui anumit limbaj de programare.

În finalul acestei sectiuni vom arata cum un tipar de proiectare rezolva o anumita problema concreta. Crearea obiectelor este una dintre operatiile care depinde nemijlocit de specificarea unei anumite clase concrete. Prin definitie nu se pot instantia clase abstracte. Deci, pentru crearea unei instante trebuie sa depindem de o clasa concreta si nu de una abstracta. Crearea obiectelor este o operatie care se utilizeaza des si în principiu în toate partile unui sistem. Grav! S-ar parea ca principiul dependentelor inverse nu poate fi satisfacut în ceea ce priveste crearea obiectelor. Totusi, aceasta problema este rezolvata elegant de tiparele de proiectare creationale.

Ne vom limita doar la descrierea modului în care tiparul de proiectare cunoscut sub numele de AbstractFactory rezolva problema mai sus amintita [8]. Acest tipar permite ca dependentele fata de clasele concrete instantiate sa existe într-un loc si doar într-un sigur loc.

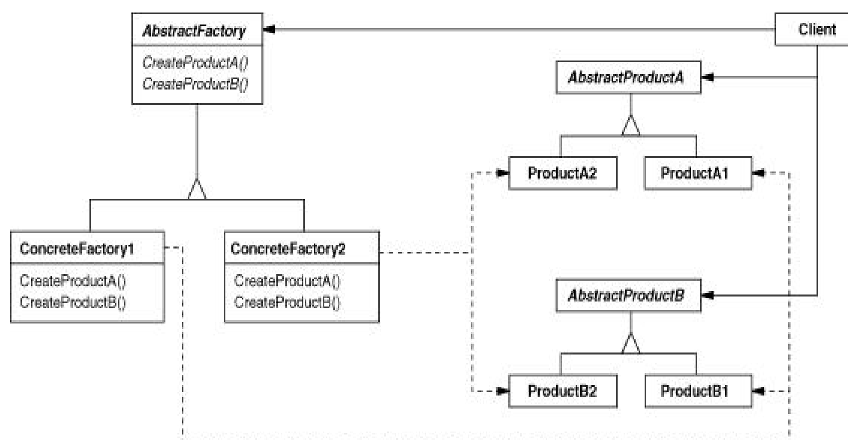

Figura 2.2. Structura tiparului de proiectare AbstractFactory

Principalele clase implicate sunt:

- AbstractFactory. Declara o interfata ce contine operatiile de creare a obiectelor de tip AbstractProduct.
- ConcreteFactory. Implementeaza operatiile de instantiere a claselor concrete corespunzatoare produselor.
- AbstractProduct. Declara o interfata pentru un anumit tip de produs.
- ConcreteProduct. Defineste un obiect produs si implementeaza interfata AbstractProduct.
- Client. Utilizeaza doar interfetele declarate de AbstractFactory si AbstractProduct.

Ideea din spatele acestei structuri consta în faptul ca doar o singura instanta a unei clase ConcreteFactory este necesara. Vom avea deci o singura dependenta înspre o clasa concreta în afara fabricii de obiecte. Avantajul acestei organizari este acela ca fabrica încapsuleaza responsabilitatea si procesul de creare a obiectelor, izolând clientul de clasele concrete ce definesc un anumit produs. Clientii manipuleaza obiectele produs prin interfetele lor abstracte. Clasele concrete ce definesc produsele sunt izolate în implementarea fabricii concrete, numele lor neaparând în codul clientului. Exista însa si dezavantaje: daca apar produse de tip nou interfata AbstractFactory trebuie extinsa pentru a permite crearea de obiecte asociate noului tip de produs. În consecinta trebuie modificate si toate subclasele sale. Exista modalitati de evitare a acestei situatii dar cresterea de flexibilitate conduce la o scadere în ceea ce priveste siguranta.

Arta utilizarii tiparelor de proiectare consta atât în identificarea necesitatii folosirii unui anumit tipar de proiectare într-o anumita situatie concreta, cât si în implementarea lui propriu-zisa într-o asemenea maniera încât cerintele importante din perspectiva situatiei respective sa fie îndeplinite.

Capitolul III

Detectia carentelor de proiectare în sistemele software organizate orientat pe obiecte

3.1 Carente de proiectare

În ultimul timp se pune tot mai mult accent pe descrierea erorilor sau carentelor în design-ul sistemelor software orientate pe obiecte. Acest lucru se datoreaza puternicului impact negativ pe care acestea le au asupra unor atribute de calitate cum ar fi flexibilitatea si mentenabilitatea. Prin urmare, identificare si detectia acestor probleme de design este esentiala pentru evaluarea si îmbunatatirea calitatii software-ului.

În cadrul acestei sectiuni vom defini carentele de proiectare, vom identifica principalele cauze ale aparitiei lor, vom prezenta principalele simptoame ale prezentei lor în cadrul sistemului software si vom clasifica principalele tipuri de carente.

3.1.1 Definire

Vom prelua definitia carentelor de proiectare asa cum apare în [14]:

Definitie. Totalitatea caracteristicilor structurale apartinând unei entitati de design sau unui fragment de design care surprind o îndepartare de la un set dat de criterii ce caracterizeaza un design de calitate, se numeste carenta de proiectare.

3.1.2 Cauzele aparitiei carentelor de proiectare

Cauzele carentelor de proiectare pot fi clasificate în doua mari categorii dupa cum urmeaza:

- directe
- indirecte

Cauzele directe sunt reprezentate de elementele ce genereaza în mod nemijlocit aparitia carentelor de proiectare. În aceasta categorie intra încalcarea principiilor de proiectare specifice orientarii pe obiecte. Totusi, nu putem sa spune ca proiectatii sistemului au încalcat în mod voit aceste principii. Cine ar dori sa-si îngreuneze munca? La baza încalcarii lor trebuie sa se gaseasca cauze de alta natura, independente de conceptele de proiectare.

Cea de-a doua categorie este reprezentata de acele cauze care nu tin în mod direct de organizarea orientata pe obiecte, dar care induc aparitia unor cauze directe. Asa cum se aminteste si în [10], imaturitatea designer-ilor programelor care nu înteleg importanta respectarii principiilor de organizare, este probabil cea mai grava cauza indirecta a aparitie carentelor de proiectare. Exista însa si alte cauze indirecte, a caror aparitie se datoreaza unor fenomene complexe ce caracterizeaza în principiu orice produs software.

Astfel, spre deosebire de alte produse materiale, software-ul se confrunta în mod frecvent cu cerinte de modificare venite din partea utilizatorilor. Aceasta caracteristica este una de esenta ce izvoraste din natura software-ului [3]. Pe de-o parte, programul ce functioneaza în cadrul unui sistem mai mare înglobeaza functiile acestuia, iar aceste functii sunt de cele mai multe ori subiectul modificarii. Pe de alta parte, un program poate fi modificat mult mai usor fiind o entitate pur logica, infinit maleabila.

În consecinta, este foarte probabil ca la un moment dat cerintele sa se modifice într-un mod ce nu a fost anticipat la realizarea design-ului initial. O rezolvarea corecta a situatiei ar necesita realizarea unor modificari la nivelul arhitecturii aplicatiei. Problema este însa ca de cele mai multe ori, inginerii care se ocupa de mentenanta nu sunt cei care au dezvoltat sistemul. Ei nu înteleg în totalitate "filosofia" din spatele design-ului si în consecinta modificarile realizate de ei vor viola conceptele acestuia. Mai mult, modificarile trebuie facute rapid si prin urmare se va alege cea mai rapida solutie si nu cea care pastreaza integritatea conceptuala a design-ului. Atât cererile de modificare repetate cât si constrângerile de timp reprezinta cauze indirecte ale aparitiei carentelor de proiectare.

O cauza mult mai subtila deriva si din faptul ca ingineria software nu este în totalitate o inginerie în adevaratul sens al cuvântului. În toate domeniile ingineresti se face distinctie între activitatea de proiectare cu caracter de rutina si cea cu caracter inovator [29]. Prima implica rezolvarea unor probleme familiare, reutilizând complet ori partial solutii anterioare. Proiectarea inovatoare implica gasirea unor solutii noi la probleme nefamiliare. Scopul unei discipline ingineresti este de a face proiectarea de rutina mai simpla, capturând cunostinte legate de activitatea de proiectare spre a fi utilizate în alte situatii asemanatoare. În proiectarea software nu avem instrumente care sa poata captura, organiza si transmite astfel de cunostinte. Nu avem proceduri clare care sa ne permita sa alegem cea mai buna varianta de proiectare dintre mai multe alternative posibile într-un anumit context. Ne bazam mai mult pe proiectarea inovativa, reinventând de multe ori "roata". Faptul ca nu avem cum sa aflam ca altcineva a inventat-o deja si ca a facut-o chiar mult mai bine decât noi, determina o probabilitate mult mai mare ca în timp, pe parcursul evolutiei aplicatiei, "roata" noastra sa înceapa sa-si prezinte carentele de proiectare.

3.1.3 Simptoamele carentelor de proiectare

Exista patru simptoame primare care ne atrag atentia ca design-ul nostru este afectat de anumite carente de proiectare [15]. Acestea sunt: rigiditatea, fragilitatea, imobilitatea si vâscozitatea.

A. **Rigiditatea.** Rigiditatea reprezinta tendinta unui program de a fi greu de modificat, chiar si în situatia în care aceste modificari sunt simple. Orice modificare, oricât de mica ar fi ea, cauzeaza o cascada de alte modificari care trebuie realizate în modulele dependente. Acest fel de comportare face imposibila estimarea cu certitudine a timpului necesar realizarii unei modificari.

B. **Fragilitatea.** Fragilitatea reprezinta tendinta unui program de a se comporta într-o maniera anormala în diferite portiuni ale sale, de fiecare data când este modificat. De multe ori aceasta comportare anormala apare în portiuni care nu se gasesc într-o relatie conceptuala cu portiunea

modificata. Cu cât sistemul e mai fragil, probabilitatea de comportare anormala creste în timp, facând imposibil procesul de mentenanta. Asta pentru ca nu se poate estima cu precizie impactul pe care o modificare îl are asupra întregului sistem.

C. **Imobilitatea.** Imobilitatea reprezinta imposibilitatea de a reutiliza unitati de program din alte proiecte sau din acelasi proiect. Nu de putine ori ni se întâmpla sa constatam ca avem nevoie de un modul foarte asemanator cu cel scris de altcineva. Totusi, ne vom da seama dupa un timp ca modulul în cauza are prea mult "bagaj" de care depinde, "bagaj" ce pentru noi nu este interesant, iar munca de separare este prea grea si riscanta. De aceea vom rescrie o buna parte a modulului respectiv în loc sa apelam la reutilizare.

D. **Vâscozitatea.** Acest simptom s-ar putea traduce astfel: este usor sa aplicam o solutie proasta, dar este greu sa o aplicam pe cea buna. Vâscozitatea are doua forme. Prima tine de design si se refera la situatia în care inginerii, pusi în situatia de a alege o varianta de modificare dintre mai multe alternative posibile, aleg o solutie ce nu conserva integritatea conceptuala a design-ului în detrimentul uneia ce o conserva, singurul motiv fiind acela ca prima varianta este usor de realizat, iar cea de-a doua nu este. A doua forma tine de mediul de dezvoltare si apare atunci când acesta este ineficient si neperformant din punctul de vedere al factorului timp. Daca timpul de compilare este mare, dezvoltatorul va fi tentat sa aleaga o solutie de modificare ce nu forteaza recompilari masive, chiar daca respectiva solutie încalca integritatea conceptuala a design-ului.

Aceste simptoame caracterizeaza orice sistem software orientat pe obiecte care este afectat de diferite carente de proiectare la nivelul arhitecturii sale. Problema este însa ca ele sunt prea mult orientate catre ansamblul sistemului si prea putin orientate catre entitatile sau fragmentele de design afectate de carente de proiectare. Ele nu sunt capabile sa identifice si sa localizeze cu exactitate fragmentele de design "bolnave" pentru a le putea prescrie un anumit "tratament".

M. Fowler si colaboratorii fac un pas înainte în directia localizarii fragmentului de design "bolnav". Ei definesc în [7] asa numitele "bad smells in code" care în esenta sunt carente de proiectare caracterizate de o anumita simptomatologie. Mai mult, ei prescriu pentru fiecare "bad smell" identificat un "tratament" sistematic menit sa elimine respectiva carenta de proiectare. Acest tratament consta într-un set de refactorizari care pot fi aplicate cu succes în contextul carentei respective. În continuare vom prezenta câteva dintre carentele identificate de autorii mai sus amintiti, insistând doar asupra simptoamelor respectivelor carente. Scopul este de a surprinde îndreptarea atentiei de la simptoamele orientate spre ansamblul design-ului software catre simptoamele orientate spre entitati sau fragmente de design.

- **Data Class.** Acestea sunt clasele care contin doar câmpuri si metode de tip accesor[4] pentru aceste câmpuri. În rest ele nu fac nimic altceva, fiind doar un suport pentru memorarea datelor. Mai mult ele pot avea si câmpuri publice. În consecinta ele sunt aproape sigur manipulate prea în detaliu de catre alte clase,

[4] O metoda de tip accesor seteaza ori returneaza valoarea unui câmp fara a face nimic altceva

lucru ce contravine conceptului de ascundere a informatiei ce sta la baza organizarii orientate pe obiecte.

- **Large Class.** Acestea sunt clasele care încearca sa faca mai mult decât este necesar. Într-o organizare organizata pe obiecte, inteligenta sistemului se distribuie uniform obiectelor ce compun sistemul si implicit claselor ce definesc implementarea respectivelor obiecte. Totusi, de multe ori se tinde ca aceasta inteligenta sau o mare parte din ea sa fie concentrata într-o singura clasa. Un prim simptom al unei astfel de situatii consta într-un numar prea mare de date membru continute de respectiva clasa. Un alt simptom este reprezentat de o cantitate prea mare de cod asociat clasei.

- **Feature Envy.** Un obiect încapsuleaza un set de date si un set de operatii primitive asociate datelor respective. O carenta de proiectare clasica este amplasarea gresita a unei metode, ea fiind mai interesata de o alta clasa decât de cea în care se gaseste. Interesul se îndreapta de cele mai multe ori catre date, o astfel de metoda facând uz foarte des de metode de tip accesor pentru a prelua date necesare calculului altor valori. Acesta este si simptomul principal al carentei.

- **Switch Statements.** O caracteristica a organizarii orientate pe obiecte este frecventa scazuta de utilizare a instructiunilor de tip if-else sau switch. Deseori însa regasim aceeasi instructiune switch împrastiata în diferite portiuni ale codului. Daca trebuie sa adaugam o clauza la una dintre aceste instructiuni, atunci trebuie gasite si toate celelalte instructiuni, pentru a le adauga si lor noua clauza. Acesta este un alt simptom al prezentei carentelor de proiectare.

3.1.4 Clasificarea carentelor de proiectare

Exista mai multe moduri de clasificare a carentelor de proiectare [14]. Pe de-o parte ele pot fi clasificate dupa criteriul de proiectare din care sunt derivate, iar pe de alta parte dupa nivelul de granularitate al entitatii de design afectata de respectiva carenta de proiectare. În lucrarea de fata ne vom concentra asupra celei de-a doua modalitati de clasificare.

În principiu exista trei tipuri de entitati de proiectare: metode, clase si subsisteme, fiecare corespunzând unui alt nivel de granularitate. Cel mai mic nivel corespunde metodelor, iar cel mai ridicat subsistemelor. Cum o carenta de proiectare afecteaza o entitate de design, un mod natural de clasificare a carentelor corespunde acestor trei nivele de granularitate. Ca urmare avem:

- carente de proiectare asociate metodelor
- carente de proiectare asociate claselor
- carente de proiectare asociate subsistemelor[5]

Pe lânga aceste trei niveluri în [14] se mai introduce unul: nivelul micro-design. Carentele de proiectare de la acest nivel afecteaza nu doar o clasa ci un grup de clase. Un caz tipic de astfel de carenta îl reprezinta un fragment de design unde ar

[5] În mod frecvent la acest nivel de granularitate se vorbeste despre pachete si vom avea carente de proiectare asociate pachetelor

fi trebuit aplicat un anumit tipar de proiectare, dar solutia oferita de acesta a fost ignorata. Îsi face aparitia o a patra categorie de carente de proiectare asociata tiparelor de proiectare.

3.1.5 Exemple

În încheierea sectiunii dedicate prezentarii carentelor de proiectare vom descrie câteva carente clasice identificate si discutate în [10,11,13,14]. Ne vom limita doar la exemplificarea carentelor de proiectare la nivel de clase, punctând de fiecare data abaterile de la criteriile de proiectare orientata pe obiecte si impactul pe care carenta îl are asupra caracteristicilor de calitate externe.

A. Data Classes: sunt acele clase ale caror scop este doar memorarea datelor fiind aproape sigur ca alte clase depind foarte mult de ele. Lipsa de functionalitate indica faptul ca datele si operatiile aferente lor nu sunt tinute în acelasi loc asa cum presupune organizarea orientata pe obiecte. Acesta este un simptom clar al unui proces de abstractizare deficitar. DataClass-urile afecteaza în mod negativ mentenabilitatea, testabilitatea si întelegerea sistemului.

B. God Classes. Într-un design corect orientat pe obiecte, inteligenta sistemului este uniform distribuita între clase. Aceasta carenta de proiectare caracterizeaza acele clase ce tind sa centralizeze inteligenta sistemului. O instanta de GodClass executa cea mai mare parte a activitatilor din sistem, delegând doar detalii minore unui set de clase triviale si utilizând date apartinând altor clase. Este clar ca apare o deviere de la criteriul de complexitate redusa, GodClass-ul tinzând sa capteze mai multe abstractiuni. Ca urmare, aceste clase tind sa fie necoezive. Aceasta carenta de proiectare are un impact negativ asupra caracteristicilor de reutilizare si întelegere a sistemului software.

C. Shotgun Surgery. Aceasta carenta de proiectare îsi face simtita prezenta în situatia în care la fiecare schimbare operata în interiorul unei clase apare necesitatea realizarii unor mici modificari într-un numar ridicat de alte clase. De obicei aceste schimbari sunt distribuite într-un numar ridicat de parti ale sistemului, lucru ce le face greu de localizat fiind foarte probabil sa ne scape o modificare importanta. În consecinta mentenabilitatea sistemului este grav perturbata.

3.2 Detectia carentelor de proiectare

Desi simptoamele carentelor de proiectare sunt în principiu usor de observat, cauzele lor sunt greu de gasit. În aceasta sectiune vom trece în revista diferite mecanisme utilizate în procesul de detectie a fragmentelor de design care îngradesc evolutia si reutilizarea unui sistem software orientat pe obiecte.

3.2.1 Necesitatea

Asa cum am aratat si în sectiunea anterioara, necesitatea de modificare a software-ului este o proprietate esentiala a acestuia [3]. Aceasta proprietate sta la baza

fenomenului de “îmbatrânire” ce caracterizeaza în principiu orice produs software [26].

Prin “software aging” se întelege incapacitatea software-ului de a satisface noile cerintele ale utilizatorilor. Programele batrâne sunt vazute ca si niste invalizi ce poarta povara unei mosteniri din trecut. Totusi, aceste programe au de multe ori o valoare economica foarte mare si nu se poate renunta usor la ele. În acelasi timp, datorita îmbatrânirii, ele împiedica dezvoltarea sistemelor ce includ aceste programe. Rezolvarea acestui cerc vicios se poate face doar prin prevenirea îmbatrânirii software-ului si prin aplicarea unui tratament de întinerire când este cazul.

Exista doua tipuri de îmbatrânire. Prima este cauzata de incapacitatea producatorului de a-si modifica programul în vederea satisfacerii noilor nevoi ale utilizatorului. În timp, utilizatorul devine tot mai nemultumit si va trece la achizitionarea unui software nou imediat ce costurile implicate vor permite acest lucru. Devine deci esential sa modificam programul pentru a preveni acest fel de îmbatrânire. Modificarile realizate constituie, din pacate, cauza principala a celei de-a doua forme de îmbatrânire. Aceste modificari sunt realizate în multe situatii de ingineri care nu înteleg design-ul initial cauzând în acest fel o deteriorare a structurii programului. Mai mult persoanele implicate sunt presate de timp si de cele mai multe ori nu au o pregatire adecvata în domeniu, neîntelegând vechia sintagma “design for change”. Dupa multe astfel de modificari, deteriorarea este atât de mare încât nici macar inginerii care au dezvoltat sistemul software nu-l mai înteleg. Cei ce l-au modificat nu l-au înteles niciodata. Rezultatul este ca nimeni nu mai întelege programul, acesta devenind foarte greu de modificat.

În fata fenomenului de îmbatrânire nu suntem însa neputinciosi. D.L. Parnas prezinta în [26] si modalitati de controlare a fenomenului. Astfel, putem sa prevenim îmbatrânirea, printr-o proiectare care sa tina cont de posibilele modificari viitoare. Prevenirea este utila dar ea nu elimina fenomenul. Îmbatrânirea este inevitabila datorita capacitatii noastre limitate de a prezice viitorul, de a prevedea corect modificarile ulterioare. Trebuie deci sa fim capabili sa ne confruntam cu programe îmbatrânite. Cum?

Dupa cum se poate observa, cauzele îmbatrânirii software-ului sunt similare cauzelor indirecte ce conduc la aparitia carentelor de proiectare. În consecinta, prezenta carentelor de proiectare si fenomenul de îmbatrânire a programelor se afla în strânsa legatura. Este de asteptat deci ca eliminarea unor carente de proiectare sa conduca, cel putin în parte, la întinerirea software-ului. Aceasta întinerire este rezultatul procesului cunoscut sub numele de reengineering.

Definitie. Prin reengineering se întelege examinarea unui sistem în vederea reconstituirii sale într-o forma noua si implementarea ulterioara a acestei forme.

Fazele procesului de reengineering pentru un sistem software orientat pe obiecte sunt urmatoarele [9]:

- **Analiza cerintelor**. Pot exista motive diverse pentru a demara un proces de reengineering asupra unui program. Spre exemplu, poate fi necesar sa marim flexibilitatea programului în vederea introducerii de noi functionalitati. Pentru a ne putea concentra în fazele urmatoare doar asupra partilor relevante ale unui sistem software mare, cerintele pe care noul sistem trebuie sa le îndeplineasca trebuie statuate si analizate în aceasta faza.

- **Capturarea modelului.** De multe ori sistemele “mostenite” sunt slab documentate si deci greu de înteles. Scopul acestei faze este de a întelege programul în ansamblu. Se formeaza astfel un model al sistemului software.

- **Detectia problemelor.** În aceasta faza se identifica violari ale cerintelor specificate în prima faza. Ca rezultata trebuie sa se obtina un set de parti ale programului în care apar aceste violari.

- **Analiza problemelor.** Dupa ce partile critice ale sistemului au fost identificate, ele trebuie inspectate urmarind sa proiectam structuri noi care sa le înlocuiasca pe cele vechi si care sa se conformeze cerintelor din prima faza.

- **Reorganizarea.** În aceasta faza are loc transformarea propriu-zisa, înlocuind vechile structuri cu cele noi.

Observam ca procesul de reengineering este unul complex ce implica un efort ridicat. O faza esentiala si dificila a acestui proces este Detectia Problemelor, în care se identifica diferite structuri afectate de carente de proiectare. Identificarea si modificarea acestor structuri stau practic la baza întregului proces. Este deci necesar sa detinem mecanisme puternice care sa faciliteze în primul rând identificarea structurilor “bolnave”.

3.2.2 Metode de detectie informale

În capitolul anterior am prezentat un set de criterii ce caracterizeaza un design orientat pe obiecte bun. În concordanta cu aceste criterii, o buna proiectare trebuie sa fie caracterizata de o complexitate redusa, trebuie sa furnizeze o buna abstractizare a datelor, trebuie sa reduca cuplajul dintre entitatile de design în timp ce coeziunea lor trebuie sa fie una ridicata. Avem nevoie de aceste criterii pentru ca în primul rând ele ne ajuta sa definim ce înseamna o proiectare buna. Pe de alta parte ele pot servi unui scop mult mai pragmatic: ne ajuta sa evitam decizii de proiectare inadecvate. În consecinta, problema obtinerii unui design bun se reduce la evitarea acelor caracteristici ce induc consecinte negative. Din experienta acumulata de dezvoltatorii de software care au încercat sa evite caracteristicile cu consecinte negative asupra design-ului orientat pe obiecte, si-au facut aparitia o serie de euristici, reguli si principii de proiectare pe care le vom defini în continuare asa cum apar în [14].

Definitie. O afirmatie sau o informatie derivata din experienta acumulata pe parcursul operarii cu o anumita metodologie de dezvoltare software si a carei aplicare îmbunatateste calitatea design-ului se numeste euristica de proiectare.

Definitie. O euristica de proiectare care statueaza o necesitate ori o interdictie cu privire la caracteristicile unui design se numeste regula de proiectare.

Definitie. O euristica de proiectare care exprima într-o maniera abstracta un anumit criteriu de evaluare a unui design se numeste principiu de proiectare.

Euristicile, regulile si principiile de proiectare reprezinta cunostinte foarte valoroase ajutându-l pe dezvoltator sa satisfaca criteriile unei proiectari orientate pe obiecte corecte. Mai mult, aceste reguli includ si criterii de identificare a fragmentelor

de design critice, foarte probabil afectate de carente de proiectare, sugerând în acelasi timp si modalitati de transformare a lor (ex. "If a class has too many methods, split it!").

O alta modalitate de identificare a carentelor de proiectare o reprezinta identificarea si descrierea simptoamelor acestora. Este exact ceea ce urmareste Fowler în [7] prin definirea conceptului de "bad-smell" despre care am vorbit mai sus. Pentru localizarea carentei ne ramâne doar sa analizam diferite structuri de design si sa vedem daca ele prezinta simptoamele unei anumite carente de proiectare.

3.2.3 Dezavantajele metodelor informale

Metodele de detectie a carentelor de proiectare amintite în paragraful anterior prezinta o serie de limitari importante[14].

1. Cea mai mare limitare a regulilor de design este reprezentata de nivelul de abstractizare eterogen al acestora. Regulile sunt fie concrete (ex. "a class should not contain more than six objects"), fie foarte abstracte (ex. "avoid centralized control"). Datorita acestui aspect, regulile sunt greu de aplicat într-o maniera sistematica de unde si caracterul informal al metodei de detectie.

2. În practica, euristicile concrete sunt de obicei respectate pe când cele mai generale sunt ignorate. Din nefericire, cele din urma au un impact mai mare asupra calitatii design-ului. Motivul pentru care regulile concrete sunt des utilizate de catre proiectanti rezida în faptul ca acestea sunt foarte simplu de cuantificat. Regulile abstracte sunt mult mai greu de cuantificat.

3. Simptoamele carentelor de proiectare descrise de Fowler sunt în principiu mai simplu de cuantificat decât regulile de proiectare cu caracter general. Totusi, apare o alta problema: ramâne la latitudinea persoanei care analizeaza un anumit fragment de design sa decida câte linii de cod într-o metoda sunt prea multe linii de cod sau câti membrii într-o clasa sunt prea multi membrii. De aici imposibilitatea de detectie sistematica a carentei de proiectare de care sufera o anumita entitate de design.

Ca urmare a caracterului informal precum si a imposibilitatii de cuantificare simpla a regulilor de proiectare cu un caracter general, metodele de identificare a carentelor de proiectare amintite mai sus nu se preteaza pentru o implementare într-o maniera automata. În consecinta aceste metode sunt:

- **Consumatoare de timp.** Datorita lipsei de rigurozitate în detectia carentelor, metodele sunt greu de aplicat în practica implicând consumul unei mari cantitati de timp si energie.

- **Nerepetabile.** Anumite aspecte legate de detectia problemelor depind de la caz la caz limitându-se reutilizarea lor în alte contexte.

- **Nescalabile.** Metodele manuale sunt practic imposibil de aplicat pe sisteme mari. Cele mai multe sisteme software industriale sunt foarte mari (de ordinul milioanelor de linii de cod).

În concluzie, pentru a accelera faza de detectie a problemelor din cadrul procesului de reengineering, avem nevoie de instrumente automate de detectie a carentelor de proiectare. Acest lucru implica necesitatea definirii unor metode de detectie sistematice, a unor mecanisme si metodologii ce sa suporte cuantificarea euristicilor.

3.3 Elemente necesare detectiei sistematice

3.3.1 Simptoame vs. Semne

Conceperea unor mecanisme de detectie sistematica a carentelor de proiectare nu este un lucru foarte simplu. Pentru o mai buna întelegere a elementelor necesare obtinerii acestui deziderat vom apela la o interesanta analogie între mecanismele medicale de detectie si identificare a bolii de care sufera o persoana si mecanismele actuale utilizate în detectia carentelor de proiectare. Vom începe prin definirea conceptului de simptom [27].

Definitie. Simptoamele bolilor au un caracter *subiectiv*, sunt relatate de catre bolnav si pot fi variabile de la un caz la altul, în functie de modul de perceptie al fiecarui pacient.

Datorita caracterului subiectiv al conceptului de simptom este practic imposibil de a determina în maniera sistematica de ce boala sufera pacientul. Prin analogie, detectia carentelor de proiectare, care la urma urmei pot fi privite ca boli ale fragmentelor de design, nu este posibila în maniera sistematica numai pe baza simptoamelor descrise anterior. Acest lucru se datoreaza factorului subiectiv ce actioneaza la nivelul simptomatologiei carentei de proiectare. Acelasi factor subiectiv apare si la metodele de detectie bazate pe utilizarea euristicilor de proiectare. În concluzie, pentru a obtine un mecanism sistematic este necesar sa utilizam un concept obiectiv. Acest concept îl gasim tot în medicina [27].

Definitie. Semnele de boala sunt manifestari obiective, deci mai exacte, constatate de medic sau chiar de persoana bolnava, astfel ca nu se poate exclude în totalitate componenta subiectiva în evaluarea semnelor de boala.

Faptul ca semnul de boala este mult mai obiectiv se datoreaza în primul rând faptului ca cele mai importante semne sunt masurabile într-o anumita masura. Sa consideram spre exemplu semnul de boala icter. Icterul se manifesta la nivel de perceptie prin îngalbenirea pielii. Pielea galbena este însa un simptom pentru ca depinde de subiectivitatea observatorului. Daca efectuam însa o analiza medicala se constata ca bilirubina din sânge depaseste 1mg% ceea ce indica imediat prezenta semnului de boala icter. Un complex de semne si simptoame constituie asa numitul termen de sindrom, iar un complex de sindroame indica boala propriu-zisa. Pentru simplitatea analogiei vom considera o relatie de unu la unu între sindrom si boala desi din punct de vedere medical acest lucru nu este totdeauna adevarat.

Prin analogie, observarea prezentei unor simptoame ale unei carente de proiectare declanseaza necesitatea efectuarii unor masuratori peste fragmentele de design suspecte. Valori anormale ale acestor masuratori vor indica prezenta unor semne asociate carentei de proiectare de care sufera un fragment de design. Un complex de semne va indica problema propriu-zisa. Prin urmare, pentru o detectie

sistematica sunt necesare modalitati de masurare a diferitelor atribute ale fragmentelor de design, stabilirea unor valori anormale pentru diferite masuratori care sa ne indice semnele de boala si compunerea acestor semne într-o maniera care sa indice în mod corect carenta de proiectare.

S-ar putea aprecia ca euristicile de proiectare sunt capabile sa indice semne ale carentelor de proiectare. Este perfect adevarat. Problema lor este însa alta. Pe de-o parte marea lor majoritate sunt prea abstracte, neindicând ce masuratori trebuie aplicate si valorile anormale ale acestora (ex. "avoid centralized control"). Pe de alta parte, multe euristici concrete sunt prea simple ("a class should not contain more than six objects"), indicând doar un sigur semn de carenta de proiectare si nu un complex de semne, lucru ce face imposibila identificarea bolii. Acesta este motivul pentru care detectia bazata pe euristici nu poate fi una sistematica.

Metoda de detectie propusa de Fowler sufera în primul rând datorita factorului subiectivitate. Totusi, el ofera anumite indicii asupra masuratorilor ce trebuie efectuate si asupra valorilor anormale asociate acestor masuratori, sugerând migrarea de la notiunea de simptom la cea de semn. Totusi, nu se furnizeaza un mecanism de compunere a acestor semne si nici nu se specifica cum anume ar trebui compuse acestea. El chiar afirma ca "nici un set de metrici nu poate concura cu intuitia umana". Este evident ca nu putem combate aceasta afirmatie, dar nici nu putem s-o acceptam pentru ca solutia propusa de el nu poate fi utilizata în contextul dimensiunilor sistemelor software actuale. Mai mult, analizele medicale sunt utilizate cu succes în medicina pentru identificarea bolii de care sufera o persoana. De ce masuratorile efectuate asupra fragmentelor de design nu ar putea avea aceeasi utilitate? Este adevarat ca uneori experienta medicului ce analizeaza rezultatele este esentiala. Totusi, la dimensiunile software-ului de azi preferam o metoda de detectie automata care sa mai greseasca din când în când, bazându-ne în astfel de situatii pe experienta inginerului software.

3.3.2 Metrici software

"You cannot controle what you cannot measure"[6]. Aceasta afirmatie celebra subliniaza în mod clar ca masuratorile joaca un rol important în îmbunatatirea sistemelor în general si a sistemelor software în particular. Înainte de a ne opri asupra metricilor software în contextul organizarii orientate pe obiecte, este necesara o scurta prezentare a conceptului de masurare. Vom prelua definitiile din [14] bazate la rândul lor pe definitiile lui N.Fenton si S.L. Pfleeger.

Definitie. Prin masurare se întelege procesul prin care numere ori simboluri sunt asociate atributelor entitatilor din lumea reala, într-o asemenea maniera încât acestea sa le descrie în conformitate cu anumite reguli bine definite.

Conceptul de entitate si atribut utilizate în aceasta definitie sunt definite în continuare.

Definitie. O entitate este definita ca fiind subiectul procesului de masurare. O entitate poate fi un obiect , o specificare software sau o faza a unui proiect.

[6] T. DeMarco. *Controlling Software Projects: Management, Measurements and Estimation.* Yourdan Press New Jersey, 1982.

Definitie. Un atribut este o proprietate ce caracterizeaza o entitate. De exemplu, un atribut al unei specificari software este lungimea sa, iar un atribut al unei faze de proiect poate fi durata sa.

Informal, asocierea realizata trebuie sa conserve orice observatie empirica si intuitiva referitoare la atributele si entitatile supuse procesului de masurare. În multe situatii un atribut poate avea diferite întelesuri pentru persoane diferite. Pentru a evita acest lucru este necesar sa definim un model.

Definitie. Un model este expresia unui punct de vedere referitor la entitatea ce este supusa procesului de masurare.

Odata ce un model a fost ales este posibil sa determinam relatii între atributele ce descriu entitatea masurata. Necesitatea existentei unui model bine definit este în particular relevanta în cazul masuratorilor efectuate în ingineria software. Spre exemplu, chiar si pentru o metrica simpla cum este lungimea unui program, este necesara existenta unui model bine definit al programului, care sa ne permita sa identificam într-o maniera neambigua liniile de program.

3.3.3 Metrici software în contextul orientarii pe obiecte

Scopul acestui paragraf este de a prezenta cele mai importante metrici asociate sistemelor software organizate orientat pe obiecte. Metricile prezentate sunt clasificate dupa caracteristicile interne esentiale organizarii orientate pe obiecte [14].

A. Masurarea cuplajului

Numarul de colaboratori

Declararea unei instante a unei clase server într-o clasa client implica posibilitatea existentei unei colaborari între cele doua clase. Aceasta colaborare este masurata de metrici ca Fan-Out sau CBO. Daca doua clase colaboreaza, atunci o unitate se aduna la valoarea Fan-Out indiferent de câte mesaje schimba cei doi colaboratori.

Numarul de servicii

În cazul în care doua clase colaboreaza, o alta posibilitate de masurare a cuplajului o constituie contorizarea numarului de servicii unice accesate. O metrica ce contorizeaza numarul de metode accesate este RFC, definita ca numarul total de metode ce pot fi invocate din clasa masurata. În consecinta :

$$RFC = NLM + NRM$$

unde NLM (Number of Local Methods) reprezinta numarul de metode locale ce pot fi invocate, iar NRM (Number of Remote Methods) numarul de metode nelocale ce se pot accesa. Termenul NRM reprezinta o masura a cuplajului contorizând numarul de servicii oferite de clasele server unei clase client.

Numarul de accese

Daca o metoda apartinând unei clase server este accesata din diferite parti ale unei clase client, fiecare acces poate fi contorizat. Metrica MPC contorizeaza numarul de mesaje trimise din cadrul unei clase. Exista si o metrica ce cuantifica cuplajul la nivel de metoda. Este vorba de metrica MC (Method Coupling) definita ca numarul de referinte nelocale ce apar în cadrul unei metode.

B. Masurarea atributelor de mostenire

Aceasta categorie de masuratori cuantifica adâncimea si densitatea ierarhiilor de clase. Ele pot fi efectuate atât la nivel de clasa cât si la nivel de sistem. La nivel de clasa, o metrica esentiala este DIT (Depth of Inheritance Tree) care se defineste ca lungimea lantului de mosteniri dintre radacina ierarhiei de clase si clasa masurata. La nivel de sistem se poate calcula o medie a acestei metrici.

O alta metrica importanta este SIX (Specialization Index) definita ca:

$$SIX = \frac{NORM \cdot DIT}{NOM}$$

unde NORM (Number of Overriden Methods) se defineste ca numarul de metode mostenite suprascrise în clasa masurata, iar NOM (Number of Methods) reprezinta numarul total de metode din clasa masurata. Metrica este importanta pentru ca permite diferentierea sistematica între cazurile în care mostenirea este utilizata ca mecanism de definire a implementarii unui obiect în termenii implementarii unui alt obiect (Implementation Sub-Classing) si cazurile în care mostenirea este utilizata ca mecanism de specializare (Specialization Sub-Classing). În primul caz valorile metricii vor fi scazute, iar în al doilea caz vor fi ridicate.

Alte metrici se orienteaza catre contorizarea numarului de clase ce extind o anumita clasa. NOC (Number of Children) contorizeaza numarul de clase ce extind direct clasa masurata, iar NOD (Number of Descendents) generalizeaza ideea contorizând nu doar subclasele imediate, ci toti descendentii clasei masurate.

U (Reuse Ratio) este o metrica ce se calculeaza la nivel sistem si cuantifica gradul de reutilizare din interiorul ierarhilor de clase. Ea se defineste ca numarul total de superclase raportat la numarul total de clase. O valoare apropiata de 1 indica o proiectare deficitara deoarece ierarhia de clase tinde sa fie liniara, iar o valoare apropiata de 0 indica o ierarhie putin adânca, puternic ramificata si cu un numar mare de clase frunze. Asemanator, S (Specialization Ratio) se defineste ca numarul total de subclase raportat la numarul total de superclase. O valoare apropiata de 1 indica o proiectare slaba, iar o valoare apropiata de 0 indica o buna operatie de abstractizare la nivelul superclaselor.

C. Metrici de coeziune

LCOM (Lack of Cohesion in Methods) reprezinta diferenta dintre numarul de perechi de metode din clasa masurata care folosesc în comun variabile instanta si numarul de perechi de metode care nu utilizeaza în comun astfel de variabile. Aceasta metrica prezinta o serie de neajunsuri. Pe de-o parte exista un numar mare de situatii distincte în care metrica ia valoarea 0, iar pe de alta parte nu exista indicatii asupra modului de interpretare a fiecarei valori pe care poate sa o ia metrica. O varianta

îmbunatatita a acestei metrici este LCOM* care se bazeaza pe notiunea de coeziune perfecta. Valoarea pe care o ia metrica la masurarea unui anumit caz particular reprezinta un procent din aceasta valoare perfecta.

O alta critica adresata metricii LCOM consta în incapacitatea ei de a distinge între clasele partial coezive, fiind eficienta doar în identificarea claselor puternic necoezive. Aceste probleme sunt eliminate în cadrul metricilor TCC (Tight Class Cohesion) si LCC (Loose Class Cohesion). Ele sunt definite pe baza accesarii variabilelor instanta de catre perechi de metode apartinând clasei masurate. Doua metode se considera a fi direct conectate daca ambele acceseaza cel putin o variabila instanta comuna. Doua metode sunt indirect conectate daca acceseaza cel putin o variabila instanta comuna prin intermediul invocarii unei alte metode.

Ca urmare, TCC se defineste ca numarul relativ de perechi de metode din clasa masurata conectate direct, iar LCC se defineste ca numarul relativ de perechi de metode direct ori indirect conectate.

D. Masurarea dimensiunii si complexitatii structurale

Aceasta categorie de metrici este extrem de importanta datorita usurintei cu care ele pot fi interpretate si datorita valorii informatiilor derivate din aceasta interpretare. Aplicând metrici de dimensiune la nivel sistem putem obtine o privire de ansamblu asupra dimensiunii sistemului, în timp ce metricile de complexitate ne pot induce o prima imagine asupra complexitatii structurale a sistemului. Aplicând aceasta categorie de metrici la nivel de clase ne asteptam sa determinam mai usor clasele ce joaca un rol important în design, precum si clasele prea mari si prea complexe.

Metrici de dimensiune

Datorita faptului ca sistemele software organizate orientat pe obiecte prezinta mai multe niveluri de abstractizare, putem defini metrici de dimensiune specifice fiecarui nivel. La nivel sistem o metrica des utilizata este numarul de clase (Number of Classes in the System) care poate fi mai departe rafinata sub forma numarului de clase abstracte si numarului de clase concrete. La cel mai coborât nivel de abstractizare, metricile de dimensiune a metodei (Method Size) pot indica cât de "orientata pe obiecte" este o clasa, stiind ca metode prea mari într-o clasa reprezinta un simptom al devierii de la aceasta forma de organizare. Dintre metricile de dimensiune asociate claselor amintim: SIZE1 care se defineste ca numarul de ';' dintr-o clasa si SIZE2 care reprezinta numarul de atribute si metode externe pentru clasa masurata.

Cele mai simple metrici de dimensiune si complexitate asociate claselor sunt NOA (Number of Attributes) respectiv NOM (Number of Methods). Exista si versiuni în care se tine cont de distinctia între variabilele instanta (NIV – Number of Instance Variables) si variabilele de clasa (NCV – Number of Class Variables). Relatia dintre NOA si ultimele doua metrici prezentate este evidenta:

$$NOA = NIV + NCV$$

În aceeasi maniera se poate distinge între metodele instanta (NIM – Number of Instance Methods) si metodele de clasa (NCM – Number of Class Methods). Mai mult, metodele mai pot fi clasificate în metode externe sau publice (NEM – Number

of External Methods) si metodele interne sau private (NHM – Number of Internal Methods). Relatiile existente între aceste metrici sunt sugerate în urmatoarea expresie:

$$NOM = NEM + NHM = NIM + NCM$$

Masurarea complexitatii structurale

Pentru calculul complexitatii structurale a unei clase complexitatea tuturor metodelor trebuie însumata. Aceasta este ideea care sta la baza metricii WCM (Weighted Method per Class).

$$WMC = \sum_{i=1}^{n} c_i$$

unde c_i este complexitatea statica a metodei m_i (ex. complexitatea ciclomatica), iar n este numarul total de metode din clasa masurata. Daca c_i este unitar atunci WMC este echivalenta cu NOM.

O observatie importanta legata de aceasta metrica este aceea ca ea poate fi utilizata în procesul de reengineering în faza de capturare a modelului. Metrica poate fi utilizata în detectia claselor ce controleaza sistemul, pornind de la prezumtia ca respectivele clase sunt mai complexe decât celelalte clase din sistem.

3.3.4 Cuantificarea principiilor de proiectare

Într-o suita de articole [16,17,18,20,21,22], R. Martin sintetizeaza un numar de principii de proiectare orientata pe obiecte. Principiile sunt organizate pe doua niveluri de abstractizare: principii la nivelul proiectarii claselor si principii la nivelul arhitecturii pachetelor. Martin mai propune un set de metrici pentru cuantificarea anumitor principii [20].

Vom prezenta aceasta abordare prin exemplificarea unui astfel de principiu cuantificat. Principiul Dependentelor Stabile, asociat stabilitatii pachetelor, este formulat initial într-o maniera abstracta: “Depend in the direction of stability”. În aceasta forma principiul este greu de aplicat. El propune însa doua metrici pentru cuantificarea acestui principiu: prima masoara cuplajul de dependenta (C_e) si reprezinta numarul de clase din pachet ce depind de clase din afara pachetului masurat, iar a doua masoara cuplajul de responsabilitate (C_a) si reprezinta numarul de clase exterioare pachetului masurat ce depind de clase din interiorul respectivului pachet. Pe baza acestor metrici Martin defineste factorul de instabilitate (I) definit astfel:

$$I = \frac{C_e}{C_e + C_a}$$

Principiul se reformuleaza în urmatoarea forma: “Depend upon packages whose I metric is lower than yours”. Principiul devine mult mai usor de aplicat si permite în acelasi timp verificarea automata a conformarii la acest principiu a unei structuri de pachete.

Din nefericire, abordarea lui Martin se limiteaza la cuantificarea a doua principii, ambele dedicate proiectarii la nivelul pachetelor. El nu sugereaza o metodologie sistematica pentru cuantificarea altor principii de proiectare.

3.3.5 Detectia carentelor de proiectare utilizând metrici software

Desi exista metrici software capabile sa cuantifice caracteristicile interne specifice unei proiectari corecte orientate pe obiecte, utilizarea lor în contextul detectiei carentelor de proiectare ridica o serie de probleme. Astfel, cele mai importante euristici de proiectare (ex. "avoid centralized control") sunt prea abstracte pentru a putea fi cuantificate în termeni unei singure metrici. Cauza o constituie incapacitatea unei metrici de a captura mai mult decât un singur aspect dintre cele relevante în contextul problemei ce se vrea a fi detectata. În esenta, aceasta înseamna ca modelul de interpretare al unei metrici ne poate ajuta sa întelegem "semnul de boala" reflectat în valorile anormale ale metricii, dar nu ne poate ajuta sa întelegem "boala" de care sufera fragmentul de design masurat. Principalul motiv este acela ca un "semn de boala" poate apare în contextul mai multor "boli" distincte. În consecinta, interpretarea individuala a unei masuratori are o granularitatea prea mica pentru a putea indica "boala" propriu-zisa. Concluzionând, exista o distanta prea mare între lucrurile pe care le masuram si lucrurile importante din punctul de vedere al proiectarii.

Pentru depasirea acestei limitari si utilizarea eficienta a metricilor în contextul detectiei carentelor de proiectare, în [14] se introduce un nou concept si mecanism: strategia de detectie. La baza acestui concept sta ideea utilizarii simultane a mai multor masuratori care împreuna sa surprinda în totalitate aspectele relevante în contextul unei carente de proiectare. Prin posibilitatea de reprezentare într-o forma cuantificabila a acestor aspecte, strategia de detectie face posibila detectia sistematica a carentelor de proiectare si face posibila implementarea într-o maniera automata a fazei de detectie a problemelor din cadrul procesului de reengineering.

3.4 Strategii de detectie

3.4.1 Definire

Vom prelua definitia strategiilor de detectie asa cum apare în [11,14]:

Definitie. Se numeste strategie de detectie o expresie cuantificata a unei reguli prin care fragmente de design conforme cu respectiva regula pot fi detectate în codul sursa.

Dupa cum reiese din definitie, strategia de detectie reprezinta un mecanism generic pentru analizarea modelului codului sursa utilizând metrici software. Înainte de a prezenta mecanismele de filtrare si compozitie care stau la baza conceptului, trebuie sa atragem atentia asupra unor aspecte importante:

- În contextul definitiei anterioare, afirmatia "o expresie cuantificata a unei reguli" denota necesitatea ca respectiva regula sa poata fi corect exprimata în termeni de metrici software.

- La fel ca în [14], în aceasta lucrare strategiile de detectie sunt utilizate pentru cuantificarea unor reguli care ajuta la detectia problemelor de design în sistemele software organizate orientat pe obiecte (spre exemplu detectarea fragmentelor de design afectate de o anumita carenta de proiectare). Trebuie

însa remarcat ca strategiile de detectie pot fi utilizate si în alte scopuri cum ar fi detectia anumitor tipare de proiectare.

3.4.2 Mecanismul de filtrare

Definitie. Un filtru de date este un mecanism (un operator pentru seturi) prin care un subset de date din setul initial de rezultate ale masuratorii este retinut pe baza scopului particular masuratorii.

Scopul unui filtru de date este de a reduce setul initial de date astfel încât sa fie retinute numai acele valori ce prezinta anumite caracteristici speciale. În contextul detectiei carentelor de proiectare, scopul filtrarii datelor este acela de a detecta acele fragmente de design care prezinta caracteristici speciale capturate de metricile calculate. Considerând setul initial ca fiind sortat, un filtru de date defineste un subset de date contiguu. În consecinta, înainte de a defini un filtru de date trebuie sa specificam valorile asociate limitelor inferioare si superioare ale subsetului filtrat. Pornind de la aceste considerente, în [14] filtrele de date sunt clasificate conform tabelului urmator.

Tipul filtrului	**Specificarea limitelor**		**Exemplu**
Marginal	Semantic	Relativ	• TopValues(10) • BottomValues(5%)
		Absolut	• HigherThan(10) • LowerThan(6)
	Statistic		• BoxPlot
Interval	Compozitie de doua filtre marginale cu specificarea semantica a limitelor (având polaritati diferite)		Between(20,30) : = HigherThan(20) and LowerThan(3)

Tabel 3.1 Clasificarea filtrelor de date

Filtrele marginale sunt definite ca filtre pentru care una dintre limitele setului rezultat este implicit determinata de limita corespunzatoare din setul de date initial. Filtrele interval sunt acele filtre pentru care ambele limite sunt specificate explicit. Filtrele marginale pot fi la rândul lor de doua tipuri: semantice sau statistice. În prima categorie intra acele filtre pentru care trebuie sa precizam o valoare de prag si o directie. Valoarea de prag reprezinta limita care trebuie specificata explicit în timp ce directia indica daca este vorba de o limita superioara ori inferioara. A doua categorie de filtre marginale sunt asa numitele filtre statistice. În cazul lor nu trebuie specificata o valoare de prag pentru ca aceasta este determinata implicit din setul de date initial folosind metode statistice. Singurul lucru ce trebuie specificat este directia. În continuare vom prezenta principalele filtre propuse în [14].

- **HigherThan si LowerThat.** Sunt filtre semantice, care pot fi parametrizate doar cu valoarea de prag. Cele doua directii posibile sunt implicit cuprinse în denumirea filtrului. Aceste filtre mai pot fi numite si filtre absolute, pentru ca valoarea de prag reprezinta o valoare absoluta.

- **TopValues si ButtomValues.** Aceste filtre determina setul rezultat pe baza unui parametru ce specifica numarul de entitati retinute si nu valoarea minima sau maxima permisa în setul rezultat. Valorile din setul rezultat vor fi deci relative la setul de date initial. Parametrul poate fi absolut (ex. "pastreaza 20 de entitati cu cele mai ridicate valori") sau procentual (ex. "pastreaza 10% dintre entitatile masurate având cele mai mici valori"). Cele doua directii posibile sunt cuprinse implicit în numele filtrului.

- **BoxPlots.** Este un filtru statistic care utilizeaza metoda box-plots prin care valori anormale dintr-un set de date pot fi detectate. Nu vom discuta alte aspecte legate de acest filtru pentru simplul motiv ca nici una dintre strategiile de detectie utilizate în detectia carentelor de proiectare nu-l utilizeaza.

3.4.3 Mecanismul de compozitie

Dupa cum am spus la început o strategie de detectie este o expresie cuantificabila a unei reguli de proiectare. Spre deosebire de metrici, o strategie de detectie trebuie sa cuantifice întreaga regula nu doar un atribut al entitatii masurate. În consecinta, pe lânga mecanismul de filtrare care interpreteaza rezultatele masuratorilor individuale, mai este necesar un alt mecanism care sa suporte o interpretare corelata a mai multor seturi de rezultate [14].

Definitie. Operatorii utilizati în compunerea unui set de metrici într-o regula "articulata" sunt denumiti operatori de compozitie.

Exista definiti un numar de trei operatori: AND, OR, BUTNOTIN. Ei pot fi priviti din doua puncte de vedere diferite:

1. Din punct de vedere logic, cei trei operatori sunt o reflectie a "punctelor de articulatie" din cadrul regulii cuantificate de strategia de detectie, în care "operanzii" sunt descrieri ale simptoamelor carentelor de proiectare. Operatorul AND sugereaza necesitatea prezentei atât a simptomului descris în partea dreapta a operatorului cât si a celui din partea stânga. Se observa cum strategiile de detectie urmeaza mecanismul medical prezentat mai devreme. Metricile pot fi puse în corespondenta cu analizele medicale, iar valorile anormale ale lor sesizate de mecanismul de filtrare precizeaza daca un anumit semn de "boala" este sau nu prezent. Operatorul AND se va plasa între doua semne care coexista întotdeauna în contextul bolii detectate de strategie.

2. Din punctul de vedere al operatiilor cu seturi, se întelege modul în care se construieste rezultatul final în urma aplicarii unei strategii de detectie. Prin intermediul mecanismului de filtrare, seturile initiale de rezultate asociate metricilor implicate în strategie sunt reduse la niste subseturi de entitati de design (însotite de valorile masurate) care sunt considerate suspicioase pe baza modelului de interpretare al metricii respective. În continuare, seturile rezultate în urma filtrarii sunt combinate utilizând operatorii de compozitie. Operatorul AND corespunde intersectiei a doua seturi, OR corespunde operatiei de reuniune, iar BUTNOTIN reprezinta diferenta a doua seturi.

3.4.4 Exemple

În continuare vom prezenta strategiile de detectie utilizate pentru localizarea unor carente de proiectare descrise în paragraful 3.1.5. Strategiile de detectie sunt preluate din [11,13,14]. Fiecare regula este însotita de o scurta descriere a particularitatilor carentei de proiectare asa cum deriva ele din regula informala ce urmeaza a fi cuantificata precum si o scurta descriere a metricilor implicate.

A. DataClasses

Strategie

Se vor detecta clasele cu urmatoarele caracteristici: vom cauta clase "usoare" (ex.clase care nu furnizeaza aproape nici un fel de functionalitate prin interfata lor). În continuare vom urmari clasele care au în interfata lor multe metode de tip accesor (ex.metode de tip set / get) si clasele care definesc câmpuri de date în interfetele lor.

Regula

DataClasses:= ((WOC,BottomValues(33%)) and (WOC,LowerThan(0.33))) and ((NOPA,HigherThan(5)) or (NOAM, HigherThan(5)))

Metrici

- **Weight of a Class (WOC)**: metrica reprezinta numarul de metode non-accesor din interfata clasei masurate raportat la numarul total de membrii ai interfetei

- **Number of Public Attributes (NOPA)**: numarul de atribute ce nu sunt mostenite si care apartin interfetei clasei masurate

- **Number of Accessor Methods (NOAM)**: numarul de metode de tip accesor ce nu sunt mostenite si care sunt declarate în interfata clasei masurate

B. GodClasses

Strategie

Detectia acestei carente de proiectare se bazeaza pe urmatoarele caracteristici: este de asteptat ca un GodClass sa acceseze multe date în mod direct sau prin metode de tip accesor, date apartinând unor clase "usoare"; este de asteptat ca aceste clase sa fie mari si sa aiba un comportament non-comunicativ. În prima instanta vom detecta acele clase care depind puternic de datele unor clase "usoare", dupa care vom filtra lista de suspecti obtinuta eliminând clasele mici si / sau coezive.

Regula

GodClasses:= ((ATFD,TopValues(20%)) and (ATFD,HigherThan(4))) and ((WMC,HigherThan(20)) or (TCC,LowerThat(0.33)))

Metrici

- **Access to Foreign Data (ATFD)**: reprezinta numarul de clase externe ale caror atribute sunt accesate de catre clasa masurata, în mod direct sau indirect prin metode de tip accesor

- **Weighted Method Count (WMC)**: **r**eprezinta suma complexitatii statice pentru toate metodele clasei masurate

- **Tight Class Cohesion (TCC)**: reprezinta numarul relativ de metode conectate direct. Doua metode sunt conectate direct daca ambele acceseaza o variabila instanta ce apartine clasei

3.4.5 Definirea si utilizarea strategiilor de detectie

Un aspect important al utilizarii strategiilor de detectie pentru identificarea si localizarea carentelor de proiectare îl constituie modul în care se ajunge de la regula de proiectare la strategia de detectie propriu-zisa. În [14] se furnizeaza metodologia prin care o regula informala de proiectare este cuantificata sub forma unei strategii de detectie si modul în care respectiva strategie este aplicata. Abordarea consta în urmatoarea secventa de pasi:

1. **Analiza regulii de proiectare.** Dupa alegerea reguli de proiectare care trebuie cuantificata, primul pas consta în exprimarea descrierii informale a reguli într-o maniera cantitativa. Cu alte cuvinte, trebuie descrise legaturile regula-fragmente de design (ex. în cadrul reguli ce descrie carenta GodClass se sugereaza ca ne vom confrunta cu clase de dimensiuni mari) si relatiile existente între aceste fragmente de design (ex. un GodClass este puternic dependent de date ce apartin unor clase "usoare"). Rezultatul acestei etape consta în definirea unei strategii concrete pentru detectia fragmentelor de design conforme cu regula de proiectare. Aceasta strategie este de fapt o descriere a caracteristicilor entitatilor de design conforme cu regula aflata în proces de cuantificare.

2. **Selectia metricilor.** Pe baza descrierii cantitative obtinuta în pasul anterior, trebuie gasite ori definite metricile care surprind cel mai bine caracteristicile entitatilor de design conforme cu regula ce se cuantifica. La sfârsitul acestei etape, strategia de detectie poate fi exprimata sub forma unei combinatii de metrici.

3. **Detectia suspectilor.** În continuare se efectueaza masuratori asupra fragmentelor de design din sistemul examinat, pornind de la metricile selectate în pasul anterior. Se aplica strategia de detectie obtinându-se în final o lista de fragmente de design suspectate a fi conforme cu regula de proiectare cuantificata de strategia de detectie.

4. **Examinarea suspectilor.** Ultima etapa consta în examinarea manuala a fragmentelor de design suspectate. Pe baza codului sursa si a altor surse de informare (ex. documentatia sistemului) se decide daca un anumit

fragment de design este într-adevar conform cu regula cuantificata de strategia de detectie ce a indicat ca si suspect fragmentul analizat.

Se poate observa ca primi doi pasi sunt dedicati definirii unei strategii de detectie, iar urmatorii doi aplicarii respectivei strategii. În continuare ne vom limita la faza de definire a strategiei unde o problema majora o constituie selectia valorii pragurilor asociate filtrelor de date.

3.4.6 Problema pragurilor

Sa reanalizam procesul de definire a unei strategii de detectie. Vom prezenta acest proces într-o maniera abstracta independenta de natura problemei ce se doreste a fi detectata, exemplificând doar cu situatii întâlnite în contextul detectiei carentelor de proiectare în sistemele software organizate orientat pe obiecte. Din aceasta descriere abstracta s-ar putea întelege ca limitarea identificata în cele ce urmeaza este una general întâlnita, indiferent de natura si domeniul problemei care urmeaza a fi detectata utilizând tehnica strategiilor de detectie. Recunoastem ca pot exista situatii când aceasta limitare nu se întâlneste, dar totusi, în contextul detectiei carentelor de proiectare ea este omniprezenta. În acelasi timp, pot fi gasite exemple de diverse alte contexte în care apare aceeasi limitare.

În figura urmatoare sunt prezentate principalele activitati care se desfasoara în momentul trecerii de la strategia obtinuta în pasul 1 (Analiza regulii) la forma cuantificata a regulii.

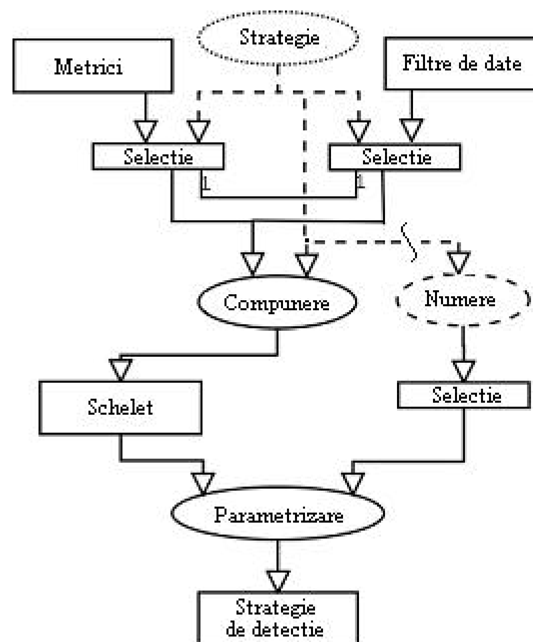

Figura 3.1. Definirea strategiei de detectie.

În pasul al doilea din cadrul procesului de definire a unei strategii de detectie este necesara selectia masuratorilor cele mai relevante în contextul strategiei obtinute în pasul anterior. În acelasi timp, pentru fiecare masuratoare selectata, trebuie furnizat un filtru de date care sa pastreze în vederea analizei ulterioare doar acele entitati (în particular fragmente de design) care prezinta valori anormale ale masuratorii asociate filtrului. Cu alte cuvinte, filtrul trebuie sa pastreze doar acele entitati care prezinta "semnul de boala" identificabil prin masuratoarea asociata lui. În continuare perechile

metrici-filtru de date sunt compuse utilizând operatori de compozitie obtinându-se în final scheletul strategiei de detectie. Acest schelet reprezinta expresia strategiei de detectie care nu contine însa nici un parametru pentru filtrele de date utilizate. Urmeaza procesul de parametrizare al scheletului, adica atribuirea de valori numerice adecvate pragurilor. Aceasta înseamna stabilirea unor limite valorice care fac distinctia între valorile normale ale masuratorilor si valorile anormale ale lor. Procesul de parametrizare este foarte important deoarece asa cum se recunoaste în [14], calitatea strategiei de detectie depinde foarte puternic de corectitudinea acestui proces.

Selectia metricilor nu este chiar atât de complicata deoarece aceasta operatie se bazeaza pe strategia obtinuta în urma analizei regulii. Strategia descrie foarte clar, într-un limbaj specific domeniului din care provine regula informala, proprietatile problemei ce se doreste a fi detectata. În multe situatii, aceste proprietati indica în mod direct masuratorile care trebuie efectuate si mai mult, ele indica si filtrele de date asociate masuratorii. Spre exemplu, în contextul carentei DataClass strategia spune printre altele: "…clase care definesc câmpuri de date în interfetele lor". Evident ca în aceasta situatie o masuratoare relevanta este NOPA (Number of Public Attributes), iar un filtru de date bun este HigherThan plecând de la premisa ca este acceptata existenta unui numar limitat de câmpuri publice în interfata unei clase.

Din pacate nu acelasi lucru se întâmpla în cazul selectiei parametrilor. "Maparea problemei sub forma de numere" este o mare problema în contextul definirii strategiilor de detectie. Urmarind caracterizarea carentei de proiectare DataClass din [14] amintita de noi în sectiunea 3.4.4 si strategia de detectie asociata, putem observa ca o clasa este considerata "usoara" daca: valoarea metricii WOC pentru clasa data este mai mica decât 0.33 si în acelasi timp clasa data se gaseste între primele 33% dintre toate clasele analizate, sortate în ordine crescatoare a valorii WOC asociate fiecareia. În mod natural se pun urmatoarele întrebari: de ce putem considera ca o clasa a carei metrica WOC este 0.4 ori 0.34 nu este o clasa "usoara"? De ce o clasa "usoara" a carei metrica NOPA ia valoarea 4 si nu are metode de tip accesor nu este un DataClass?

Problema fundamentala care ridica întrebarile enuntate mai sus nu rezida atât de mult în faptul ca utilizam valori constante pentru pragurile filtrelor de date. Problema de esenta este ca aceste valori nu sunt obtinute ca rezultat al unui rationament clar. În [14] se furnizeaza o metodologie de selectie a filtrelor de date pe baza modului în care regula de proiectare indica diferite praguri care fac distinctia între valorile normale si cele anormale ale unei metrici. În esenta, se recomanda utilizarea unui filtru semantic absolut când regula precizeaza nemijlocit o valoare de prag. Filtrele semantice relative se recomanda a se utiliza când pragurile sunt specificate în termeni "valori mari / mici" sau "cele mai mari / mici valori". De asemenea, se spune ca este bine sa se parametrizeze aceste filtre cu valori procentuale daca sistemul analizat este mare si cu valori absolute daca sistemul este mic.

Principalul argument care demonstreaza ca metodologia propusa nu este utila îl regasim tot în [14]. Regulile de proiectare cele mai importante sunt prea abstracte si nu furnizeaza nici un fel de informatie referitoare la valorile de prag (ex."avoid centralized control"). Pe de alta parte s-ar putea afirma ca metodologia trebuie aplicata si la nivelul transformarii strategiei obtinute în urma analizei regulii. Din pacate problema ramâne: tot trebuie specificate valori de prag pentru filtrele de date, iar strategia nu spune clar care sunt valorile corecte. O mica variatie a acestor parametrii ar putea sa ascunda carente de proiectare importante în momentul aplicarii strategiei de detectie.

Un alt lucru important de observat consta în faptul ca în [14] se defineste un numar mare de strategii de detectie dedicate identificarii diverselor carente de proiectare. Din pacate, autorul nu spune nimic despre modul în care a fost aplicata metodologia de parametrizare a filtrelor de date utilizate în respectivele strategii de detectie. Acest lucru ridica serioase semne de întrebare atât asupra metodologiei propuse cât si asupra valorilor de prag utilizate în aceste strategii.

Concluzionând, metoda actuala de stabilire a valorilor pragurilor filtrelor de date este vaga si neconvingatoare. Nu avem o metodologie clara si generala pe care sa ne bazam în momentul în care trebuie sa stabilim valoarea unui prag pentru un filtru de date. Cu alte cuvinte, asa cum sugereaza si figura anterioara printr-o linie întrerupta, actualmente exista o distanta prea mare între strategia urmata în definirea strategiei de detectie si valorile numerice asociate pragurilor. Nu putem aprecia "formal" granita dintre bine si rau, granita între valori normale si valori anormale ale unei metrici într-un anumit context. Unii ar putea afirma ca aceasta granita nici nu poate fi stabilita perfect, mai ales datorita valorii constante asociate pragurilor. Raspuns: nici nu urmarim acest lucru. Dupa cum o strategie de detectie furnizeaza o lista de suspecti, noi, ca ingineri, trebuie sa stabilim un echilibru acceptabil, o granita acceptabila între valorile normale si anormale ale unei metrici. Singura cerinta este sa aratam ca am stabilit cel mai bun echilibru posibil la momentul respectiv.

Totusi, dupa cum am mai spus, exista definite multe strategii de detectie pentru diferite carente de proiectare. Aceste strategii de detectie s-au dovedit destul de utile în vederea identificarii carentelor de proiectare din sistemele software orientate pe obiecte ceea ce arata ca valorile pragurilor filtrelor de date nu au fost stabilite chiar întâmplator. Cum metodologia propusa în [14] nu ne poate ajuta prea mult în rezolvarea problemei pragurilor, putem spune ca parametrizarea unei strategii de detectie menite sa detecteze o anumita carenta de proiectare, spre exemplu DataClass, s-a bazat numai pe experienta autorului în ceea ce priveste proiectarea orientata pe obiecte, pe perceptia lui asupra conceptului de DataClass formata în timp. Abstract, aceasta echivaleaza cu o analiza a unui set mental de exemple pozitive si negative de carente de proiectare DataClass în vederea estimarii adecvate a valorii pragurilor filtrelor de date. O astfel de abordare a problemei pragurilor prezinta urmatoarele dezavantaje:

- Nu poate descrie concret modul în care exemplele pozitive si negative trebuie analizate în vederea obtinerii pragurilor asociate filtrelor de date. Acest fapt afecteaza negativ si repetabilitatea analizei.

- Împiedica analiza unui set mare de exemple. Este evident ca într-un set restrâns este putin probabil sa fie surprinse toate particularitatile problemei. Mai mult, ar putea apare în cadrul setului o componenta subiectiva inacceptabila pentru ca exemplele vor reflecta perceptia asupra problemei ce se doreste a fi detectata corespunzatoare cel mult unui grup mic de persoane. Setul ar putea fi cu totul particular mai ales când aceste persoane se cunosc si / sau cunosc modul în care strategia de detectie urmeaza sa detecteze respectiva problema.

Capitolul IV

Conceptul de masina de tuning a strategiilor de detectie

În capitolul anterior am descris cea mai mare problema cu care ne confruntam în momentul definirii unei strategii de detectie: cum anume stabilim parametrii filtrelor de date utilizate? Desi în [10,11,13,14] sunt definite multe strategii de detectie pentru identificarea carentelor de proiectare în sistemele software organizate orientat pe obiecte, nu se precizeaza nimic concret despre modul de stabilire a parametrilor filtrelor de date. Cu alte cuvinte, nu se precizeaza cum a fost traversat golul existent între setul de proprietati ce caracterizeaza o carenta de proiectare si setul de parametrii ce ajuta la identificarea acestor proprietati.

Scopul acestei lucrari este de a clarifica operatia de parametrizare a strategiilor de detectie din cadrul procesului lor de definire. Cu alte cuvinte, va trebui sa descriem o metodologie concreta care odata aplicata sa permita parametrizarea corespunzatoare a mecanismului de filtrare asociat strategiilor de detectie. Asa cum va reiesi din acest capitol, elementul de noutate introdus de noi consta în utilizarea unor exemple pozitive si negative concrete corespunzatoare carentei de proiectare vizate de strategia de detectie ai carei parametrii trebuie stabiliti.

4.1 Un proces de definire incrementala a strategiei de detectie

Dupa cum se specifica în [14] o strategie de detectie identifica entitati de design, fragmente de design suspectate de a fi conforme cu regula de proiectare pe baza careia a fost creata respectiva strategie. Aceste entitati trebuie ulterior inspectate de catre un operator uman pentru a decide în cazul fiecarei entitati daca ea este într-adevar conforma cu respectiva regula. Motivul unei astfel de abordari este evident: nu avem cum sa demonstram ca o strategia de detectie consistenta în raport cu un set de observatii este general valabila pentru orice alta posibila observatie de acelasi tip.

Spre exemplu, sa consideram strategia de detectie corespunzatoare carentei de proiectare DataClass. Am putea constata în urma unor experimente ca strategia de detectie a identificat exact toate clasele afectate de aceasta carenta de proiectare. Nu s-a detectat nici o clasa care sa nu fie un DataClass si nu s-a omis nici una care prezinta aceasta carenta de proiectare. Nu putem însa sa afirmam ca acest lucru se va întâmpla tot timpul. Daca strategia e consistenta cu un numar suficient de mare de observatii, putem spune cel mult ca este probabil ca respectiva strategie de detectie sa functioneze aproximativ corect, fara a putea fi absolut siguri de corectitudinea ei. Aceste afirmatii deriva din [5] unde se arata caracteristica de nesiguranta a inferentei inductive.

În timp, am putea constata ca strategia de detectie identifica drept suspecte clase care nu sunt afectate de carenta de proiectare DataClass. Mai rau, am putea constata ca anumite clase ar trebui detectate dar nu sunt. În contextul procesului de reengineering, dar nu numai, faptul ca anumite clase “bolnave” nu sunt detectate este mai grav decât suspectarea alteia de a fi “bolnava”. La dimensiunile actuale ale programelor putem accepta ca anumite clase sa fie suspectate “nefondat”, dar nu prea ne convine sa cautam manual prea multe entitati de design afectate de o carenta de proiectare. Aceasta lipsa de acuratete poate fi motivata în diferite moduri: strategia de detectie nu surprinde anumite aspecte ale carentei de proiectare pe care spune ca o

identifica, pentru ca acestea nu au fost observate la definirea strategiei; anumite aspecte legate de problema de proiectare sunt surprinse de strategia de detectie mult prea general; o anumita metrica nu surprinde suficient de bine aspectul vizat; un anumit parametru al unui filtru de date specializeaza ori generalizeaza prea mult aspectul vizat de metrica asociata filtrului. La momentul actual, cea mai controversata operatie din procesul de definire a unei strategii de detectie este parametrizarea. Din acest motiv suspectii numarul unu care ar putea cauza acuratetea scazuta a strategiilor de detectie actuale sunt parametrii filtrelor de date.

În concluzie, nu putem crea o strategie de detectie pentru identificarea unei carente de proiectare, dupa care sa o utilizam în aceeasi forma pentru totdeauna. În timp, putem constata ca anumite elemente ale ei trebuie schimbate (un parametru, o metrica, etc.) sau alte elemente trebuie adaugate (alte metrici, alte filtre, etc.). Asadar, o strategia de detectie se dezvolta incremental în timp, evolueaza, tinzând asimptotic catre "perfectiune". Procesul ei de definire nu este unul strict limitat si localizat în timp, ci este unul continuu derulat pe întreaga durata de viata a strategiei. Acest proces este surprins în figura 4.1.

Principalele faze ale acestui proces vor fi descrise în cele ce urmeaza. Se porneste de la regula de proiectare si de la un set initial de observatii, un set initial de exemple pozitive si negative asociate regulii.

1. **Analiza regulii de proiectare.** Prima faza a acestui proces o constituie analiza regulii de proiectare, la fel ca în metodologia propusa în [14] si descrisa de noi în capitolul anterior. Descrierea informala a regulii de proiectare este exprimata într-o maniera cantitativa. Ca rezultat se va obtine strategia concreta urmarita în detectia entitatilor conforme cu descrierea informala. Cu alte cuvinte, se obtine setul de proprietati concrete ale acestor entitati sau ,dupa cum se spune în [11], cazul patologic tipic ce descrie o entitate de design conforma cu aceasta descriere. Setul de exemple poate fi si el util în aceasta faza.

2. **Formarea scheletului strategiei de detectie.** Aceasta faza este amintita partial si în [14] dupa cum rezulta din capitolul anterior. Ea consta în selectia metricilor cele mai relevante în contextul carentei de proiectare ce se doreste a fi detectata, pe baza cazului patologic tipic obtinut în faza anterioara. Mai mult, fiecarei metrici selectate i se asociaza un filtru de date corespunzator, dupa care aceste perechi metrica-filtru de date se compun utilizând operatorii de compozitie pe baza aceluiasi caz patologic. Se obtine astfel scheletul strategiei de detectie, expresia ei care nu contine nici un parametru pentru filtrele de date. Setul de exemple poate fi si el util în aceasta faza.

3. **Parametrizarea.** În continuare scheletul strategiei de detectie trebuie parametrizat corespunzator. În esenta, se regleaza acesti parametrii astfel încât strategia de detectie sa fie consistenta în raport cu setul de exemple avut la dispozitie. Modul concret în care se realizeaza acest lucru nu intereseaza momentan. Ideea este ca la sfârsitul acestei faze se obtine strategia de detectie propriu-zisa parametrizata corespunzator.

4. **Validarea.** Strategia de detectie obtinuta în faza anterioara trebuie validata, adica trebuie sa vedem daca ea detecteaza corect entitatile de

design conforme cu descrierea informala corespunzatoare regulii de proiectare ce sta la baza respectivei strategii de detectie. Acest lucru este absolut necesar pentru ca nu cumva strategia de detectie sa surprinda aspecte particulare setului de observatii utilizat în definire. Modul concret în care se face acest lucru nu intereseaza momentan. La sfârsitul acestei faze trebuie însa sa se poata raspunde la întrebarea: este acceptabila aceasta strategie de detectie? Daca raspunsul este negativ trebuie sa se revina la faza 1 sau 3 în functie de situatia concreta care a condus la respingerea strategiei de detectie.

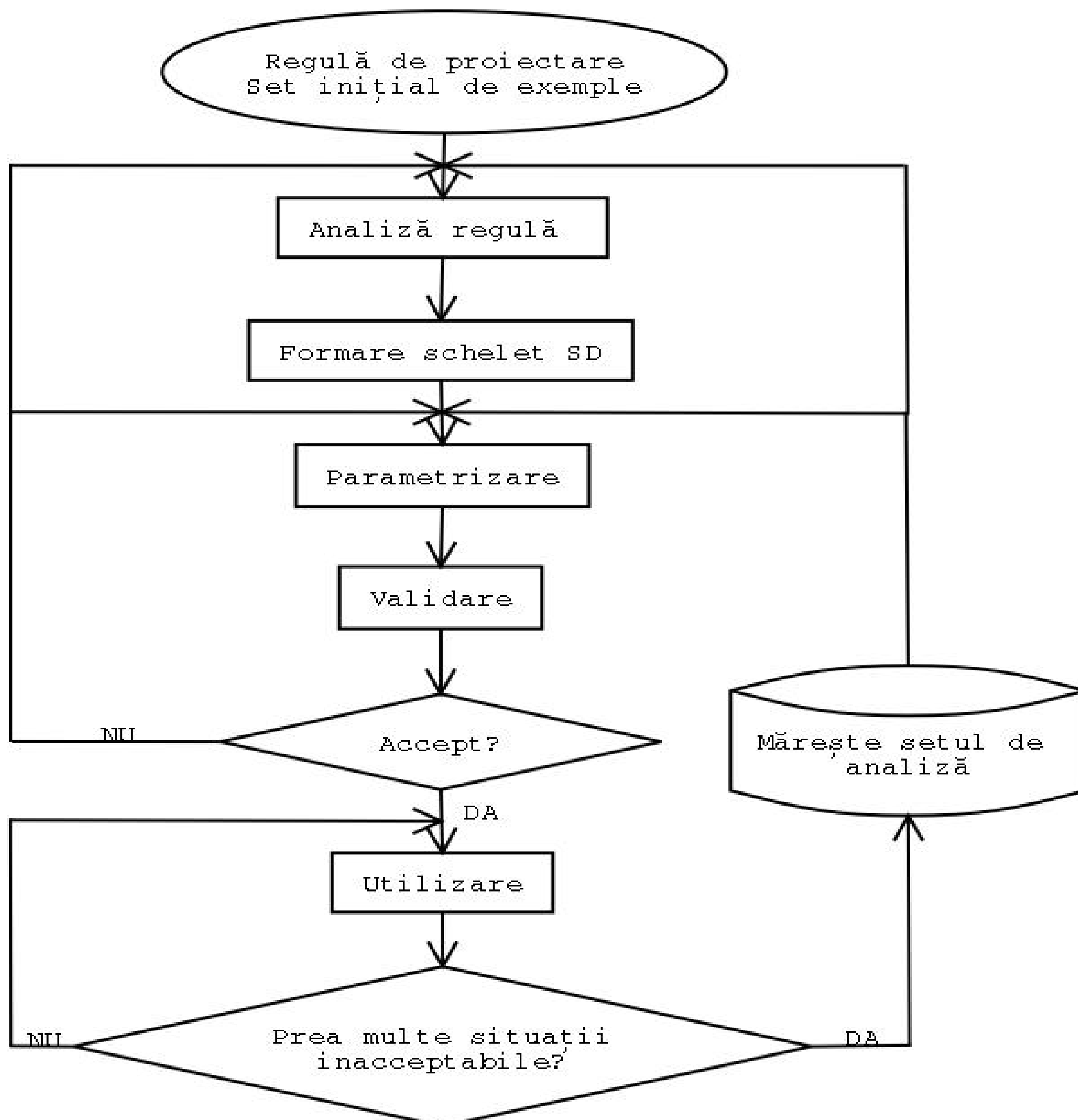

Figura 4.1. Definirea incrementala a strategiilor de detectie

5. **Utilizarea strategiei de detectie.** Daca strategia de detectie a fost declarata ca acceptabila, atunci ea poate fi utilizata pentru detectia automata a problemelor. Aceasta faza este si ea amintita în [14] lucru specificat de noi în capitolul anterior. În timp însa, pot aparea situatii în care ea nu detecteaza corect entitatile conforme cu regula ce sta la baza

respectivei strategii de detectie: ea detecteaza entitati ce nu ar trebui detectate si / sau nu detecteaza entitati ce ar trebui detectate. În functie de gravitatea acestor situatii se poate hotarî ca respectiva strategie nu mai este acceptabila. În raport cu situatia concreta care a condus la respingerea strategiei se revine la faza 1 sau 3. Odata cu aceasta revenire se mareste si setul de analiza care trebuie sa includa si exemplele concrete cu care strategia de detectie nu este consistenta. Lipsa de consistenta este foarte probabil datorata unor aspecte ale carentei de proiectare care nu au fost surprinse de setul initial analizat.

Fazele 1, 2 ,3 si 4 constituie asa numita operatie de analiza a problemei. Aici se desfasoara activitatea de definire propriu-zisa a strategiei de detectie, redeclansata ori de câte ori e nevoie. Principial vorbind, întreaga operatie ar putea fi complet automatizata daca ne gândim la mecanismul de învatare automata bazat pe arbori de identificare descris în [5,28]. Totusi, factorul uman este foarte important în anumite puncte ale procesului descris anterior: în cadrul validarii factorul uman poate decide ca anumite compromisuri sunt acceptabile iar altele nu; el poate hotarî mai bine relevanta unei metrici într-un anumit context; tot operatorul uman implicat în procesul de reengineering poate decide ca o anumita strategie de detectie nu mai este acceptabila.

Dupa cum am mai precizat, aceasta lucrare se va concentra doar asupra operatiei de parametrizare deoarece momentan ea este cea mai controversata. Unii ar spune ca tratarea exclusiva a parametrizarii nu este suficienta pentru o buna rafinare a strategiilor de detectie deoarece între ea si fazele premergatoare ei ar putea exista o strânsa legatura. Totusi, rafinarea strategiilor de detectie este o operatie foarte complexa, iar aceasta lucrare îsi propune sa deschida drumul în aceasta directie.

4.2 Atacarea problemei

Dupa cum reiese din discutia anterioara, definirea unei strategii de detectie trebuie sa conduca la obtinerea unei strategii despre care sa putem spune la momentul respectiv ca este probabil aproximativ corecta. Lucrarea de fata se concentreaza doar asupra parametrizarii, deci vom considera ca scheletul strategiei de detectie este cel mai corect posibil la acest moment. Ceea ce trebuie sa facem noi consta în stabilirea acelor parametrii pentru mecanismul de filtrare care sa asigure consistenta strategiei de detectie cu un numar cât mai mare de observatii, de exemple pozitive si negative. În acelasi timp, va trebui sa analizam comportamentul strategiei de detectie obtinute peste alte exemple pentru a putea trage în final o concluzie referitoare la gradul ei de eficacitate.

Ideea noastra este de a construi un program, o masina care sa fie capabila sa analizeze un numar mare de exemple si sa ne ajute în stabilirea celor mai buni parametrii la momentul respectiv. Datele necesare sunt: un schelet de strategie de detectie si un set de exemple pozitive si negative corespunzatoare problemei de proiectare detectate de strategia de detectie ai carei parametrii trebuie reglati. În acest context, vom introduce conceptul de masina de tuning a strategiilor de detectie.

4.3 Masina de tuning a strategiilor de detectie

4.3.1 Definire

Definitie. Masina de tuning a strategiilor de detectie este un mecanism care ajuta în luarea deciziei de acceptare a unei strategii de detectie prin reglarea automata a parametrilor acesteia în asa fel încât, strategia rezultata sa fie consistenta cu un numar cât de mare posibil de exemple pozitive si negative corespunzatoare regulii cuantificate de respectiva strategie de detectie.

Pentru a elimina posibilitatea unei întelegeri inadecvate a conceptului, vom furniza o serie de explicatii suplimentare legate de definitia de mai sus. În primul rând trebuie, sa specificam exact, ce întelegem prin exemplu pozitiv si exemplu negativ în acest context.

Definitie. Prin exemplu pozitiv corespunzator unei reguli se întelege o entitate de design conforma cu respectiva regula.

Definitie. Prin exemplu negativ corespunzator unei reguli se întelege o entitate de design, care nu este conforma cu respectiva regula.

În continuare vom discuta anumite aspecte ale definitiei.

- "…ajuta la luarea deciziei de acceptare…". Masina de tuning indica faptul ca o anumita strategie de detectie este mai buna decât alta deoarece parametrii primea asigura consistenta ei cu un numar mai mare de exemple pozitive si negative. Este sarcina operatorului uman de a lua decizia de acceptare pe baza informatiilor furnizate de masina.

- "…consistenta cu un numar cât de mare posibil de exemple pozitive si negative…". O strategie de detectie e consistenta cu un exemplu pozitiv daca si numai daca respectiva entitate de design este suspectata de strategia de detectie ca fiind conforma cu regula de proiectare cuantificata de respectiva strategie. În mod analog, o strategie de detectie e consistenta cu un exemplu negativ daca si numai daca respectiva entitate de design nu este suspectata de strategia de detectie ca fiind conforma cu regula de proiectare cuantificata de respectiva strategie.

4.3.2 Meta - arhitectura

În aceasta sectiune vom defini principalele elemente si notiuni implicate în descrierea completa a unei masini de tuning a strategiilor de detectie. În esenta, vom descrie într-o maniera abstracta independenta de implementare, principalele componente ale masinii, componente pe care orice implementare completa trebuie sa le aiba. Evident, aspectul unei anumite componente poate diferi de la o masina reala la alta, depinzând de contextul de implementare, dar în esenta este vorba despre una si aceeasi componenta. În acelasi timp, vom descrie relatiile dintre aceste componente si modul de functionare principial al masinii. Toate aceste elemente constituie meta-

arhitectura masinii de tuning. Prezentarea se bazeaza pe [24] aducându-se însa o serie de îmbunatatiri.

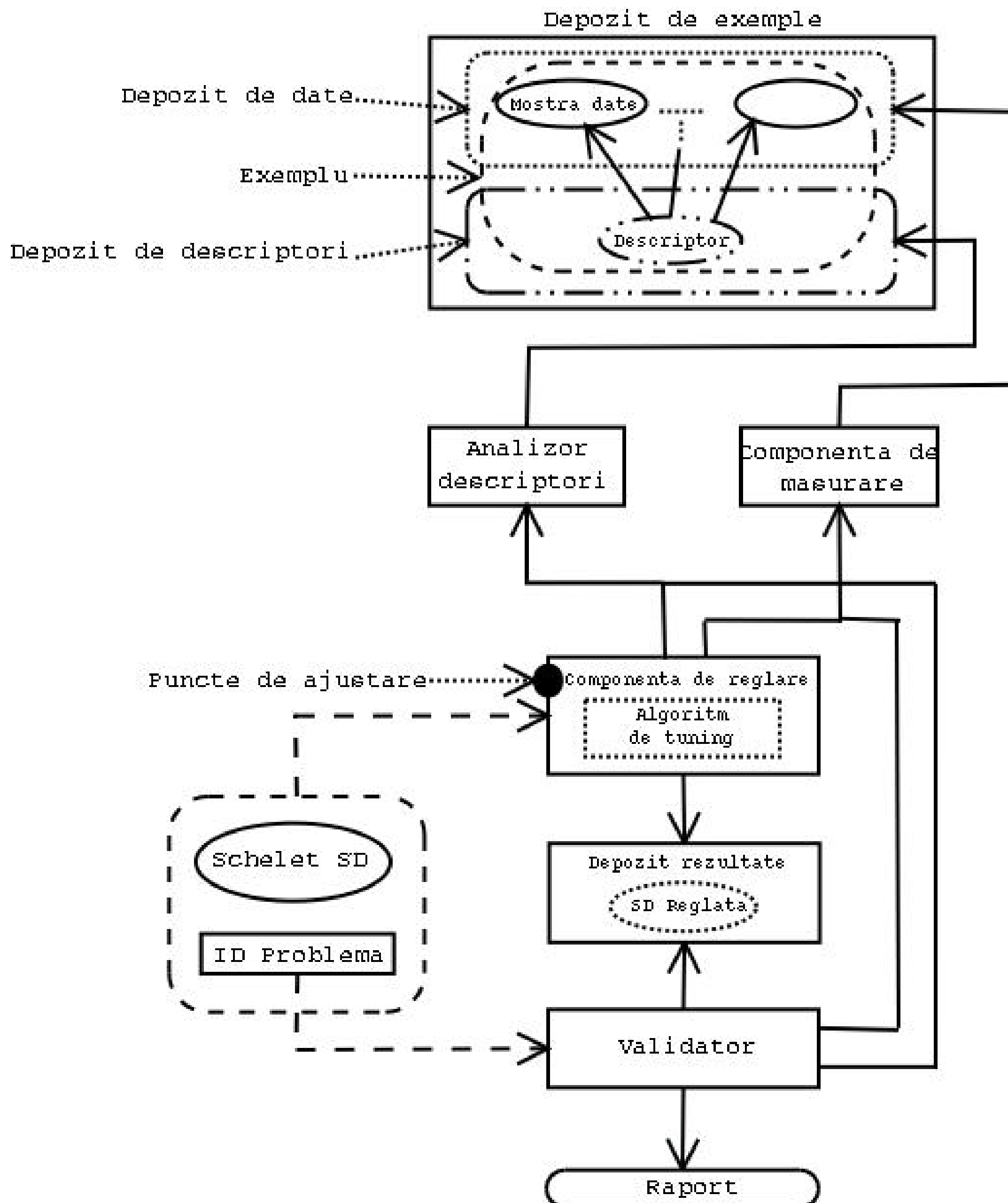

Figura 4.2. Meta-arhitectura masinii de tuning

4.3.2.1 Depozitul de exemple

Definitie. Depozitul de exemple ("Examples Repository") este o componenta a masinii de tuning responsabila cu toate operatiile necesare gestionarii setului de exemple.

Aceasta componenta este compusa la rândul ei din doua alte subcomponente: depozitul de date si depozitul de descriptori. Înainte de a explica rolul acestora va trebui sa introducem doua noi notiuni.

Definitie. O mostra de date ("Data sample") este o entitate indivizibila, complet ori incomplet specificata, care singura ori împreuna cu alte mostre de date, poate reproduce complet un exemplu.

Definitie. Un descriptor ("Descriptor") este o entitate care descrie complet un exemplu particular.

Pe baza acestor definitii vom descrie în cele ce urmeaza cele doua subcomponente ale depozitului.

Definitie. Depozitul de date ("Data Repository") este o subcomponenta a depozitului de exemple responsabila cu toate operatiile necesare gestionarii mostrelor de date.

Definitie. Depozitul de descriptori ("Descriptors Repository") este o subcomponenta a depozitului de exemple responsabila cu toate operatiile necesare gestionarii descriptorilor.

Dupa cum vedem în figura de mai sus, un exemplu este reprezentat în depozit printr-un descriptor si una sau mai multe mostre de date. O observatie importanta este aceea ca o aceeasi mostra de date poate fi utilizata în specificarea mai multor exemple distincte, putându-se astfel reduce dimensiunea depozitului de date. Un descriptor trebuie sa contina: un identificator de problema (ID problema) care sa identifice în mod unic carenta de proiectare careia i se adreseaza respectivul exemplu; informatii care sa arate daca exemplul asociat lui este negativ sau este pozitiv; identificatorii tuturor mostrelor de date ce vor recompune exemplul propriu-zis, fragmentul de design utilizat ca exemplu; o informatie din care sa rezulte daca respectivul exemplu va fi utilizat sau nu pentru parametrizarea propriu-zisa (vom întelege mai târziu de ce este necesar acest lucru). Identificatorul de problema va fi utilizat la identificarea exemplelor ce vor fi analizate de masina în vederea reglarii parametrilor unei strategii de detectie ce vizeaza respectiva carenta ori problema de proiectare.

Pentru exemplificare, sa consideram acum o masina reala care regleaza parametrii unei strategii de detectie ce identifica o anume carenta de proiectare. O mostra de date poate fi în aceasta situatie o clasa. Depozitul de date va gestiona fisiere, fiecare continând o clasa, mai exact codul sau. O clasa este acceptata si daca nu este complet specificata. Singura cerinta pentru a putea fi considerata o mostra de date corecta este sa putem calcula corect toate metricile software necesare la parametrizarea strategiei de detectie ce vizeaza carenta de proiectare exemplificata prin clasa respectiva. Cum clasa nu trebuie sa fie complet specificata, este posibila o noua reducere a dimensiunilor depozitului de date. Sa nu uitam ca trebuie sa lucram cu multe exemple, deci dimensiunea depozitului în ansamblu este critica. Descriptorul ar putea fi si el un fisier care contine toate informatiile necesare în descrierea exemplului. O mostra de date va fi identificata prin numele si calea fisierului ce contine codul clasei.

4.3.2.2 Analizorul de descriptori

Definitie. Analizorul de descriptori ("Descriptors Analyzer") este o componenta a masinii de tuning responsabila cu verificarea lexicala, sintactica si semantica a descriptorilor tuturor exemplelor compunând un set ce va fi utilizat la parametrizarea

ori validarea unei anumite strategii de detectie si translatând informatiile continute de acestia într-o forma recunoscuta de componentele corespunzatoare.

Complexitatea acestei componente depinde puternic de limbajul de specificare al descriptorului. Daca avem un descriptor simplu care contine de exemplu doar o enumerare a identificatorului de problema, tipului exemplului si identificatorilor mostrelor de date, analizorul va fi foarte simplu. Complexitatea lui va creste însa daca descriptorul va fi un fisier XML care mai contine si alte informatii necesare în contextul unei implementari particulare. Modul de specificare al descriptorului este stabilit în momentul implementarii unei masini reale. Este posibil ca descriptorul sa trebuiasca sa contina diverse alte informatii specifice implementarii. Aceste informatii nu depind exclusiv de aspecte legate de constructia masinii, ci pot depinde si de domeniul de activitate concret în cadrul caruia functioneaza masina si în care strategiile de detectie sunt utilizate ca tehnica de identificare si localizare a problemelor.

Analizorul de descriptori va fi utilizat de componenta de reglare în vederea accesului la setul de exemple utilizat în reglarea parametrilor unei strategii de detectie.

4.3.2.3 Componenta de masurare

Definitie. Componenta de masurare ("Metrics") este un modul al masinii de tuning capabil sa masoare orice atribut necesar asociat unei entitati dintr-un exemplu concret.

Pentru exemplificare, vom considera o strategie de detectie utilizata în identificarea unei carente de proiectare. Aceasta componenta trebuie sa fie capabila sa calculeze metricile software utilizate în definirea respectivei strategii, peste mostrele de date (clase) ce reproduc un exemplu asociat respectivei carente de proiectare.

4.3.2.4 Componenta de reglare

Definitie. Componenta de reglare ("Tuner") este un modul al masinii de tuning ce regleaza corespunzator parametrii unei strategii de detectie pe baza unui algoritm de reglare (algoritm de tuning).

Dupa cum reiese si din figura 4.2 elementul principal al tuner-ului este reprezentat de algoritmul de reglare. Acesta poate utiliza pentru realizarea efectiva a parametrizarii metode statistice, metode specifice inteligentei artificiale sau alte tehnici. Evident, calitatea parametrizarii depinde de acest algoritm de reglare.

Definitie. Un punct de ajustare ("Adjusting point") reprezinta o interfata utilizator-masina prin care operatorul poate influenta procesul de reglare.

Componenta de reglare poate avea nici unul, unul ori mai multe astfel de puncte de ajustare în functie de algoritmul de reglare si de modalitatile de interactiune permise de el. Totusi, suntem de parere ca orice implementare a masinii trebuie sa aiba cel putin un astfel de punct de ajustare. Procesul de reglare se bazeaza pe un numar mare de exemple. Masina va analiza un numar mare de exemple pentru a gasi cei mai buni parametrii. Este însa posibil ca strategia obtinuta sa nu fie consistenta chiar cu întregul set de exemple utilizat la parametrizarea sa fie datorita faptului ca

algoritmul de reglare impune constrângeri prea severe, fie ca pur si simplu nu se poate acest lucru. În aceasta situatie vom spune ca au aparut situatii fals-negative ("false-negatives") ori fals-pozitive ("false-pozitives"). Vom defini aceste concepte pe baza definitiilor din [28].

Definitie. Un exemplu este un fals-negativ pentru o strategie de detectie, daca ea identifica respectivul exemplu ca negativ dar în realitate el fiind pozitiv.

Definitie. Un exemplu este un fals-pozitiv pentru o strategie de detectie, daca ea identifica respectivul exemplu ca pozitiv dar în realitate el fiind negativ.

În cadrul acestor definitii trebuie sa se înteleaga ca ne referim la exemple corespunzatoare regulii de proiectare cuantificata de strategia de detectie care ar trebui sa fie consistenta cu respectivele exemple. În esenta, un fals-negativ reprezinta situatia în care o entitate de design afectata de o carenta de proiectare nu este suspectata de strategia de detectie dedicata identificarii respectivei carente. În mod asemanator, un fals-pozitiv reprezinta situatia în care o entitate de design ce nu este afectata de o carenta de proiectare este suspectata de strategia de detectie dedicata identificarii respectivei carente. Evident, ideal ar fi sa nu avem astfel de situatii. În practica mai mult ca sigur ca vom întâlni astfel de situatii. Cum în contextul detectiei carentelor de proiectare fals-negativ-urile sunt mult mai grave decât fals-pozitiv-urile trebuie sa putem controla în procesul de parametrizare numarul celor dintâi. Evident, pot exista si situatii în care numarul de fals-pozitiv-uri nu este acceptabil. Cu alte cuvinte, este necesara o modalitate de echilibrare convenabila a acestor doua situatii.

Definitie. Balansul ("Balance") este un punct de ajustare a componentei de reglare ce poate fi utilizat de operatorul masinii de tuning în vederea echilibrarii convenabile a numarului de situatii fals-negativ si fals-pozitiv.

Dupa cum reiese si din figura 4.2 componenta de reglare primeste la intrare scheletul strategiei de detectie ce trebuie reglata.

Definitie. Scheletul unei strategii de detectie reprezinta expresia respectivei strategii mai putin parametrii asociati mecanismului de filtrare.

Este probabil ca aceasta componenta sa contina la rândul ei o subcomponenta responsabila cu analiza respectivului schelet. Totusi, aceasta componenta desfasoara o activitate irelevanta din punctul de vedere a procesului de parametrizare în sine motiv pentru care aceasta subcomponenta nu este inclusa în meta-arhitectura masinii.

4.3.2.5 Depozitul de rezultate

Înainte de a defini aceasta componenta trebuie sa introducem a noua definitie.

Definitie. Spunem despre o strategie de detectie ca este reglata daca ea a fost parametrizata de o masina de tuning.

Definitie. Depozitul de rezultate ("Results Repository") este o componenta a masinii de tuning responsabila cu gestiunea unei istorii a strategiilor de detectie reglate de respectiva masina.

În forma sa cea mai simpla, depozitul de rezultate poate memora doar ultima strategie de detectie obtinuta de masina. Mentinând mai multe strategii de detectie reglate vom avea posibilitatea sa analizam comparativ performantele unei strategii de detectie în raport cu o alta (evident ca o astfel de comparatie are sens doar daca strategiile vizeaza o aceeasi carenta de proiectare).

4.3.2.6 Validatorul

Definitie. Validatorul este o componenta a masinii de tuning responsabila de compararea performantelor tuturor strategiilor de detectie reglate care adreseaza aceeasi problema si care sunt memorate în depozitul de rezultate, în vederea indicarii celei mai bune strategii la momentul respectiv.

Aceasta componenta este foarte importanta din punctul de vedere al operatorului, pentru ca pe baza analizei comparative el trebuie sa decida daca si care strategie de detectie reglata este acceptabila. Multe teste de performanta pot fi imaginate. Spre exemplu, se va putea compara performantele a doua strategii de detectie construite pe acelasi schelet dar parametrizate utilizând seturi de exemple diferite. Prima strategie este reglata peste un set initial, iar a doua este reglata dupa ce s-au adaugat câteva exemple noi. Putem vedea astfel daca setul initial surprinde corect toate particularitatile problemei. Pe de alta parte, am putea compara performantele a doua strategii de detectie construite pe schelete diferite dar parametrizate utilizând acelasi set. Vom putea astfel compara relevanta unei metrici fata de alta metrica în contextul detectiei unei anumite carente de proiectare. Evident, vom putea analiza si performantele unei strategii de detectie reglate, peste un set de exemple care nu a fost utilizate la parametrizarea ei, dar care exemplifica si ele problema de proiectare vizata de respectiva strategie de detectie. Un asemenea test este absolut necesar datorita pericolului reprezentat de fenomenul de overfitting [28]. În esenta, acesta consta în stabilirea unor parametrii irelevanti datorita unei particularitati nesemnificative a setului de exemple utilizat în reglare. Cross-validarea [28] este o tehnica utila în identificarea unei strategii de detectie afectata de fenomenul de overfitting a carei idee de baza consta tocmai în evaluarea performantelor strategiei de detectie peste un set de date ce nu a fost utilizat la parametrizarea ei.

4.3.2.7 Functionarea masinii

Primul lucru care trebuie facut înainte de a începe procesul de parametrizare a unei strategii de detectie este evident construirea unui set de exemple corespunzatoare respectivei strategii. Pe baza regulii de proiectare cuantificata de aceasta strategie de detectie vom identifica manual entitati de design conforme cu ea (exemple pozitive) precum si entitati de design ce nu sunt conforme cu ea (exemple negative). Fiecare exemplu va trebui introdus în depozit-ul masinii de tuning. Pentru acesta, fiecare fragment de design va fi descompus în mostre de date ce vor fi stocate în depozitul de date. În continuare vom scrie descriptorul respectivului exemplu ce va contine identificatorul problemei de proiectare careia i se adreseaza exemplul, tipul exemplului, identificatorii mostrelor de date asociate exemplului, iar daca utilizam cross-validarea va trebui sa specificam si daca respectivul exemplu se va include în setul de parametrizare sau în cel de validare. Se procedeaza asemanator pentru toate exemplele ce se introduc în depozit. Când consideram ca avem destule exemple se poate începe parametrizarea.

Scheletul strategiei de detectie si identificatorul problemei de proiectare vizata de strategia de detectie construita pe respectivul schelet sunt intrarile componentei de reglare. Ea va analiza scheletul pentru a vedea ce metrici trebuie calculate si ce filtre de date trebuie parametrizate. Apoi, ea va cere analizorului de descriptori sa gaseasca în depozitul lor descriptorii a caror identificator de problema este egal cu cel furnizat de operator. Cu alte cuvinte, îi cere sa identifice exemplele necesare parametrizarii. Analizorul va cauta acesti descriptori si va translata informatiile continute de ei într-o forma recunoscuta de componenta de reglare. Pentru fiecare exemplu componenta de reglare va cere componentei de masurare sa calculeze toate metricile necesare. Aceasta va accesa în acest scop depozitul de date. Toate rezultatele masuratorilor vor fi ulterior utilizate de algoritmul de tuning pentru stabilirea parametrilor strategiei de detectie. Daca este necesar, componenta de reglare va putea accesa depozitul de rezultate pentru a vedea modul de parametrizare al altor strategii de detectie reglate ce vizeaza aceeasi problema de proiectare ca si strategia aflata în curs de parametrizare. Când procesul de parametrizare este terminat noua strategie de detectie obtinuta prin amplasarea parametrilor corespunzatori în schelet este plasata în depozitul de rezultate.

În continuare intra în functiune validatorul. Activarea sa va putea fi facuta fie de operator fie de componenta de reglare. Oricum, acest lucru nu este important în contextul meta-arhitecturii. Validatorul va avea ca intrare un identificator de problema care va identifica toate strategiile de detectie din depozitul de rezultate care se adreseaza aceleasi probleme de proiectare si care vor fi testate în continuare. Acelasi identificator de problema va fi utilizat pentru identificarea setului de exemple utilizate în validare. Componenta de masurare si analizorul de descriptori sunt utilizati de validator într-o maniera asemanatoare componentei de reglare. În continuare fiecare strategie de detectie selectata din depozitul de rezultate va fi verificata din punctul de vedere al consistentei în raport cu fiecare exemplu din setul de validare. În final vom sti pentru fiecare strategie în parte câte situatii de inconsistenta prezinta (numar de fals-pozitiv-uri si numar de fals-negativ-uri). Toate informatiile vor fi scrise într-un raport de parametrizare. Pe baza lui operatorul va putea selecta pentru utilizare o anumita strategie de detectie.

Totusi, vor putea apare situatii în care nici o strategie nu este acceptabila. De exemplu, toate ar putea prezenta prea multe fals-negativ-uri. Aceasta situatie va apare în principiu când scheletul strategiei de detectie nu captureaza integral chiar toate aspectele carentei de proiectare vizate de el. Daca dorim sa utilizam acelasi schelet de strategie, dar vrem sa reducem si numarul de situatii fals-negativ vom relua parametrizarea dupa ce în prealabil vom fi modificat constrângerile impuse de algoritm. Pentru a realiza acest lucru operatorul actioneaza la nivelul punctului de ajustare balance. Evident, aceasta maniera de reducere a fals-negativ-urilor prezinta si dezavantaje: numarul de situatii fals-pozitive va creste. Oricum, dupa cum am mai spus aceasta situatie este mai putin daunatoare: preferam sa fie suspectate entitati de design “sanatoase” decât sa nu fie suspectate entitatile “bolnave”. Evident, cea mai buna modalitate de reducere a numarului de fals-negativ-uri consta în identificarea aspectul problemei de proiectare care nu este surprins de schelet si modificarea corespunzatoare a acestuia.

4.3.3 Considerente de implementare ale algoritmului de tuning

Elementul esential al masinii de tuning a strategiilor de detectie îl constituie evident algoritmul de reglare. În cele ce urmeaza, vom discuta câteva alternative posibile referitoare la acest element.

În general, orice task abstract ce trebuie îndeplinit, poate fi privit ca rezolvarea unei probleme. La rândul ei, solutionarea problemei poate fi perceputa ca o cautare într-un spatiu de solutii potentiale. Deoarece întotdeauna se cauta "cea mai buna solutie", acest task poate fi privit ca un proces de optimizare. Pentru multe probleme practice, singura cale sigura de gasire a unei solutii optime consta în cautarea ei în întregul set corespunzator tuturor solutiilor posibile. O astfel de cautare exaustiva consta în explorarea întregului "spatiu parametric" al problemei. Totusi, în multe situatii, acest spatiu parametric este atât de mare încât doar o mica fractiune a sa poate fi explorata într-un interval de timp rezonabil. Întrebarea care se pune atunci este: cum putem organiza cautarea astfel încât sa existe o probabilitate ridicata de localizare a unei solutii aproximativ optime, într-un interval de timp rezonabil? Abordarea uzuala consta în rafinarea iterativa a unei solutii pâna când euristica de rafinare nu mai aduce nici o îmbunatatire. Aceasta abordare sta la baza algoritmilor cu îmbunatatire iterativa. Algoritmii genetici reprezinta o alta alternativa. Inspirati din evolutia biologica, ei "încruciseaza" solutii permitând doar "celor mai bune" sa "supravietuiasca" în generatiile urmatoare.

În contextul discutiei anterioare, operatia de parametrizare executata de o masina de tuning poate fi vazuta ca un proces de optimizare: masina cauta acei parametrii pentru filtrele de date care sa asigure consistenta strategiei rezultate cu un numar cât mai mare posibil de exemple negative si pozitive. Altfel spus, masina încearca sa minimizeze numarul de situatii fals-pozitiv si fals-negativ în raport cu setul de exemple dat. Explorarea întregului spatiu de solutii potentiale asociat unei strategii de detectie, care trebuie parametrizata, nu este posibila. O solutie potentiala reprezinta o combinatie de valori numerice corespunzatoare parametrilor. Spatiul de solutii potentiale reprezinta toate combinatiile posibile de astfel de valori, cu alte cuvinte, toate variantele de parametrizare posibile. Ordinul de marime pentru acest spatiu creste de la 10^6 pentru strategiile de detectie relativ simple, la cel putin 10^{10} pentru strategiile de detectie mai puternice (Lack of Bridge, Lack of Strategy [14]). Este foarte probabil ca în viitor, odata cu rafinarea strategiilor de detectie, aceste dimensiuni sa devina si mai mari. Este evident ca la asemenea dimensiuni nu putem explora întregul spatiu de solutii potentiale. În schimb, putem folosi tehnicile amintite mai sus pentru a localiza o solutie probabil aproximativ optima, adica un set de parametrii pentru care numarul de situatii fals-negativ si fals-pozitiv este aproape minim. În continuare vom prezenta pe scurt aceste tehnici, pentru ca în capitolul urmator sa discutam si alte aspecte legate de implementarea componentei de reglare.

4.3.3.1 Algoritmi de îmbunatatire iterativa

Ideea ce sta la baza principiului de functionare a acestei familii de algoritmi este de a porni de la o solutie potentiala oarecare si de a aduce modificari asupra ei în vederea îmbunatatirii calitatii sale [28]. Cea mai buna maniera de întelegere a algoritmilor de îmbunatatire iterativa consta în considerarea tuturor starilor (în acest context solutii potentiale) ca fiind amplasate pe o suprafata (un teren în relief). Înaltimea fiecarui punct de pe suprafata corespunde valorii pe care o ia o functie de

evaluare pentru starea asociata respectivului punct. Aceasta functie de evaluare trebuie sa reprezinte o masura a calitatii solutiilor. Ideea îmbunatatirii iterative consta în deplasarea pe aceasta suprafata pentru a identifica cele mai înalte puncte, care sunt asociate solutiilor optime. Trebuie sa amintim de la început ca acesti algoritmi urmaresc de obicei, în timpul executiei lor, doar starea curenta si nu privesc dincolo de vecinii imediati ai acestei stari.

Cautarea Hill-Climbing

Algoritmul hill-climbing este constituit dintr-o singura bucla de program. Tehnica se aplica unui singur punct (cel curent) din spatiul de cautare. În timpul unei singure iteratii, un nou punct este selectat din vecinatatea punctului curent. Daca noul punct furnizeaza o valoare mai buna pentru functia de evaluare decât punctul curent (mai mica pentru probleme de minimizare si mai mare pentru probleme de maximizare), noul punct devine punct curent. În caz contrar, un alt vecin este selectat si testat în raport cu punctul curent. Algoritmul se termina daca nici o alta îmbunatatire nu mai poate fi adusa.

Acest algoritm prezinta însa un dezavantaj important: exista o probabilitate ridicata de oprire a algoritmului într-un optim local. Aceasta înseamna ca algoritmul se opreste într-o stare cu o "înaltime" mai mica decât starea de "înaltime" maxima din spatiul starilor (în cazul problemelor de maximizare). Este evident ca, odata ce algoritmul atinge un optim local, executia se opreste chiar daca solutia ce se ofera este departe de a fi satisfacatoare, iar acest lucru este total inacceptabil.

Pentru a creste probabilitatea de succes, metoda hill-climbing se poate executa succesiv pentru un numar mare de puncte initiale distincte din spatiul de cautare. Aceasta tehnica se numeste Random Restart Hill – Climbing [28].

Simulated annealing

Aceasta metoda este asemanatoare tehnicii hill-climbing dar elimina dezavantajul amintit mai sus: solutia furnizata nu depinde de punctul de start si este de obicei apropiata de solutia optima [23]. Acest lucru este posibil datorita introducerii unei probabilitati de acceptare p (probabilitatea de înlocuire a punctului curent cu alt punct). Aceasta probabilitate este unitara daca noul punct selectat prezinta o valoare mai buna pentru functia de evaluare decât punctul curent si este mai mare ca zero în caz contrar. În cea de-a doua situatie, probabilitatea p este o functie de valoarea functiei de evaluare a noului punct selectat si a punctului curent analizat, precum si de un parametru de control aditional T numit "temperatura". În general, cu cât temperatura este mai mica, cu atât este mai mica probabilitatea de acceptare. În timpul executiei algoritmului, temperatura T a sistemului este scazuta treptat. Algoritmul se termina pentru o valoare suficient de mica a lui T astfel încât probabilitatea de acceptare p sa fie practic nula.

4.3.3.2 Algoritmi genetici

Natura prezinta un mod robust de dezvoltare a organismelor de succes. Acele organisme, care nu sunt adaptate la mediul în care traiesc, mor, în timp ce cele adaptate supravietuiesc si se reproduc. Descendentii sunt similari cu parintii lor, deci membrii fiecarei noi generatii de organisme sunt similari cu membrii adaptati din generatia anterioara. Daca mediul se modifica încet, speciile pot evolua gradual în

paralel cu el. Daca însa o schimbare brusca apare în mediu este foarte probabil ca speciile adaptate la mediul initial sa dispara. Ocazional, apar mutatii aleatoare, si desi multe dintre acestea produc moartea rapida a individului mutant, exista situatii în care anumite mutatii conduc obtinerea unor specii de succes.

Acest mecanism natural este în prezent utilizat si în domeniul stiintei calculatoarelor. Algoritmii genetici utilizeaza un vocabular împrumutat din genetica. Astfel, se vorbeste despre indivizi care formeaza o populatie. Un individ se mai numeste si cromozom (desi o celula din orice organism prezinta mai multi cromozomi, în contextul algoritmilor genetici un individ este identificat cu un cromozom). Fiecare cromozom este compus dintru-un numar de unitati numite gene sau caractere, aranjate într-o succesiune liniara (de aceea un cromozom se mai numeste string). Fiecare gena controleaza mostenirea a cel putin unei caracteristici de la parinte la descendentii sai. Genele responsabile de anumite caracteristici sunt localizate în pozitii precise din cadrul cromozomului. Aceste pozitii se numesc loci, iar în contextul algoritmilor genetici aceste gene se localizeaza prin pozitia lor în cromozom (string position).

În general, un individ reprezinta o solutie potentiala a problemei care trebuie rezolvata. Un proces evolutiv ce se executa peste o populatie de cromozomi corespunde unei cautari într-un spatiu de solutii potentiale. Algoritmii genetici sunt o clasa de metode de cautare independente de domeniul problemei ce trebuie rezolvata, care balanseaza într-o maniera excelenta doua cerinte fundamentale ce trebuie avute în vedere la cautarea într-un spatiu de solutii: exploatarea celor mai bune solutii de la momentul respectiv si explorarea spatiului de solutii potentiale. Metode hill-climbing este un exemplu de strategie care exploateaza cea mai buna solutie în vederea aduceri unor îmbunatatiri calitative asupra ei. Pe de alta parte metoda neglijeaza explorarea spatiului de solutii potentiale. Cautarea aleatoare este un exemplu de strategie care exploreaza spatiul de solutii potentiale, dar care neglijeaza exploatarea celor mai promitatoare regiuni din acest spatiu. Algoritmii genetici apartin clasei algoritmilor probabilistici, dar sunt diferiti de algoritmii aleatori pentru ca utilizeaza simultan elemente ce apartin cautarilor directe si stohastice. Din aceasta cauza acesti algoritmi sunt mai robusti decât alte metode de cautare directa.

O alta proprietate importanta a algoritmilor genetici consta în mentinerea unei populatii de solutii potentiale, spre deosebire de alte metode de cautare care proceseaza la un moment dat doar un singur punct din spatiul de cautare. Cu alte cuvinte, algoritmii genetici executa o cautare multi-directionala simultana prin mentinerea unei populatii de solutii potentiale si încurajeaza formarea si schimbul de informatii între aceste directii.

Principiul de functionare

În forma sa cea mai simpla, o cautare genetica functioneaza dupa cum vom prezenta în continuare [6,23,28]. Într-o abordare clasica a algoritmilor genetici, un individ (o solutie) este reprezentat sub forma unui sir (string) de simboluri peste un alfabet finit. Fiecare element al sirului se numeste gena. Uzual se foloseste alfabetul binar format din simbolurile 0 si 1, vorbindu-se în aceasta situatie de o reprezentare sub forma de sir binar. Se pot folosi si alte alfabete pentru reprezentare, unii autori vorbind în aceasta situatie despre programare evolutiva. În consecinta, prima operatie care trebuie facuta consta în formularea problemei de rezolvat astfel încât orice solutie sa poata fi codificata într-un sir de cifre binare.

Fiecarui astfel de sir trebuie sa i se poata asocia o asa numita valoare de fitness, în functie de cât de apropiata este solutia codificata în sir de scopul urmarit. Aceasta valoare este deci o masura a calitatii unei solutii, si este calculata de asa numita functie de fitness. Aceasta functie depinde în principiu de problema de rezolvat, dar în orice caz, ea este o functie care primeste ca intrare un individ si furnizeaza o valoare reala ca iesire.

Odata stabilite modul de reprezentare al unui individ si modul de calcul al functiei de fitness, devine posibila executia algoritmului genetic. Se porneste de la o populatie initiala de indivizi. Acestia reprezinta solutii potentiale ale problemei care pot fi obtinute si în mod aleator. Aceasta populatie suporta în continuare un proces de evolutie: în fiecare generatie solutiile relativ “bune” se reproduc, în timp ce solutiile relativ “rele” mor. Distinctia între solutiile rele si cele bune se face cu ajutorul functiei de fitness.

În timpul iteratie t, un algoritm genetic mentine o populatie de solutii potentiale (cromozomi) $P(t) = (X_1^t, X_2^t, \ldots, X_n^t)$. Fiecare solutie X_i^t este evaluata de functia de fitness pentru a estima calitatea sa. În continuare, o noua populatie este formata prin aplicarea unui operator de selectie peste populatia initiala, selectându-se în principiu cei mai puternici indivizi. Exista mai multe strategii de selectie care pot fi urmate. În principiu, strategiile de selectie sunt aleatoare, cu probabilitatea de selectie a unui individ proportionala cu valoare sa de fitness. Astfel, daca functia de fitness arata ca o solutie X este de doua ori mai buna decât o solutie Y, atunci probabilitatea de selectie a lui X este de doua ori mai mare. În mod uzual, selectia se executa prin “clonare”, astfel încât un individ puternic sa poata fi selectat de mai multe ori.

Anumiti membrii din noua populatie obtinuta prin selectie vor suferi în continuare anumite modificari. Acestea se datoreaza operatiilor de crossover (încrucisare) si de mutare, care au drept scop obtinerea de solutii noi. Crossover-ul combina caracteristicile apartinând celor doi cromozomi parinti împerechiati aleator, formând doi cromozomi descendenti, similari parintilor, prin interschimbarea a doua segmente corespunzatoare din reprezentarea parintilor. Pentru exemplificare, sa consideram parintii reprezentati prin vectori de dimensiune cinci: (a, b, c, d, e) si (f, g, h, i, j). Daca punctul de încrucisare se stabileste dupa cea de-a doua gena (de obicei acest punct e stabilit tot aleator), atunci cromozomii rezultati vor fi: (a, b, h, i, j) si (f, g, c, d, e). Schimbul de informatie între diferite solutii potentiale constituie esenta operatorului de crossover.

Mutatia consta în alterarea uneia sau a mai multor gene apartinând unui anumit cromozom, printr-o modificare aleatoare ce apare cu o anumita probabilitate. Esenta operatorului de mutare consta în introducerea unei cantitati suplimentare de diversitate în cadrul populatiei. Asa cum se arata în [6], operatorul de mutare poate împiedica evolutia populatiei catre un optim local.

Crossover-ul si mutatia reprezinta asa numita faza de reproducere a indivizilor. La sfârsitul acestei faze, noua populatie este complet stabilita si algoritmul poate trece la iteratia urmatoare. În principiu, algoritmul se poate opri dupa ce populatia initiala a evoluat un anumit numar de generatii. Evident, alte criterii de oprire a algoritmului pot fi imaginate (ex. s-a gasit o solutie cu o valoare de fitness acceptabila). Cea mai buna solutie (individ) din ultima populatie creata reprezinta solutia determinata de algoritm.

Operatiile simple de selectie, crossover si mutare reprezinta esenta algoritmilor genetici. Metodele traditionale de localizare a unei solutii optime utilizeaza euristici ce exploreaza vecinatatea imediata a unei singure solutii probabile. Chiar daca la unele metode este posibil sa apara salturi catre alte zone apartinând

spatiului de cautare, aceste euristici prezinta totusi o probabilitate prea ridicata de determinare a unui optim local. Prin mentinerea unei perspective asupra mai multor regiuni ale spatiului de cautare, algoritmii genetici au mult mai multe sanse de localizare a optimului global. Ei pot determina optimul chiar si în situatii în care functia de fitness este discontinua, iregulara sau zgomotoasa. Din aceste motive, prototipul de masina de tuning a strategiilor de detectie prezentat în capitolul care urmeaza are ca algoritm de reglare un algoritm genetic.

Capitolul V

Implementare si evaluare. Prototipul LRG – DSTM

O implementare completa a conceptului de masina de tuning ce sa prezinte toate caracteristicile descrise în capitolul anterior ar fi total nefolositoare daca rezultatele obtinute nu ar fi utile. În consecinta, la implementare ne-am limitat doar la aspectele esentiale ale masini, aspecte necesare în speta executiei operatiei de parametrizare. În acelasi timp, în vederea evaluarii rezultatelor, a fost implementata si o forma de validare, asa cum se va vedea în continuare. S-a nascut astfel prototipul de masina de tuning LRG-DSTM (Loose Research Group – Detection Strategy Tuning Machine). Acest prototip va fi prezentat în capitolul de fata, împreuna cu o analiza a performantelor rezultatelor furnizate.

5.1 Implementarea conceptului

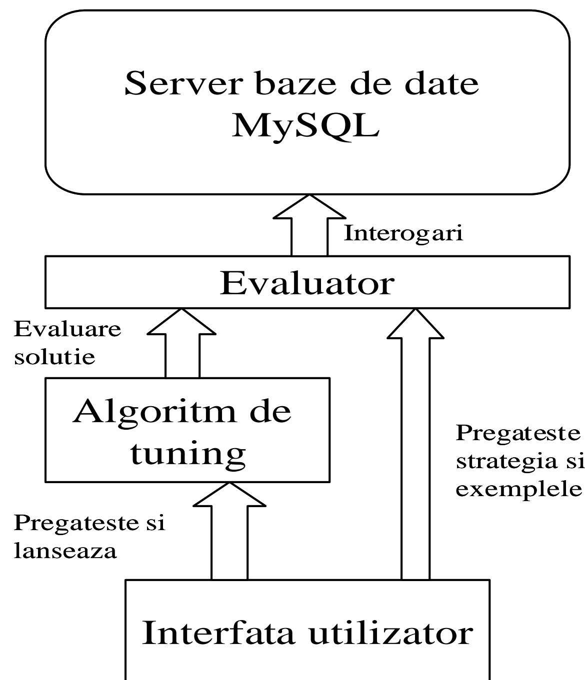

Figura 5.1 Structura prototipului LRG-DSTM

Structura prototipului de masina de tuning este prezentata în figura 5.1. Ea trebuie privita ca o "umbra" a meta-arhitecturii masinii de tuning descrisa în capitolul anterior. Anumite componente din meta-arhitectura au fost ignorate sau combinate la nivelul prototipului într-una sau mai multe componente. Acest lucru se datoreaza scopului urmarit de noi: dorim sa evaluam rezultatele operatiei de parametrizare a strategiilor de detectie executata de o implementare a conceptului de masina de tuning, pentru a determina eficacitatea abordarii problemei parametrilor. În cele ce urmeaza, vom prezenta principial componentele prototipului, insistând doar asupra aspectelor de esenta si mai putin asupra design-ului programului propriu-zis. Limbajul de programare utilizat a fost Java.

5.1.1 Serverul de baze de date MySQL

Serverul MySQL este responsabil de gestiunea bazei de date ce contine exemplele pozitive si negative de carente de proiectare. Cu alte cuvinte, serverul reprezinta depozitul de exemple descris în cadrul meta-arhitecturii masinii de tuning. În cele ce urmeaza vom prezenta modul de organizare a bazei de date.

În cadrul prototipului nostru, un exemplu de carenta de proiectare este compus dintr-o singura mostra de date si un descriptor. Descriptorul apare doar daca exemplul descris este unul pozitiv pentru respectiva carenta, exemplele negative ne-necesitând un descriptor. O mostra de date este o entitate de design (metoda, clasa sau pachet) si este reprezentata printr-o înregistrare din cadrul unei tabele dedicate. În esenta, o astfel de înregistrare contine un identificator al respectivei mostre, numele entitatii de design data ca exemplu, un identificator al sistemului din care provine si valorile diferitelor metrici software calculate peste entitatea respectiva. Exista trei tabele, care memoreaza mostrele de date, corespunzatoare celor trei tipuri de carente de proiectare prezentate în capitolul trei (mai general, corespunzatoare celor trei categorii de entitati de design): class_datasamples, method_datasamples si package_datasamples. În strânsa legatura cu acestea sunt tabelele systems si languages. Prima tabela contine informatii referitoare la sistemele din care provin mostrele de date: un identificator al sistemului, numele sistemului si un cod reprezentând limbajul de programare în care a fost scris programul. Aceste coduri sunt specificate în tabela languages. Toate aceste tabele compun asa numitul depozit de date.

Descriptorii sunt memorati si ei într-un numar de trei tabele corespunzatoare celor trei categorii de entitati de design: class_descriptors, method_descriptors, package_descriptors. Aceste tabele contin identificatorul mostrei de date asociate si codul carentei de proiectare ce afecteaza entitatea de design data ca exemplu pozitiv pentru respectiva carenta. Orice mostra de date, pentru care nu exista un descriptor care sa specifice ca respectiva mostra exemplifica o anumita carenta de proiectare particulara, va fi considerata implicit ca fiind un exemplu negativ pentru respectiva carenta. Codurile carentelor sunt descrise în tabela designflaws, iar identificatorii mostrelor de date specifica înregistrari din tabelele de mostre de date corespunzatoare (class_datasamples, method_datasamples respectiv package_datasamples). Toate aceste tabele reprezinta depozitul de descriptori.

Dupa cum am mai spus, o înregistrare din tabelele de mostre de date contine valorile unor metrici software calculate peste entitatea de design asociata mostrei de date. Prin urmare, calcularea unei metrici software pentru o anumita entitate de design se reduce la o simpla executie a unui query sql. Se poate spune ca serverul de baze de date îndeplineste, cel putin în parte, si sarcinile componentei de masurare din cadrul meta-arhitecturii masinii de tuning.

În contextul prototipului nostru, doua operatii asupra tabelelor descrise mai sus intereseaza în mod deosebit. O operatie de join între tabela de descriptori si tabela de mostre de date asociata, utilizând ca si câmp de join identificatorul de mostra de date, va produce ca rezultat o lista a tuturor mostrelor de date afectate de o carenta de proiectare oarecare. Observam faptul ca este posibil sa determinam toate mostrele de date afectate de o anumita carenta de proiectare, utilizând un simplu query sql. Spre exemplu, toate clasele afectate de carenta de proiectare DataClass pot fi determinate astfel:

```
SELECT t1.ClassId,… FROM class_datasamples t1 INNER JOIN
class_descriptors t2 USING(ClassId) WHERE t2.FlawCode = 4
```

În mod asemanator, se pot determina si mostrele de date ce compun exemple negative corespunzatoare unei anumite carente. O operatie corespunzatoare de left join între o tabela de mostre de date si o tabela a mostrelor ce compun exemple pozitive, corespunzatoare aceleasi carente (ce poate fi obtinuta ca mai sus), utilizând ca si câmp de join identificatorul mostrelor de date, va permite identificarea tuturor mostrelor de date ce nu sunt afectate de respectiva carenta de proiectare. Spre exemplu, determinarea tuturor claselor ce nu sunt afectate de carenta de proiectare DataClass se poate realiza astfel:

```
SELECT t1.ClassId,… FROM class_datasamples t1 LEFT JOIN
exemple_pozitive t2 USING(ClassId) WHERE t2.ClassId IS NULL
```

Exemple_pozitive trebuie privita ca o tabela temporara ce contine rezultatul primului query sql prezentat. Din modul de constructie a listelor de mostre ce reprezinta exemple pozitive si negative pentru o anumita carenta de proiectare, subliniem înca o data urmatorul aspect. Daca o anumita entitate de design (mostra de date) este afectata de o carenta de proiectare, ea va constitui un exemplu pozitiv pentru respectiva carenta, lucru specificat prin existenta unui descriptor corespunzator. Daca ne intereseaza toate entitatile afectate de aceasta carenta, entitatea de mai sus va fi inclusa în lista entitatilor asociate exemplelor pozitive. Daca ne intereseaza toate entitatile afectate de o alta carenta de proiectare, care nu afecteaza entitatea amintita mai sus, atunci se va considera implicit ca ea constituie un exemplu negativ relativ la respectiva carenta, si va fi inclusa în lista asociata exemplelor negative.

Trebuie remarcat faptul ca o anumita entitate de design poate fi afectata de mai multe carente de proiectare distincte. Se vor putea construi mai multe exemple pozitive, câte unul pentru fiecare carenta individuala. Fiecare exemplu pozitiv va avea asociat câte un descriptor corespunzator, toate referind aceeasi mostra de date dar având coduri de carenta diferite. Entitatea respectiva va fi inclusa în lista mostrelor ce compun exemple pozitive când intereseaza una dintre carentele de proiectare ce afecteaza entitatea noastra. În toate celelalte cazuri ea va fi considerata implicit ca reprezentând un exemplu negativ.

Modalitatile de determinare a mostrelor exemplelor negative si pozitive prezentate mai sus sunt importante pentru ca permit, dupa cum vom vedea imediat, determinarea foarte simpla a numarului de situatii fals-negative si fals-pozitive.

5.1.2 Evaluatorul

Aceasta componenta detine o sarcina foarte importanta: determinarea numarului de situatii fals-negative si fals-pozitive la aplicarea unei strategii de detectie având dat un anumit set de parametrii corespunzatori filtrelor de date. Aceasta informatie este necesara pentru calcularea calitatii unei solutii de parametrizare potentiale, asa cum va reiesi din discutia despre algoritmul de tuning. Pe de alta parte, cunoasterea numarului de situatii fals-negative si fals-pozitive este necesara în vederea validarii. Pentru a putea determina aceste informatii, componenta trebuie sa stie în primul rând cum se aplica o strategie de detectie si care este scheletul ei.

Pentru a determina daca o anumita entitate de design este sau nu detectata de o strategie de detectie particulara, nu este suficient sa se cunoasca valorile metricilor software calculate peste entitatea respectiva si urmarite de respectiva strategie de

detectie. Acest lucru se datoreaza filtrelor de date BottomValues si TopValues. Acestea necesita cunoasterea valorilor metricilor, care au asociate astfel de filtre, pentru toate entitatile de design similare din sistem. Principial vorbind, asta nu înseamna ca toate entitatile respective din sistem trebuie date ca exemple (de la caz la caz negative sau pozitive). Descriptorul ar putea contine informatiile necesare rezolvarii acestor situatii. În implementarea prototipului nostru, am preferat sa utilizam pe post de exemple toate entitatile de design similare dintr-un sistem. Spre exemplu, pentru reglarea strategiei de detectie DataClass, se introduc în tabelele de mostre de date (class_datasamples) informatiile corespunzatoare pentru toate clasele ce apar într-un sistem. DataClass-urile identificate în respectivul sistem sunt înregistrate ca exemple pozitive, printr-un descriptor în tabela class_descriptors. Clasele ce nu prezinta aceasta carenta de proiectare nu au un descriptor si vor fi considerate exemple negative când se va regla strategia de detectie DataClass. Evident, si ele ar putea avea un descriptor daca sunt afectate de alte carente de proiectare.

Deoarece toate entitatile de design similare (clase, metode, pachete) dintr-un sistem se gasesc în baza de date a masinii, este posibila aplicarea simpla a unei strategii de detectie. Evaluatorul procedeaza în urmatorul mod pentru identificarea numarului de situatii fals-negative si fals-pozitive la aplicarea unei strategii de detectie. În primul rând are loc o initializare:

- Se creeaza o tabela temporara cacheFn în care sunt aduse toate entitatile de design (mostre de date) care reprezinta un exemplu pozitiv de carenta de proiectare vizata de strategia de detectie ce se va aplica. Modul în care se determina mostrele de date ce compun exemple pozitive a fost prezentat în paragraful anterior. Informatia transferata consta în valorile metricilor software ce apar în scheletul strategiei de detectie aplicate, identificatorul mostrei de date si identificatorul sistemului din care provine.

- În mod asemanator se construieste tabela temporara cacheFp în care sunt aduse toate entitatile de design care reprezinta un exemplu negativ de carenta de proiectare vizata de strategia de detectie ce se va aplica.

- Pentru fiecare sistem în parte se realizeaza un clasament al entitatilor de design cei apartin, dupa valorile metricii care are asociat un filtru TopValues ori ButtomValues în scheletul strategiei ce se aplica. Daca exista mai multe astfel de metrici, se realizeaza un clasament dupa fiecare. Spre exemplu, în cazul lui ButtomValues, se sorteaza entitatile crescator dupa valoarea metricii ce-l are ca filtru pe ButtomValues. Toate entitatile ce prezinta pentru metrica respectiva o valoare egala cu valoarea minima sunt plasate pe locul 1. Toate entitatile cu valoarea metricii egala cu valoarea imediat mai mare decât cea minima sunt plasate pe locul 1 + Numar_clase_locul_1, s.a.m.d. În general, toate entitatile cu o valoare egala pentru metrica respectiva sunt amplasate pe locul Loc_Anterior_Ocupat + Numar_Entitati_Loc_Anterior. Aceste pozitii sunt memorate într-un câmp special dedicat acestui lucru, din tabelele temporare.

Toate aceste operatii sunt executate doar o singura data, la pregatirea componentei pentru a aplica o anumita strategie. Activitatile sunt efectuate în colaborare cu serverul de baze de date. În continuare determinarea numarului de

situatii fals-negative si fals-pozitive este triviala. Pentru fiecare sistem utilizat ca exemplu se interogheaza serverul în vederea determinarii numarului de entitati de design din respectivul sistem ce nu sunt detectate de strategia aplicata. Practic, se cere executia unei operatii de selectie (în combinatie cu functia agregat COUNT) peste tabela cacheFn, cu o clauza where ce traduce strategia de detectie. Evident, se selecteaza doar entitatile ce apartin sistemului analizat la momentul respectiv. Traducerea strategiei în limbajul sql este foarte simpla (inclusiv a filtrelor TopValues si ButtomValues având la dispozitie informatiile de la initializare) fiind practic vorba de interpretarea strategiei de detectie ca o functie logica (vezi capitolul 3). Însumarea rezultatelor pentru fiecare sistem în parte furnizeaza numarul de situatii fals-negative. În mod asemanator se determina si numarul de situatii fals-pozitive, doar ca se determina numarul de entitati din tabela cacheFp care sunt identificate de strategia aplicata.

La aplicarea unei strategii de detectie se evalueaza de fapt setul de parametrii pentru filtrele de date asociate strategiei. Un astfel de set reprezinta o solutie potentiala de parametrizare. Multe astfel de solutii vor fi evaluate în timpul procesului de tuning. Din acest motiv au fost create tabelele temporare. Pe lânga informatia suplimentara continuta (pozitia în clasamentul realizat dupa valoarea unei metrici) ele contin si valorile metricilor necesare evaluarii, calculate peste entitatile de design asociate mostrelor de date. Aceste valori sunt transferate din tabelele mostrelor de date, eliminând astfel necesitatea efectuarii unui join la fiecare evaluare.

În finalul discutiei despre aceasta componenta, mai precizam câteva aspecte. Exista posibilitatea configurarii acestei componente astfel încât sa foloseasca în procesul de evaluare doar anumite sisteme exemplu. Acest lucru este foarte important din punctul de vedere al validarii. Prin functiile sale aceasta componenta înglobeaza mai multe componente descrise la nivelul meta-arhitecturii masinii de tuning. Astfel, prin faptul ca pregateste setul de exemple utilizat la reglarea parametrilor unei strategii de detectie, componenta executa sarcinile analizorului de descriptori. Prin faptul ca evalueaza o solutie potentiala, un set de parametrii, ea poate fi vazuta si ca un validator. În fine, prin faptul ca ea cunoaste scheletul strategiei de detectie ce se va parametriza, componenta executa în parte si sarcini apartinând componentei de reglare. Evaluatorul poate determina si domeniul valoric al fiecarui parametru ce apare în cadrul unui schelet de strategie de detectie (prin interogarea serverului). Dupa cum vom vedea în paragraful urmator, va fi utila utilizarea tiparului de proiectare Builder [8] în vederea constructiei reprezentarii solutiei potentiale. La nivel de design, componenta este suficient de flexibila pentru a permite extinderea ei în vederea aplicarii oricarei strategii de detectie actuale.

5.1.3 Algoritmul de tuning

Algoritmul implementat în cadrul prototipului de masini de tuning al strategiilor de detectie este un algoritm genetic "clasic". Se porneste de la o populatie de cromozomi (solutii potentiale) care se initializeaza într-o maniera aleatoare. Reprezentarea unei solutii va fi prezentata mai târziu. Dimensiunea populatiei este un parametru al algoritmului specificat de catre operatorul masinii. Algoritmul se executa pentru un numar specificat de generatii, acesta fiind un alt parametru al algoritmului specificat de asemenea de catre operatorul masinii. În cadrul fiecarei iteratii, o noua generatie de solutii potentiale este creata pe baza operatiilor clasice asociate algoritmilor genetici.

Prima operatie consta în selectia celor mai "puternice" solutii, pe baza functiei de fitness. Dimensiunea populatiei este constanta de la o generatie la alta, iar un cromozom poate fi selectat de mai multe ori în vederea producerii urmatoarei generatii de solutii potentiale. Strategia concreta de selectie va fi prezentata mai târziu.

În continuare urmeaza operatia de crossover (încrucisarea). Fiecare cromozom, selectat în cadrul operatiei anterioare, poate fi ales, cu o anumita probabilitate, sa se împerecheze cu o alta solutie aleasa în acelasi mod. Aceasta probabilitate de crossover este un al treilea parametru al algoritmului specificat tot de catre utilizator. Doi cromozomi alesi consecutiv în aceasta maniera vor suferi operatia de încrucisare propriu-zisa. Strategia urmata de operatie va fi si ea prezentata mai târziu.

Ultima operatie este mutatia. Utilizatorul specifica un parametru numit probabilitate de mutare, ea reprezentând probabilitatea de alterare, într-o anumita maniera, asociata fiecarei gene din cadrul unui cromozom. Strategia de alterare a genei va fi prezentata mai târziu.

Reprezentarea solutiei

În contextul viziunii asupra operatiei de parametrizare a strategiilor de detectie prezentata în capitolul anterior, aceasta activitate poate fi privita ca o operatie de optimizare numerica: gasirea acelor numere, praguri valorice pentru filtrele de date din scheletul unei strategii de detectie, care sa permita strategiei de detectie rezultate sa faca cât mai bine distinctia între exemplele pozitive si negative avute la dispozitie. În consecinta, o solutie va contine toti parametrii necesari filtrelor de date din scheletul strategiei de detectie reglata. Se pune în continuare problema codificarii acestor numere în cadrul unui cromozom. Prototipul realizat permite doua modalitati de codificare: binara si reala. Înainte de a descrie cele doua moduri de reprezentare a solutiilor, trebuie sa amintim ca un parametru este specificat prin domeniul în care el ia valori, tipul sau (întreg sau real), iar în cazul parametrilor de tip real precizia sa (numarul de cifre zecimale semnificative de dupa virgula).

Reprezentarea binara este simpla: pentru fiecare parametru se rezerva în cadrul cromozomului numarul minim de biti necesari reprezentarii întregului domeniu valoric (împreuna cu precizia necesara în cazul parametrilor reali) asociat lui. Fara a omite aspecte legate de implementare, vom prezenta doar codificarea parametrilor reali, parametrii întregi codificându-se similar considerând precizia de reprezentare zero. Fie parametru x_i un parametru real cu domeniul $D_i = [a_i, b_i]$ si precizia p. Pentru a obtine o astfel de precizie domeniul trebuie divizat în $(b_i - a_i) \cdot 10^p$ parti egale. Fie m_i cel mai mic întreg care respecta relatia $(b_i - a_i) \cdot 10^p \leq 2^{m_i} - 1$. Acest m_i reprezinta numarul de biti necesari reprezentarii parametrului. Prin urmare, reprezentarea unei solutii potentiale consta dintr-un sir binar de lungime $m = \sum_{i=1} m_i$; m_1 sunt bitii corespunzatori primului parametru, urmatorii m_2 corespund celui de-al doilea parametru si asa mai departe. Conversia catre valoarea parametrului se poate face utilizând urmatoarea formula:

$$x_i = a_i + zecimal((bit_1...bit_{m_i})_2) \cdot \frac{b_i - a_i}{2^{m_i} - 1}$$

Aceasta reprezentare are avantajul ca modificarea unuia sau a mai multor biti (ex. prin operatorul de mutare) conduce la obtinerea unei solutii potentiale corecte,

neprovocând iesirea parametrului de care apartine bitul sau bitii din domeniul sau valoric. Nu vor fi deci necesare operatii suplimentare de corectie. Aceasta proprietate poate fi obtinuta si prin alte reprezentari. Reprezentarea binara prezinta însa unele dezavantaje [23]: performante scazute pentru reprezentari foarte lungi; operatia de decodificare este foarte costisitoare ca timp; reprezentarea binara este mai instabila decât reprezentarea în virgula flotanta (deviatia standard a rezultatelor obtinute de algoritm este mai mare în cazul reprezentarii binare), desi câteodata ea furnizeaza rezultate mai bune. Pentru a oferi mai multa flexibilitate operatorului masinii de tuning si pentru a putea face uz de operatori de mutatie si crossover mai sofisticati, a fost implementata si posibilitatea utilizarii reprezentarilor în virgula flotanta.

În reprezentarea reala, un cromozom este constituit dintr-un vector de elemente reale. Fiecare gena este responsabila pentru un anumit parametru. Daca acesta este real, atunci o gena memoreaza direct valoarea parametrului (în momentul decodificarii trebuie avut grija doar la precizie, pentru a elimina cifre nesemnificative). Daca parametru este întreg atunci domeniul sau este translatat corespunzator într-un domeniu inclus în intervalul [0, 1]. Operatia de decodificare va necesita niste operatii suplimentare foarte simple. În ceea ce priveste consistenta unui cromozom real dupa executia unei operatii de crossover sau mutare trebuie sa amintim ca nu exista probleme. Operatorii respectivi pot fi proiectati simplu în vederea pastrarii consistentei cromozomului (parametrii codificati nu-si parasesc domeniul). În acelasi timp se pot codifica în reprezentari scurte domenii valorice foarte mari, se micsoreaza considerabil timpul necesar decodificarilor si se pot utiliza precizii mari având totusi reprezentari scurte. Singurul dezavantaj, nesemnificativ în contextul nostru, consta în dependenta preciziei de masina pe care se executa algoritmul.

Functia de fitness

Functia de fitness trebuie sa masoare calitatea unei solutii potentiale. În cadrul prototipului nostru, functia de fitness utilizata denota abordarea pragmatica a problemei stabilirii parametrilor filtrelor de date: fiind data o solutie potentiala X, functia asociaza lui X o valoare reala direct proportionala cu numarul de situatii fals-negative si fals-pozitive aparute în urma aplicarii strategiei de detectie reglate peste setul de exemple dat, utilizându-se parametrii corespunzatori codificati în X. Formal, aceasta functie poate fi scrisa astfel:

$$f(X) = PFn \cdot Nr_Fn(X) + PFp \cdot Nr_Fp(X) + 1$$

unde: PFn reprezinta penalitatea aplicata pentru fiecare situatie fals-negativa produsa, PFp este penalitatea pentru fiecare situatie fals-pozitiv generata, Nr_Fn(X) reprezinta numarul de situatii fals-negative produse în momentul aplicarii strategiei de detectie aflate în curs de reglare utilizându-se parametrii corespunzatori codificati în X, iar Nr_Fp(X) reprezinta numarul de situatii fals-pozitive aparute în momentul aplicarii strategiei de detectie aflate în curs de reglare utilizându-se parametrii corespunzatori codificati în X. Ultimii doi membrii sunt calculati de evaluator. Tuturor acestor parametrii le corespund valori pozitive. Dupa cum se observa, o solutie potentiala este cu atât mai buna cu cât valoarea asociata ei de catre functia de mai sus este mai mica. În consecinta, parametrizarea este vazuta ca o problema de minimizare. Masina va încerca sa determine acei parametrii pentru care functia ia o valoare cât mai mica (ideal 1).

În principiu se pot utiliza si alte functii pentru evaluarea cromozomilor. Singura cerinta necesara, datorata unor considerente de implementare, este ca aceasta functie sa fie mai mare sau egala cu 1 oricare este X. Faptul ca o eventuala functie returneaza sau nu o valoare direct proportionala cu numarul de situatii fals-negative si fals-pozitive depinde de scopul urmarit în cadrul minimizarii.

Parametrii PFn si PFp sunt utilizati pentru implementarea punctului de ajustare balans. Pe de-o parte, înainte de executia parametrizarii, utilizatorul furnizeaza masinii asa numita amplitudine de penalitate. Ea reprezinta "pedeapsa" pe care o primeste un cromozom care conduce la aparitia unui singur fals-negativ si a unui singur fals-pozitiv. Rolul amplitudinii va fi înteles când vom vorbi despre strategia de selectie. Pe de alta parte, utilizatorul mai furnizeaza masinii un numar real din domeniul (0,1). Acest numar reprezinta balansul. Considerând ca acest numar este b parametrii PFn si PFp vor fi calculati astfel:

$$PFn = Amplitudine_Penalitate \cdot b$$
$$PFp = Amplitudine_Penalitate \cdot (1 - b)$$

Se poate observa ca pentru un balans de 0.5 o situatie fals-negativa si una fals-pozitiva sunt penalizate în acelasi mod. Pentru un balans de 0.6, o situatie fals-negativa este penalizata cu 60% din amplitudinea de penalitate, iar una fals-pozitiva cu doar 40%. Ca urmare, o solutie potentiala ce prezinta un singur fals-pozitiv (fara situatii fals-negative) este penalizata mai putin decât o solutie ce prezinta un singur fals-negativ (fara situatii fals-pozitive). În consecinta, primul cromozom este considerat mai bun. Prin balans se ofera o posibilitate de reducere a situatiilor fals-negativ prin intermediul controlului penalitatii. Evident, numarul de situatii fals-pozitiv va creste.

Strategia de selectie

Strategia de selectie implementata în cadrul prototipului nostru este una clasica, cunoscuta si sub numele de strategia de selectie pe baza de ruleta (Wheel Selection Strategy). Ideea de baza este ca fiecare cromozom prezinta o probabilitate de selectie direct proportionala cu valoare sa de fitness. Cu cât este mai puternic (mai mare fitness-ul sau), cu atât este mai probabil de a fi selectat. Evident, o astfel de abordare este aplicabila problemelor de maximizare. Pentru simplitate, vom prezenta aceasta strategie de selectie asa cum apare ea pentru problemele de maximizare, iar în final vom arata modul în care strategia este aplicata în cadrul prototipului nostru.

Se porneste de la populatia de indivizi de dimensiune N. Etapele selectiei sunt urmatoarele:

- Pentru fiecare individ v_i se calculeaza valoare de fitness eval(v_i).
- Se calculeaza fitness-ul total al populatiei $F = \sum_{i=1}^{N} eval(v_i)$
- Se calculeaza probabilitatea de selectie a unui individ $p_i = eval(v_i) \div F$
- Se calculeaza probabilitatea cumulativa de selectie a unui individ $q_i = \sum_{j=1}^{i} p_j$

- În continuare, se "învârte ruleta" de N ori. De fiecare data se genereaza un numar aleator r în intervalul [0,1]. Daca $r < q_1$ atunci primul cromozom este selectat. În caz contrar se selecteaza cromozomul i pentru care $q_{i-1} < r \leq q_i$.

Dupa cum se poate observa, un cromozom poate fi selectat de mai multe ori. Aceasta este si esenta algoritmilor genetici: cel mai bun cromozom se copiaza de mai multe ori în timp ce cromozomii slabi mor. Copierea de mai multe ori a unui cromozom implica necesitatea existentei la nivelul design-ului prototipului a unui mecanism de clonare a acestora. Aici este utila aplicarea tiparului de proiectare Prototype [8].

Implementarea acestei strategii de selectie în cadrul prototipului nostru ce rezolva o problema de minimizare este foarte simpla. Întregul mecanism este acelasi, doar ca se utilizeaza un fitness virtual al unui individ si nu fitness-ul sau real. Acest fitness virtual se calculeaza astfel: cel mai bun individ are un fitness virtual egal cu fitness-ul real al celui mai slab individ; individul imediat urmator celui mai bun (cu fitness-ul cel mai mic dupa cel mai bun individ) are un fitness virtual egal cu fitness-ul virtual al celui mai bun individ din care se scade valoarea diferentei dintre fitness-ul real al celui mai slab individ si fitness-ul real al individului imediat mai bun celui mai slab individ (cel cu fitness-ul cel mai mare dupa cel mai slab individ); s.a.m.d. Se poate observa ca distantele dintre indivizi se conserva, ceea ce denota un proces de selectie corect.

Un aspect interesant îl constituie amplitudinea pedepsei. Cu cât ea este mai mare cu atât va fi mai probabil ca un individ puternic sa fie selectat de mai multe ori în detrimentul unuia mai slab.

Mai exista si alte strategii de selectie ce ar putea fi implementate (Tournement Selection [23]). Design-ul prototipului este realizat în asemenea maniera încât sa poata fi extins extrem de simplu cu noi modalitati de selectie. Mai exista însa si variatii ale aceleiasi strategii de selectie. Modelul Elitist [23] este o variatie a strategiei descrise mai sus. Singura diferenta consta în faptul ca se forteaza conservarea celui mai bun individ cunoscut pâna la momentul respectiv. Acest lucru se realizeaza foarte simplu: initial se memoreaza cel mai bun cromozom dupa care, înainte de fiecare operatie de selectie se determina cel mai bun si cel mai slab cromozom. Daca cel mai bun este mai puternic decât cel memorat atunci se memoreaza primul cromozom. Daca nu, se elimina din populatie cel mai slab individ si se introduce o copie a celui mai bun cromozom cunoscut pâna în acel moment, acesta fiind memorat. Aceasta abordare ar putea fi aplicata si altor strategii de selectie. În consecinta este utila aplicarea tiparului de proiectare Decorator [8].

Operatori de crossover

Prototipul nostru de masina de tuning ofera posibilitatea utilizarii mai multor tipuri de operatori de crossover. În cele ce urmeaza vom prezenta acesti operatori. Pentru cromozomul cu reprezentare binara operatorul de crossover este unul clasic. Se stabileste într-o maniera aleatoare o pozitie de încrucisare. Aceasta poate fi oriunde în cadrul sirului de biti, mai putin înainte de primul si dupa ultimul bit din reprezentare. În continuare lucrurile functioneaza asa cum am mai spus: considerând doi cromozomi de forma (1101001) si (0010110) si presupunând ca pozitia de încrucisare stabilita este 2, descendentii rezultati vor avea forma (1110110) respectiv (0001001).

Pentru cromozomul cu reprezentare reala masina ofera posibilitatea utilizarii mai multor tipuri de operatori de crossover. Simple Crossover [23] este identic cu

operatorul clasic utilizat în cadrul reprezentarii binare. Singura diferenta este ca nu se lucreaza cu vectori binari ci cu vectori de numere reale. Se observa ca acest operator nu poate produce descendenti neconsistenti, parametrii neparasind domeniul lor valoric.

Un alt operator utilizat pentru reprezentari reale este Arithmetical Crossover [23]. El se defineste ca o combinatie liniara a doi vectori. Daca x_1 si x_2 sunt cromozomii ce trebuie încrucisati, descendentii obtinuti vor fi:

$$x_1' = a \cdot x_1 + (1-a) \cdot x_2$$
$$x_2' = a \cdot x_2 + (1-a) \cdot x_1$$

unde a reprezinta o valoare aleatoare din intervalul [0,1]. Se poate demonstra ca acest operator pastreaza totdeauna consistenta cromozomilor. Experimentele din [23] arata ca algoritmii genetici ce nu folosesc acest operator prezinta o convergenta mai înceata catre solutia optima. În acelasi timp, acest operator ofera sistemului o stabilitate mai mare, cu o redusa deviatie standard a solutiilor optime furnizate de sistem la rulari diferite.

Un ultim operator este Heuristic Crossover [23]. Acest operator prezinta niste particularitati interesante: utilizeaza valoarea de fitness a cromozomilor încrucisati pentru stabilirea "directiei" de cautare; produce doar un descendent; poate sa nu produca nici un descendent. Considerând x_1 si x_2 cromozomii de încrucisat, descendentul obtinut este:

$$x_3 = r \cdot (x_2 - x_1) + x_2$$

unde r este o valoare aleatoare din intervalul [0,1], iar x_2 nu este mai slab ca x_1 (are un fitness mai mic sau egal cu fitness-ul lui x_1). Implementarea operatorului este simpla. Se stabileste cel mai bun cromozom dupa care se produce descendentul pe baza formulei de mai sus. Cromozomul mai slab este înlocuit de descendent. Problema acestui operator este ca el poate produce descendenti neconsistenti. Situatia se rezolva astfel: daca apare neconsistenta se produce alt r si alt descendent pornind de la cromozomii initiali; daca dupa w (stabilit de utilizator) încercari nu se produce un descendent consistent, atunci operatorul renunta si nu va mai produce descendenti. Avantajul acestui operator este însa foarte important, contribuind la determinarea solutiei cu o precizie crescuta. El prezinta capabilitati de reglare fina si de cautare în directiile cele mai promitatoare din spatiul de cautare. Desi în prezent nu utilizam metrici software care sa necesite determinarea unui numar ridicat de zecimale, exista definite câteva metrici care ar putea necesita acest lucru.

Operatori de mutatie

Prototipul nostru de masina de tuning ofera posibilitatea utilizarii mai multor tipuri de operatori de mutare. În cele ce urmeaza vom prezenta acesti operatori. Operatorul masinii furnizeaza o probabilitate de alterare a unei gene (bit ori numar real). Daca o anumita gena se doreste a se modifica, atunci se aplica operatorul de mutare propriu-zis. În cazul reprezentarii binare operatorul este cel clasic: un bit 1 trece în 0 si invers.

În cazul reprezentarii reale operatorii de mutare sunt ceva mai sofisticati. Uniform Mutation [23] este un operator important pentru ca poate evita convergerea

algoritmului într-un optim local. Operatia executata de el este simpla, înlocuind valoarea genei mutate cu o valoare aleatoare din domeniul de definitie corespunzator respectivei gene (domeniul parametrului codificat în aceea gena). Un alt operator este Non-Uniform Mutation [23] înzestrat cu capabilitati de reglare fina. Daca x_k este valoarea genei mutate, atunci noua valoare x_k' se calculeaza astfel:

$$x_k' = x_k + \Delta(t, right(k) - x_k)$$ daca un bit aleator ia valoarea 0

$$x_k' = x_k - \Delta(t, x_k - left(k))$$ daca un bit aleator ia valoarea 1

Valoarea furnizata de functia Δ(t,y) este cuprinsa în intervalul [0,y] astfel încât probabilitatea va aceasta valoare sa fie apropiata de 0 sa creasca pe masura ce t creste (t reprezinta generatia curenta generata). Operatorul permite o cautare uniforma la începutul procesului evolutiv si una locala catre sfârsitul procesului. Right si Left reprezinta limitele domeniului valoric a parametrului codificat în gena mutata. Se poate observa ca operatorul nu produce cromozomi neconsistenti. Functia Δ utilizata are urmatoarea forma:

$$\Delta(t, y) = y \cdot r \cdot (1 - \frac{t}{T})^{b}$$

unde: r este un numar aleator din intervalul [0,1], T este numarul total de generatii ce vor fi executate, iar b este un parametru dat de utilizator reprezentând gradul de neuniformitate. La nivel de design este utila utilizarea tiparului de proiectare Observer [8] în vederea urmaririi progresului executiei algoritmului.

Mai exista si alti operatori de mutare (Boundary Mutation [23]). Totusi ne-am limitat doar la implementarea celor descrisi mai sus. Design-ul prototipului este astfel realizat încât sa permita implementarea simpla si a altor operatori (de crossover sau selectie) daca acest lucru va fi necesar.

5.1.4 Interfata cu utilizatorul

Aceasta componenta nu are o corespondenta directa în cadrul meta-arhitecturii masinii de tuning pentru simplul motiv ca meta-arhitectura se concentreaza exclusiv asupra procesului de parametrizare. În cadrul prototipului nostru, ea reprezinta interfata cu utilizatorul. Astfel, prin intermediul acestei componente, operatorul poate configura algoritmul genetic (specifica ce reprezentare sa se foloseasca, ce operatori de crossover, etc.), poate seta parametrii sai (dimensiune populatie, probabilitate de crossover, de mutatie, balansul, etc.) si poate selecta un anumit schelet de strategie care sa fie parametrizat. Mai mult, utilizatorul poate preciza ce exemple sa se foloseasca pentru reglarea parametrilor, si ce exemple sa fie ascunse acestui proces. Aceasta posibilitate este foarte utila. Înainte ca procesul de reglare sa înceapa, aceasta componenta seteaza evaluatorul în vederea utilizarii la evaluare doar a exemplelor precizate de utilizator. Dupa terminarea procesului de tuning, înainte de afisarea rezultatelor, aceasta componenta seteaza evaluatorul în vederea utilizarii la evaluare a unui alt set de exemple, specificat de asemenea de utilizator, si forteaza o noua evaluare a setului de solutii furnizate de algoritmul de tuning. Aceasta noua evaluare se poate face peste exemple ce nu au fost utilizate în procesul de parametrizare, fiind

deci posibila aplicarea unei tehnici de validare în vederea luarii deciziei de acceptare a unei solutii de parametrizare.

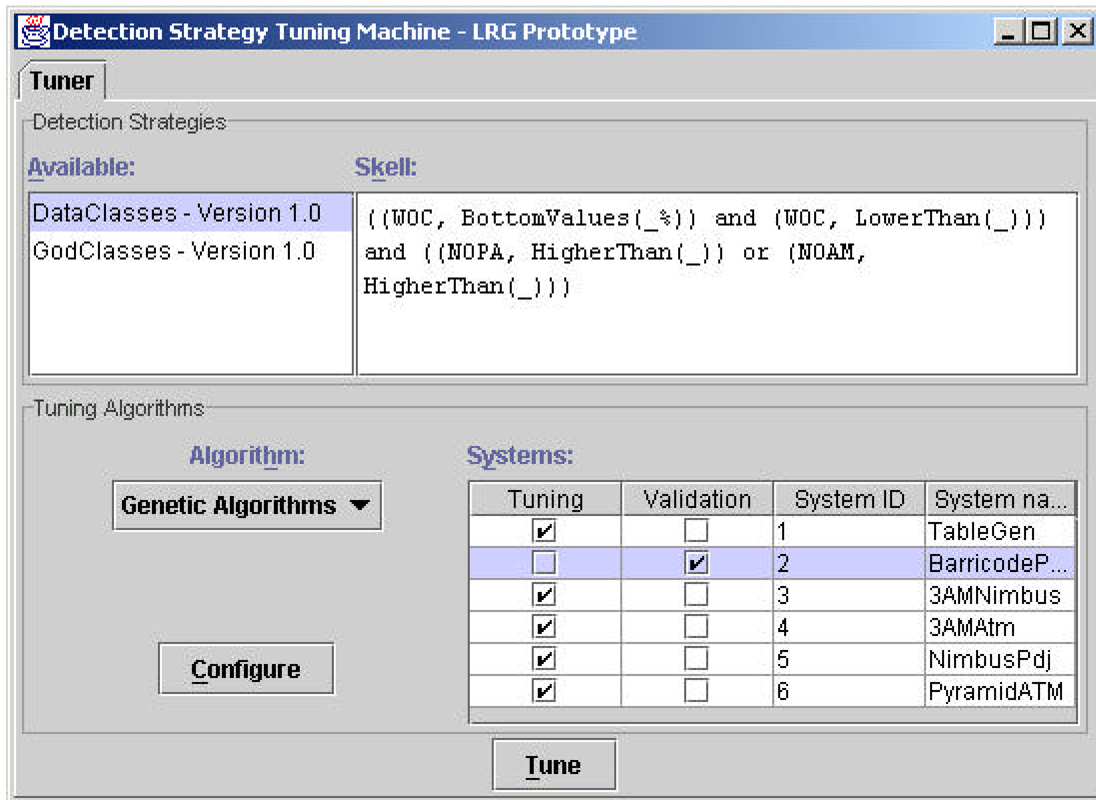

Figura 5.2 Fereastra principala a prototipului de masina de tuning LRG-DSTM

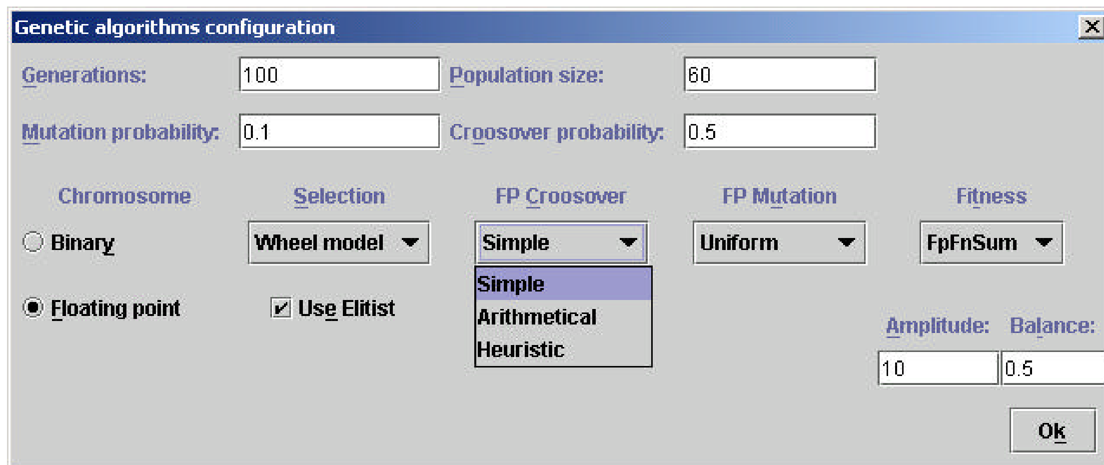

Figura 5.3 Fereastra de configurare a algoritmului de tuning

5.2 Evaluare

În cadrul acestei sectiuni vom analiza solutiile de parametrizare, propuse de prototipul de masina de tuning LRG-DSTM, pentru doua strategii de detectie: DataClasses si GodClasses. Evaluarea urmareste pe de-o parte sa determine daca aceste strategii de detectie, în forma lor din [14], pot fi îmbunatatite din punctul de vedere al calitatii detectiei, daca scheletul lor este parametrizat de o masina de tuning. În acelasi timp vom urmarii si comportamentul masinii si modul în care ea reuseste sa stabileasca parametrii filtrelor de date. Înainte de a prezenta rezultatele, vom trece în

revista modul de desfasurare al unei astfel de activitati si vom analiza anumite situatii anormale ce ar putea apare pe parcursul acestei evaluari.

Se porneste de la un numar de sisteme software orientate obiect, care au fost inspectate manual în vederea determinarii carentelor de proiectare DataClass si GodClass. Aceste sisteme se împart în doua grupuri: un numar de sisteme va fi utilizat pentru operatia de parametrizare executata de masina de tuning, iar restul vor fi utilizate pentru validare. Analiza consta în evaluarea mai multor aspecte, nu neaparat distincte. Pe de-o parte trebuie analizate comparativ performantele de detectie a strategiilor, înainte si dupa operatia de parametrizare executata de prototip. Aceasta înseamna compararea numarului de situatii fals-pozitive si fals-negative, înainte si dupa operatia de parametrizare executata de masina. Pe de alta parte trebuie analizata relevanta parametrilor furnizati de masina, adica trebuie vazut daca ei sunt acceptabili din punctul de vedere al proiectantului scheletului strategiei de detectie. El este cel care stie daca aspectul vizat de metrica si filtrul asociat ei poate fi surprins corect prin utilizarea parametrului filtrului de date furnizat ca solutie.

În vederea compararii performantelor de detectie, pentru fiecare din cele doua grupuri de sisteme, se va determina capacitatea de detectie a strategiilor, când se folosesc parametrii dati în [14]. În continuare, strategiile de detectie vor fi parametrizate de masina de tuning utilizând ca exemple grupul de sisteme dedicate acestei operatii. Apoi, pentru fiecare din cele doua grupuri de sisteme, se va determina aceeasi capacitate de detectie a strategiilor, când se folosesc noii parametrii. Un aspect interesant, legat de operatia de parametrizare, trebuie amintit aici. Daca setul de exemple este suficient de complet, solutia de parametrizare va varia foarte putin, teoretic deloc, de la o executie la alta a procesului de reglare[7] (executata în aceleasi conditii). Daca apar totusi variatii mari este foarte probabil ca setul de exemple sa nu fie suficient de reprezentativ. Principial, aceasta situatie va fi însotita de un numar redus de situatii fals-negativ si fals-pozitiv peste exemplele utilizate în reglare. Întrebarea este cum alegem o singura solutie? Practic, poate fi aleasa oricare dintre solutii pentru ca toate conduc la aceleasi performante (mai ales când solutia variaza foarte putin). Cel mai bine este ca întotdeauna sa se aleaga varianta ce generalizeaza cel mai mult strategia. Daca totusi setul de exemple e suficient de complet, atunci scheletul strategiei de detectie este total irelevant. Principial, aceasta situatie va fi însotita de un numar mare de situatii fals-negativ si fals-pozitiv peste exemplele utilizate în reglare. Reamintim ca noi pornim de la premisa ca scheletele strategiilor sunt corecte.

Este de asteptat ca pentru grupul de sisteme utilizat la reglarea parametrilor precizia de detectie sa fie foarte ridicata, teoretic maxima. Daca acest lucru nu se întâmpla, se pot ridica semne de întrebare serioase asupra corectitudinii scheletului strategiei de detectie (putem vorbi despre un overfitting la nivel de schelet: el surprinde doar anumite aspecte ale carentei). În aceasta situatie, se poate apela la facilitatea de balans pentru cresterea preciziei de detectie a carentelor reale, prin acceptarea aparitiei mai multor situatii fals-pozitive. Atragem atentia ca introducerea unui dezechilibru mare în procesul de reglare a parametrilor, prin facilitatea de balans, poate afecta relevanta acestora. Daca se obtine o precizie de detectie acceptabila pentru grupul de sisteme utilizat la parametrizare, se trece la analiza grupului de

[7] Prin variatie se întelege modificarea solutiei de parametrizare (a valorii parametrilor) pastrându-se însa aceeasi valoare optima a functiei de fitness. Acest lucru se datoreaza platourilor existente în spatiul parametric. Daca si valoarea de fitness optima variaza de la o rulare la alta, atunci înseamna ca au aparut blocari în optime locale. O discutie despre acest aspect ar trebui realizat în contextul evaluarii diferitilor algoritmi de tuning.

validare. Daca precizia de detectie este mai buna ca cea initiala (în principiu mai putine situatii fals-negative, dar si un numar redus de situatii fals-pozitive), atunci se poate trage concluzia ca strategia de detectie, ce utilizeaza parametrii dati de masina de tuning, este mai performanta. În caz contrar, spunem ca a aparut un overfitting: în setul de exemple utilizat la reglare exista niste particularitati parametrice negenerale mai puternice decât cele vizate. Acest lucru se datoreaza faptului ca exemplele nu surprind toate aspectele parametrice ale problemei de proiectare pe care o vizeaza. Situatia se poate rezolva prin adaugarea de noi exemple în setul de reglare.

Înainte de a trage concluzia ca strategia de detectie parametrizata de masina de tuning este mai performanta, trebuie analizata relevanta parametrilor, asa cum am amintit la începutul acestei discutii. În esenta, aceasta relevanta este furnizata de însemnatatea semantica a unui parametru. Spre exemplu, în cazul strategiei de detectie DataClasses este de asteptat ca parametrul filtrului LowerThan asociat metricii WOC sa fie relativ mic, pentru ca ea vizeaza aspectul de clasa "usoara". Spre exemplu, un parametru 0.9 nu este relevant. Nu putem spune ca metrica si filtrul sau surprind aspectul de clasa "usoara" vizat de metrica. Deci, nu vom putea accepta setul de parametrii chiar daca el asigura precizie de detectie crescuta atât pentru setul de reglare cât si pentru cel de validare. Daca irelevanta afecteaza multi parametrii, iar acestia mai sunt si stabili de la o rulare la alta, este mai mult ca sigur ca a aparut un overfitting global: atât setul de reglare cât si cel de validare prezinta regularitati parametrice negenerale mai puternice decât cele vizate. În concluzie, sunt necesare noi exemple. Daca irelevanta afecteaza însa doar un parametru, iar acesta variaza mult de la o rulare la alta realizata în aceleasi conditii, fara a afecta capacitatea de detectie a strategiei, atunci este evident ca exemplele de reglare si de validare nu surprind suficient de bine aspectul vizat de acest parametru.

5.2.1 Procesul de selectie a exemplelor

O atentie deosebita a fost acordata selectiei exemplelor utilizate în experimentul nostru. Calitatea strategiei de detectie parametrizate de masina de tuning depinde foarte mult de aceste exemple. Din acest motiv este impetuos necesar ca setul de exemple sa contina cât mai putine zgomote (exemple pozitive utilizate ca negative si invers). În acelasi timp, se pune problema subiectivitatii. O persoana, care cunoaste conceptul de DataClass, dar care stie exact si modul în care se identifica clasele afectate de aceasta carenta de proiectare utilizând tehnica strategiilor de detectie, va fi tentata sa indice exemple pozitive ori negative pe baza aspectelor urmarite de strategia de detectie (mai exact urmarite de metricile utilizate) si pe baza parametrilor ei. Astfel, s-ar putea indica drept exemple pozitive doar acele clase care prezinta simptoamele surprinse în scheletul strategiei, în timp ce alte clase, si ele afectate de aceeasi carenta de proiectare, sa nu fie indicate ca exemple pozitive doar pentru ca ele nu se potrivesc respectivelor simptoame. Acest lucru nu ar fi daunator daca am sti absolut sigur ca un schelet de strategie surprinde chiar toate aspectele problemei de proiectare vizate. Din pacate acest lucru nu poate fi afirmat. Se pune întrebarea: de ce ne-ar deranja acest lucru? Noi am specificat ca presupunem corectitudinea scheletului strategiilor. Adevarat, însa în urma operatiei de parametrizare nu am putea observa "comportamentul anormal" al masinii de tuning datorat unui schelet incomplet în raport cu exemplele utilizate în stabilirea parametrilor lui.

Pentru a evita aceste probleme, exemplele au fost selectate dupa cum urmeaza. În primul rând, a fost creat un catalog de reguli informale, care stau la baza definirii strategiilor de detectie descrise în [14]. Identificarea manuala a exemplelor pozitive

pentru o anumita carenta de proiectare urmeaza a se realiza pe baza descrierii informale a respectivei carente, descriere cuprinsa în catalog mai sus amintit. Astfel, evitam problema subiectivitatii descrisa mai sus. În continuare, pentru gasirea exemplelor propriu-zise, s-au utilizat un numar de sisteme software de dimensiuni similare, scrise în limbajele de programare Java sau C++, despre care se stie cu siguranta ca sunt afectate de carente de proiectare (în special DataClass si GodClass). Codul fiecarui sistem a fost dat uneia sau mai multor persoane în vederea efectuarii unei analize manuale cu scopul identificarii tuturor carentelor de proiectare (în special al celor de tip DataClass si GodClass) ce-l afecteaza. Identificarea urma sa se realizeze pe baza catalogului de reguli informale. Fiecare persoana cunoaste foarte bine sistemul pe care l-a primit pentru analiza, deoarece acelasi sistem a fost refactorizat cu succes de aceeasi persoana în cadrul proiectului G.O.O.D. (disciplina "Ingineria programarii II",anul V, specializarea Soft I, din cadrul Facultatii de Automatica si Calculatoare din Timisoara). Acest lucru ne-a asigurat ca exemplele furnizate vor contine un numar redus de zgomote. Toate carentele de proiectare DataClass si GodClass, identificate de aceste persoane, au fost utilizate ca exemple pozitive pentru reglarea parametrilor strategiei de detectie DataClasses, respectiv ca exemple pozitive pentru reglarea parametrilor strategiei de detectie GodClasses. Toate clasele din sistemele mai sus amintite, care nu au fost raportate ca fiind afectate de carenta DataClass, au fost utilizate ca exemple negative în vederea reglarii parametrilor strategiei de detectie DataClasses. În mod asemanator au fost tratate si clasele ce nu au fost remarcate ca fiind afectate de carenta de proiectare GodClass.

Toate exemplele astfel obtinute au fost introduse în depozitul de exemple al prototipului. Pentru calcularea metricilor software s-au utilizat programele TableGen, JTable si definitiile implementarii lor din cadrul sistemului Prodeoos. Toate aceste programe fac parte din pachetul ProDetection, utilizat în detectia carentelor de proiectare în sistemele orientate pe obiecte si care utilizeaza în acest sens conceptul de strategie de detectie.

5.2.2 Strategia de detectie GodClasses

În cadrul acestei sectiuni vom prezenta modul de desfasurare al activitatii de parametrizare a scheletului strategiei de detectie GodClasses. Initial vom descrie setul de exemple utilizat si performantele de detectie a strategiei cu parametrii initiali în raport cu acest set. Strategia are urmatoarea forma:

GodClasses:= ((ATFD,TopValues(20%)) and (ATFD,HigherThan(4))) and ((WMC,HigherThan(20)) or (TCC,LowerThat(0.33)))

Sistem	**Exemple pozitive**	**Exemple negative**
TableGen	6	15
Barricode-Payroll	1	44
3AM-Atm	1	28
PDJ-Nimbus	1	55
Total	**9**	**142**

Tabel 5.1 Setul de exemple utilizat la reglarea parametrilor

Sistem	Exemple pozitive	Exemple negative
3AM-Nimbus	1	22
Total	**1**	**22**

Tabel 5.2 Setul de exemple utilizat la validare

Sistem	Exemple pozitive detectate (Adevarat-Pozitiv)	Exemple negative detectate (Fals-Pozitiv)
TableGen	0	0
Barricode-Payroll	1	1
3AM-Atm	1	3
PDJ-Nimbus	1	0
Total	**3**	**4**

Tabel 5.3 Performantele de detectie a strategiei cu parametrii initiali peste sistemele utilizate la reglarea parametrilor

Sistem	Exemple pozitive detectate (Adevarat-Pozitiv)	Exemple negative detectate (Fals-Pozitiv)
3AM-Nimbus	0	0
Total	**0**	**0**

Tabel 5.4 Performantele de detectie a strategiei cu parametrii initiali peste sistemele utilizate la operatia de validare

Urmarind aceste tabele putem sa tragem o concluzie asupra performantelor de detectie a strategiei cu parametrii initiali. Pentru sistemele utilizate la stabilirea parametrilor, doar 3 dintr-un total de 9 carente reale au fost detectate (33.33%) si 4 clase neafectate de aceasta carenta au fost suspectate eronat (2,8%). În mod asemanator, în sistemul utilizat la validare, carenta GodClass nu a fost detectata si nu s-au introdus situatii fals-pozitive. În continuare vom încerca sa îmbunatatim capacitatea de detectie a strategiei prin stabilirea unor parametrii cu ajutorul masinii de tuning. Parametrii algoritmului de tuning au fost:

- Dimensiune populatie 60
- Numar de generatii 100
- Probabilitatea de mutare 0.1
- Probabilitatea de încrucisare 0.5
- Reprezentare Reala
- Strategia de selectie Wheel Model în combinatie cu modelul Elitist
- Operatorul de crossover Euristic cu numar maxim de încercari 10
- Operator de mutatie Mutatie uniforma
- Amplitudine de penalitate 100
- Balans 0.5

S-au efectuat un numar de 10 rulari ale algoritmului, toate efectuate în aceleasi conditii. Cele mai bune solutii identificate la fiecare rulare, adica cele cu o valoare de fitness minima, sunt prezentate în anexa.

Înca de la bun început observam ca a aparut o situatie în care masina s-a blocat într-un optim local. Solutiile de parametrizare furnizate în aceasta situatie sunt

suboptimale în raport cu solutiile gasite la alte rulari. Când algoritmul nu s-a blocat în optimul local, aceleasi solutii au fost indicate de masina ca fiind suboptimale. Aceasta blocare nu este însa grava având în vedere ca doar o singura inconsistenta nu a fost eliminata. Utilizatorul poate evita aceasta blocare furnizând masinii un balans putin mai mare, de exemplu 0.55. Mai mult, daca am fi avut un numar mai mare de exemple pozitive similare celor ce apar ca fals-negative în solutiile suboptimale atunci blocarea în optim local ar fi fost aproape imposibila. În continuare solutiile suboptimale vor fi ignorate. Solutiile optime gasite prezinta un fals-negativ si doua fals-pozitive peste sistemele utilizate la tuning-ul parametrilor si un singur fals-pozitiv pentru sistemul de validare. Exemplele ce apar ca fals-negative si fals-pozitive sunt întotdeauna aceleasi.

Analizând valorile pragurilor filtrelor de date furnizate de masina în raport cu valorile initiale constatam ca pentru detectia corecta a unui numar mare de exemple pozitive din cadrul sistemelor utilizate la parametrizare este necesar un prag de valoare 2 pentru membrul (Atfd,HigherThan(_)), mai mic decât cel initial. Acest lucru indica faptul ca parametrul initial era prea mare, cu alte cuvinte exista o problema de parametrizare. Mai mult, noua valoare a acestui prag asigura si detectia carentei GodClass din sistemul de validare (ce are o metrica Atfd de valoare 3), ea nefiind detectata initial. S-ar putea întreba de unde stim ca pragul prea mare a fost cauza lipsei de acuratete a strategiei. Raspunsul e foarte simplu: o analiza manuala asupra valorilor metricilor a demonstrat acest lucru. În acelasi timp, observam ca valorile parametrilor membrilor (Wmc,HigherThan(_)) si (Tcc,LowerThan(_)) au crescut respectiv scazut fata de valorile initiale ceea ce arata ca acesti membrii constrâng mai mult. Initial exemplul pozitiv din sistemul de validare nu era detectat lucru ce arata ca valorile pragurilor initiale erau prea specifice. Singurul membru al strategiei care devine mai general este (Atfd,HigherThan(_)) masina de tuning furnizând valoarea 2 pentru acest prag. Parametrul membrului (Atfd,TopValues(_)) ramâne la o valoare aproximativ egala cu valoarea initiala. În concluzie, îmbunatatirea capacitatii de detectie a exemplelor pozitive se datoreaza valorii mai reduse pentru pragul membrului (Atfd,HigherThan(_)) ajungând la aceeasi concluzie ca si în cazul analizei manuale a valorii metricilor.

Din punctul de vedere al oscilatiei valorii parametrilor de la o rulare la alta se constata urmatoarele. Valoarea pragului asociat membrului (Atfd,TopValues(_)) oscileaza foarte putin într-o zona extrem de apropiata de valoarea initiala a acestui prag. Oscilatia este mai mult ca sigur datorata unui platou din spatiul parametric. Existenta sa e foarte probabila daca ne gândim ca utilizam sisteme de dimensiuni reduse (în medie 34 de clase pe sistem) iar o variatie mica a valorii unui procent de clase selectate pentru analiza nu aduce practic schimbari semnificative în setul selectat (mai ales atunci când exista clase cu aceeasi valoare pentru metrica Atfd). Parametrul membrului (Atfd, HigherThan(_)) este perfect stabil. Singurele oscilatii demne de analizat în profunzime apar la parametrii membrilor (Wmc,HigherThan(_)) si (Tcc,LowerThan(_)). Este important de observat ca exista o legatura între domeniile valorice în care oscileaza cei doi parametrii. În principiu, pragul membrului (Tcc,LowerThan(_)) oscileaza într-un domeniu de valori fie mai mici de 0.19 fie în intervalul 0.28-0.32. În acest timp, pragul membrului (Wmc,HigherThan(_)) oscileaza în intervalul 31-32 respectiv 34-44. Acest lucru se datoreaza faptului ca exista cel putin un exemplu pozitiv asupra caruia masina nu se poate hotarî pe care ramura a arborelui de detectie (un arbore asociat strategiei de detectie asemanator arborelui asociat unei expresii aritmetice) trebuie detectat respectivul exemplul: prin membrul (Tcc,LowerThan(_)), (Wmc,HigherThan(_)) sau prin amândoua. Motivul este setul mic de exemple avut la dispozitie care nu contine informatie suficienta

pentru luarea acestei decizii. Introducerea de exemple noi va rezolva cu siguranta aceasta problema.

Sistem	Exemple pozitive detectate (Adevarat-Pozitiv)	Exemple negative detectate (Fals-Pozitiv)
TableGen	5	0
Barricode-Payroll	1	0
3AM-Atm	1	2
PDJ-Nimbus	1	0
Total	**8**	**2**

Tabel 5.5 Performantele de detectie a strategiei cu parametrii furnizati de masina de tuning peste sistemele utilizate la reglarea parametrilor

Sistem	Exemple pozitive detectate (Adevarat-Pozitiv)	Exemple negative detectate (Fals-Pozitiv)
3AM-Nimbus	1	1
Total	**1**	**1**

Tabel 5.6 Performantele de detectie a strategiei cu parametrii furnizati de masina de tuning peste sistemele utilizate la operatia de validare

Din punctul de vedere al performantelor de detectie obtinute în urma utilizarii noilor parametrii observam o îmbunatatire, indiferent de solutia de parametrizare aleasa. Era de asteptat ca peste sistemele utilizate la reglarea parametrilor numarul de inconsistente sa se reduca. Astfel, 8 din 9 exemple pozitive au fost detectate (88.88%), în timp ce numarul de exemple negative clasificate ca pozitive s-a redus la 2 dintr-un total de 142 de exemple negative (1.4%). Reducerea numarului de situatii fals-negative arata ca a existat o problema de parametrizare si la nivelul membrilor (Wmc,HigherThan(_)) sau (Tcc,LowerThan(_)). Pentru sistemul de validare observam detectia exemplului pozitiv de GodClass, dar si introducerea unei situatii fals-pozitive din dintr-un numar de 22 exemple negative. Introducerea unui fals-pozitiv nu este importanta atâta timp cât carenta reala a fost detectata. În concluzie avem performante de detectie mai bune decât initial peste sistemul de validare.

Din punctul de vedere al proiectantului strategiei de detectie, care stie ce aspect al carentei de proiectare trebuie surprins de fiecare metrica, valorile pragurilor furnizate de masina de tuning sunt relevante. Cu alte cuvinte, particularitatile parametrice intuite de proiectant chiar exista. Valoarea pragului membrului (Atfd,TopValues(_)) este relativ mica. Acest lucru era de asteptat întrucât numarul de carente GodClass dintr-un sistem e relativ mic raportat la numarul total de clase din sistem. Un GodClass tinde sa centralizeze inteligenta sistemului delegând detalii minore unui set de clase triviale, de obicei suport de date (DataClass). Parametrul membrului (Atfd,HigherThan(_)) surprinde si el destul de bine faptul ca un GodClass acceseaza date apartinând altor clase. Membrul (Wmn,HigherThan(_)) surprinde excelent prin valoarea pragului aspectul de clasa mare, complexa, ce tinde sa centralizeze inteligenta sistemului, iar parametrul membrului (Tcc,LowerThan(_)) este foarte relevant în contextul identificarii claselor necoezive. Este de asteptat ca o clasa necoeziva sa aiba o metrica Tcc coborâta. Din acest punct de vedere categoria de solutii cu pragul asociat membrului (Tcc,LowerThan(_)) mai mic sunt teoretic ceva mai bune (exceptând solutia ce furnizeaza un prag de valoare 0.02 pentru acest membru, ce ar însemna practic clase total necoezive). Totusi nu putem afirma

categoric acest lucru. Introducerea de noi exemple în setul de reglare va clarifica cu siguranta aceasta problema.

Nu putem însa încheia înainte de a observa ca exista o situatie fals-negativa si doua fals-pozitive în sistemele utilizate la tuning-ul parametrilor care nu sunt eliminate. O situatie fals-pozitiva apare si peste sistemul de validare. Numarul redus de situatii de inconsistenta pot indica faptul ca exista zgomote în setul de exemple, adica carente de proiectare neidentificate de analizorii umani ai sistemelor sau raportate eronat de acestia. Pe de alta parte, se poate considera si faptul ca scheletul strategiei de detectie nu surprinde un aspect foarte particular al carentei de proiectare GodClass. Cea mai grava situatie ar apare daca solutiile reprezinta optime locale datorate blocarii masinii de tuning în optime locale.

O analiza la nivel de cod asupra exemplului ce apare ca fals-negativ arata ca el este într-adevar un GodClass, deci nu e un zgomot. Problema este însa alta: metrica Atfd asociata exemplului are valoarea 1. Întelegem acum de ce masina de tuning nu elimina aceasta situatie fals-negativa. Acest lucru ar necesita doar o valoare de prag pentru (Atfd,HigherThan(_)) egala cu 1. Din pacate se introduc astfel multe situatii fals-pozitive (în numar de 5) dupa cum rezulta din executiile algoritmului de tuning la un balans ridicat. Acest lucru demonstreaza ca masina nu s-a blocat într-un optim local, ci exista anumite contradictii între acest exemplu pozitiv si mai multe exemple negative în raport cu acest schelet de strategie. Identificarea inexactitatii din scheletul strategiei necesita investigatii destul de complexe. Din acest motiv ne vom rezuma doar la observatia ca un prag de valoare 1 pentru (Atfd,HigherThan(_)) confera membrului un statut mult prea general: orice clasa ce acceseaza datele altei clase devine automat suspecta de a fi GodClass din punctul de vedere al acestui membru. Aceasta e o generalizare prea mare deoarece exista situatii în care accesul la date e necesar (ex. tiparul de proiectare Observer [8]). Din acest motiv banuim ca problema se gaseste la nivelul acestui membru care nu cuantifica suficient de bine aspectul carentei de proiectare GodClass ce se refera la accesul la datele unor clase triviale.

În ceea ce priveste situatiile fals-pozitive, se observa ca ele nu pot fi eliminate de masina de tuning. Metrica Atfd ia în cazul lor valoarea 5 în timp ce valoarea Tcc este 0. Pentru a putea fi eliminate masina trebuie sa ridice pragul membrului (Atfd,HigherThan(_)) la 6 lucru ce conduce dupa cum am vazut la imposibilitatea detectiei unor exemple pozitive mai numeroase. Este important de observat ca masina nu s-a blocat într-un optim local. O analiza la nivel de cod asupra celor doua exemple negative arata ca ele nu sunt nici zgomote în setul de exemple, de unde rezulta ca scheletul strategiei este incomplet. Numarul mic de situatii fals-pozitive arata ca incompletitudinea e probabil mica. Desi atât membrul (Atfd,HigherThan(_)) cât si (Tcc,LowerThan(_)) ar putea fi suspectati de aparitia acestor falsitati pozitive, la momentul actual înclinam sa credem ca tot primul membru este cel care nu-si face datoria. În ceea ce priveste fals-pozitivul din sistemul de validare, o analiza la nivel de cod a aratat ca este posibil ca acest exemplu sa fie pozitiv si nu negativ, adica un reprezinta un zgomot.

5.2.3 Strategia de detectie DataClasses

În aceasta sectiune vom prezenta modul în care s-a desfasurat activitatea de parametrizare a scheletului strategiei de detectie DataClasses. În cadrul acestui experiment vom vedea ce se întâmpla în situatia în care scheletul este incomplet si cum ne putem da seama de acest aspect. La început, vom prezenta setul de exemple

utilizat si performantele de detectie a strategiei cu parametrii initiali în raport cu acest set. Strategia initiala are forma:

DataClasses:= ((WOC,BottomValues(33%)) and (WOC,LowerThan(0.33))) and ((NOPA,HigherThan(5)) or (NOAM, HigherThan(5)))

Sistem	**Exemple pozitive**	**Exemple negative**
TableGen	7	14
3AM-Nimbus	0	23
3AM-Atm	6	23
PDJ-Nimbus	2	54
Pyramid-Atm	3	12
Total	**18**	**126**

Tabel 5.7 Setul de exemple utilizat la reglarea parametrilor

Sistem	**Exemple pozitive**	**Exemple negative**
Barricode-Payroll	4	41
Total	**4**	**41**

Tabel 5.8 Setul de exemple utilizat la validare

Sistem	**Exemple pozitive detectate (Adevarat-Pozitiv)**	**Exemple negative detectate (Fals-Pozitiv)**
TableGen	3	0
3AM-Nimbus	0	0
3AM-Atm	2	1
PDJ-Nimbus	1	1
Pyramid-Atm	0	1
Total	**6**	**3**

Tabel 5.9 Performantele de detectie a strategiei cu parametrii initiali peste sistemele utilizate la reglarea parametrilor

Sistem	**Nr. exemple pozitive detectate (Adevarat-Pozitiv)**	**Nr. exemple negative detectate (Fals-Pozitiv)**
Barricode-Payroll	1	0
Total	**1**	**0**

Tabel 5.10 Performantele de detectie a strategiei cu parametrii initiali peste sistemele utilizate la operatia de validare

Urmarind aceste tabele putem sa tragem o concluzie asupra performantelor de detectie a strategiei cu parametrii initiali: pentru sistemele utilizate la stabilirea parametrilor, 33% din carentele reale au fost detectate si 2.38% dintre clasele neafectate de aceasta carenta au fost suspectate eronat. În mod asemanator, pentru sistemul utilizat la validare, avem 25% respectiv 0%. Din punctul de vedere al persoanei ce cauta carentele de proiectare DataClasses, primul procent este mult prea mic.

În continuare vom discuta solutiile de parametrizare furnizate de masina de tuning. Parametrii algoritmului de tuning au fost:

- Dimensiune populatie 60
- Numar de generatii 100
- Probabilitatea de mutare 0.1
- Probabilitatea de încrucisare 0.5
- Reprezentare Reala
- Strategia de selectie Wheel Model în combinatie cu modelul Elitist
- Operatorul de crossover Euristic cu numar maxim de încercari 10
- Operator de mutatie Mutatie uniforma
- Amplitudine de penalitate 100

Singurul parametru ce se modifica pe parcursul experimentului este balansul. Pentru fiecare configuratie fixa a algoritmului (incluzând balansul) s-au efectuat un numar de 5 rulari. Cele mai bune solutii, adica cele cu o valoare de fitness minima, gasite pentru fiecare astfel de configuratie sunt prezentate în anexa.

Solutiile de parametrizare obtinute la un balans de 0.5

La un balans de 0.5, o situatie fals-negativ este penalizata la fel ca o situatie fals-pozitiv. Se observa de la bun început ca exista doua categorii de solutii de parametrizare posibile: o prima categorie prezinta un numar de 6 fals-negative si 2 fals-pozitive (întotdeauna aceleasi exemple apar ca fals-negative respectiv fals-pozitive), iar a doua categorie prezinta 7 fals-negative si 1 fals-pozitiv (întotdeauna aceleasi exemple apar ca fals-negative respectiv fals-pozitive). Toate aceste situatii apar peste sistemele utilizate pentru tuning. Sistemul nu poate decide care dintre cele doua categorii de solutii este mai buna, pentru ca un fals-pozitiv si un fals-negativ sunt penalizate exact la fel, iar numarul de inconsistente este acelasi (8). Daca s-ar efectua un tuning la un balans mai mare, de exemplu 0.55, atunci a doua categorie de solutii va dispare. Aceste solutii prezinta un fals-negativ în plus fata de solutiile din prima categorie, dar au si un fals-pozitiv mai putin. Cum la acest balans (0.55), un fals-negativ e penalizat mai mult decât un fals-pozitiv, prima categorie este considerata mai buna. Diferitele rulari ale masinii de tuning, efectuate la un balans de 0.55, arata ca solutiile din prima categorie devin mai puternice.

Este important de remarcat faptul ca s-a identificat o prima problema: exista un exemplu pozitiv si unul negativ care nu pot fi satisfacute simultan. Prin urmare, exista un aspect al carentei DataClass ce nu e surprins în schelet. Aceasta problema a scheletului va fi discutata putin mai târziu.

În cadrul categoriei de solutii cu 7 fals-negative si 1 fals-pozitiv se mai observa o particularitate: parametrul membrului (Woc,BottomValues(_)) este stabilit fie undeva peste 70%, fie undeva în jurul valorii 55%-54%. În ambele cazuri, parametrii membrilor (Nopa,HigherThan(_)) si (Noam,HigherThan(_)) sunt foarte stabili. Însa, în cel de-al doilea caz, parametrul membrului (Woc,LowerThan(_)) oscileaza puternic, în timp ce în primul caz el se stabilizeaza undeva în jurul valorii 0.35. În concluzie, putem spune ca în aceasta categorie de solutii, avem o varianta de parametrizare stabila si una instabila. Evident, varianta stabila este mai importanta din punctul nostru de vedere. Vom vedea însa ca aceasta stabilizare este irelevanta.

Toate solutiile de parametrizare se caracterizeaza printr-un numar mare de situatii fals-negative. Mai mult, nici una nu aduce îmbunatatiri de detectie pentru sistemul de validare, în ciuda faptului ca exista exemple pozitive în cadrul sistemelor utilizate la parametrizare, ce sunt similare celor din sistemul de validate. Aceste

exemple pozitive apar însa ca fals-negative în solutiile de parametrizare indicate de masina, ceea ce explica lipsa de îmbunatatire a performantelor de detectie peste sistemul de validare. Pentru ca exemplele pozitive din sistemul de validare sa fie detectate este necesar mai întâi ca exemplele pozitive similare din sistemele utilizate la reglare sa fie detectate, adica sa nu mai reprezinte situatii fals-negative. Din pacate, sistemul nu reduce mai mult numarul de situatii fals-negative pentru ca ar introduce si mai multe situatii fals-pozitive. În concluzie, exista contradictii numeroase între exemplele negative si pozitive în raport cu acest schelet de strategie. Trebuie mentionat ca, la o analiza manuala a codului, s-a constatat ca exemplele pozitive indicate ca fals-negative sunt tipice si ar trebui detectate. Datorita acestui fapt, nu putem accepta nici una din solutiile de parametrizare furnizate de masina.

Analizând problema mai sus amintita, se constata o contradictie între exemplul pozitiv FriendStructT din sistemul TableGen si exemplul negativ SetMyTime din sistemul PDJ-Nimbus. FriendStructT are o metrica Nopa de valoare 5 iar Woc este 0.28. Pentru SetMyTime avem valorile 7 respectiv 0. Pentru categoria de solutii cu 7 fals-negative si 1 fals-pozitiv, (Nopa,HigherThan(_)) are ca parametru pe 8 sau 9 pentru a elimina acest fals-pozitiv. Parametrul ar mai creste pentru a elimina si fals-pozitivul CAtm1Dlg din sistemul Pyramid-Atm, similar lui SetMyTime, dar acest lucru nu e posibil pentru ca ar apare mai multe situatii fals-negative. În categoria de solutii cu 6 fals-negative si 2 fals-pozitive parametrul mai sus amintit ia valoarea 5, exact cât trebuie pentru a elimina fals-negativul FriendStructT. Dupa cum vedem, cele doua exemple sunt contradictorii si nu pot fi satisfacute prin acest schelet. Printr-o analiza manuala la nivel de cod, s-a ajuns la concluzia ca metrica Woc ar fi trebuit sa faca distinctia între exemplul pozitiv si cel negativ mai sus mentionate. SetMyTime este o clasa cu multe atribute publice dar nu este o clasa simpla sau "usoara". Ea face parte din interfata grafica a sistemului de care apartine si are deosebit de multa functionalitate. Totusi, metrica Woc o interpreteaza ca fiind o clasa usoara datorita numarului mare de atribute publice, care, prin modul de definire al metricii, conduc la neglijarea functionalitatii ridicate a clasei. Cu alte cuvinte, membrul (Woc,LowerThan(_)) este prea general în contextul detectiei carentei de proiectare DataClass, el neputând sa cuantifice singur si corect aspectul de "clasa usoara". Parametrul acestui membru variaza atât de haotic tocmai datorita acestui fapt.

Solutiile de parametrizare obtinute la un balans de 0.7

Când parametrul balans al algoritmului de tuning este stabilit la valoarea 0.7, se constata ca toate solutiile de parametrizare prezinta peste exemplele utilizate la reglare un numar de 3 situatii fals-negative si 8 situatii fals-pozitive. Exemplele asociate situatiilor fals-negative si fals-pozitive sunt aceleasi în fiecare caz. Pentru sistemul de validare nu se constata nici un fel de îmbunatatire fata de situatia initiala.

Din punctul de vedere al parametrilor, se constata o stabilizare a valorii pragurilor pentru membrii (Nopa,HigherThan(_)) si (Noam,HigherThan(_)) la valoarea 5. În ceea ce priveste parametrul membrului (Woc,BottomValues(_)) se constata o usoara oscilatie aproximativ în intervalul 70%-90%. Parametrul membrului (Woc,LowerThan(_)) oscileaza din nou destul de mult, în ciuda faptului ca initial parea sa se fi stabilizat. Aceasta oscilatie arata ca exista în continuare contradictii serioase între exemplele pozitive si cele negative cauzate de incompletitudinea scheletului strategiei.

Comparând, din punctul de vedere al valorii parametrilor, solutiile obtinute de masina pentru un balans de 0.55 cu cele obtinute pentru un balans de 0.7 se constata

urmatoarele: valoarea procentului a crescut în timp ce pragul membrului (Noam,HigherThan(_)) a coborât. Procedând asa, a fost posibila eliminarea a 3 fals-negative. Din pacate, s-au introdus astfel si 6 fals-pozitive. Prin urmare, între exemplele corespunzatoare fals-negativelor eliminate si fals-pozitivelor nou introduse apar iarasi contradictii, în sensul ca exemplele nu pot fi satisfacute simultan de scheletul strategiei de detectie. În continuare vom prezenta concluziile obtinute în urma analizei manuale a codului si a valorii metricilor asociate acestor exemple.

Sistem	**Clasa**	**Woc**	**Nopa**	**Noam**
3AM-Atm	Account	0.46	0	8
3AM-Atm	Bank	0.45	0	6
Pyramid-Atm	CAccount	0.28	0	5

Tabel 5.11 Lista exemplelor pozitive satisfacute la un balans de 0.7

Sistem	**Clasa**	**Woc**	**Nopa**	**Noam**
3AM-Atm	ATM	0.37	0	10
3AM-Atm	ATMFrame	0.41	13	11
3AM-Atm	GetPinPanel	0.16	0	5
3AM-Nimbus	CCalibrate	0.25	6	0
Pyramid-Atm	CMoneyRoll	0.45	0	6

Tabel 5.12 Dintre exemplele negative ce apar ca fals-pozitive la un balans de 0.7

Pe baza acestor tabele si a solutiilor de parametrizare obtinute la un balans de 0.5 se poate observa imposibilitatea satisfacerii simultane a tuturor exemplelor mai sus amintite. Înca de la bun început se observa ca metrica Woc nu poate face distinctia între exemplele pozitive si cele negative, desi, pe baza analizei manuale a codului, s-a ajuns la concluzia ca ea ar fi trebuit sa faca acest lucru. Exemplele pozitive cer ca membrul (Woc,HigherThan(_)) sa aiba un parametru de minimul 0.46 ceea ce conduce si la detectie exemplelor negative. Prin urmare masina de tuning încearca sa utilizeze ceilalti parametrii pentru a realiza corect clasificarea exemplelor. Se poate întelege acum cum a procedat masina pentru reducerea situatiilor de clasificare gresita. În cadrul solutiilor de parametrizare obtinute la balansul de 0.5, categoria de solutii cu 7 fals-negative si 1 fals-negativ prezentau un procent ridicat pentru ca se încerca satisfacerea exemplelor pozitive mai sus amintite. Totusi, masina nu putea reduce parametrul membrului (Noam,HigherThan(_)) la 5 pentru ca ar fi introdus multe fals-pozitive lucru nepermis la acel balans. În acelasi timp, masina a crescut parametrul (Nopa,HigherThan(_)) pentru a reduce numarul de fals-pozitive. Exemplele negative SetMyTime si CCalibrate cereau acest lucru (Obs. În cadrul solutiilor 7 fals-negative si 1 fals-pozitiv ce aveau un procent mai redus, CCalibrate nu intra în calcul pentru ca membrul (Woc,BottomValues(_)) îl elimina de la o clasificare gresita). Parametrul membrului (Woc,HigherThan(_)) se stabiliza datorita exemplelor ATM si ATMFrame, pentru ca oricum exemplele pozitive nu puteau fi satisfacute. La un balans de 0.7, se asigura o penalitate mai redusa pentru celelalte fals-pozitive introduse decât pentru pierderea exemplelor pozitive, motiv pentru care parametrul membrului (Noam,HigherThan(_)) scade la 5.

În cadrul solutiilor obtinute la un balans de 0.55 parametru membrului (Woc,ButtonValues(_)) era mai redus pentru ca altfel s-ar mai fi introdus un fals-pozitiv si anume CCalibrate, lucru nepermis la acel balans. Totusi, la un balans de 0.7, procentul a crescut pentru a se clasifica într-un mod corect cele trei exemple pozitive (aceasta crestere e absolut necesara dupa cum a reiesit din analiza manuala). Acest

lucru introduce CCalibrate ca fals-pozitiv el fiind foarte asemanator lui SetMyTime. Parametrul membrului (Nopa,HigherThan(_)) nu poate fi crescut pentru ca s-ar introduce un fals-negativ (FriendStrunctT) care este mult mai mult penalizat decât cele doua situatii fals-pozitive (SetMyTime si CCalibrate).

O analiza manuala a codului exemplelor pozitive si negative prezentate în tabelele de mai sus a condus la identificarea urmatoarelor situatii contradictorii. AtmFrame si CCalibrate nu vor mai fi discutate pentru ca sunt asemanatoare lui SetMyTime despre care am vorbit.

Clasele ATM, CMoneyRoll si nu sunt raportate ca DataClasses de catre analizorii umani ai sistemelor pentru ca ele au metode complexe care ofera claselor multa functionalitate. Totusi, numarul mare de metode de tip accesor din interfata publica asociata acestor clase si modul de definire al metricii Woc conduc la interpretarea clasei ca fiind usoara si lipsita de functionalitate. Apare aceeasi problema asociata metricii Woc: desi exista multa functionalitate în aceste clase, numarul mare de metode accesor conduc la neglijarea functionalitatii lor. Prin urmare, membrul (Woc,LowerThan(_)) este prea general în contextul detectiei carentei de proiectare DataClass, el neputând sa cuantifice singur si corect aspectul de "clasa usoara". În ceea ce priveste fals-pozitivul GetPinPanel lucrurile sunt asemanatoare. Din codul ei nu reiese ca ar avea prea multa functionalitate, dar totusi, ea face parte din interfata grafica a sistemului, mostenind din libraria AWT multa functionalitate. Acesta este motivul pentru care ea nu a fost considerata DataClass de catre analizori. Observam ca metrica Woc nu poate captura acest aspect.

Trebuie remarcat ca pentru sistemul de validare nu se constata nici un fel de îmbunatatire a capacitatii de detectie a strategiei fata de situatia initiala. Dupa cum am mai spus, exista exemple pozitive în sistemele utilizate la reglare care sunt similare cu exemple pozitive din programul de validare. Din pacate, respectivele exemple pozitive, desi DataClass-uri tipice, reprezinta în continuare situatii fals-negative. În consecinta, nu este posibil sa acceptam nici o solutie de parametrizare furnizata de masina la un balans de 0.7.

Solutiile de parametrizare obtinute la un balans de 0.8

Solutiile de parametrizare obtinute la acest balans prezinta un numar de 2 fals-negative si 11 fals-pozitive. Eliminarea unei situatii fals-negative a condus la introducerea unui numar de 3 fals-pozitive în plus. Este foarte important de amintit ca apare o îmbunatatire a capacitatii de detectie peste sistemul de validare.

Din punctul de vedere al parametrilor solutiilor, observatiile sunt similare celor amintite la discutia solutiilor de parametrizare obtinute la un balans de 0.7. Din acest motiv nu le vom mai aminti si aici. Singura diferenta consta în coborârea parametrului (Noam,HigherThan(_)) la valoarea 4. Acest lucru permite clasificarea corecta a unui nou exemplu pozitiv din sistemele utilizate la reglare. Este vorba de clasa CCurency din sistemul Pyramid-Atm, care este un exemplu tipic de DataClass. În acelasi timp se introduc alte 3 fals-pozitive. Nu vom mai discuta contradictia ce apare între exemplul pozitiv si cel negativ deoarece ea e similara cu cea discutata în paragraful anterior legata de exemplul negativ GetPinPanel.

Este însa foarte important sa amintim ca apare o îmbunatatire a capacitatii de detectie a strategiei peste sistemul de validare. Clasa EmployeeComm din acest sistem, un caz tipic de DataClass, nu mai apare ca fals-negativ. Din punct de vedere practic, daca acceptam sa apara situati fals-pozitive de genul celor discutate, am putea utiliza o solutie de parametrizare furnizata la acest balans în vederea detectiei

carentelor de proiectare DataClass si în cazul altor sisteme software. Oricare din solutiile date de masina prezinta o capacitate de detectie mai buna decât strategia cu parametrii initiali. Totusi, variatia puternica a pragului membrului (Woc,LowerThan(_)) nu prea ne permite sa facem acest lucru. Mai mult, fals-negativul Card din cadrul sistemului 3AM-Atm utilizat ca exemplu de regalare a parametrilor, este un exemplu de DataClass ce ar trebui detectat. Vom încerca eliminarea acestui fals-negativ marind si mai mult balansul.

Solutiile de parametrizare obtinute la un balans de 0.9

Solutiile de parametrizare obtinute la acest balans prezinta un fals-negativ si 19 fals-pozitive. Eliminarea unei situatii fals-negative a condus la introducerea unui numar de 8 fals-pozitive în plus. Nu vom mai discuta contradictiile dintre exemplul pozitiv si cele negative, întrucât cele din urma sunt similare exemplului negativ GetPinPanel.

Din punctul de vedere al parametrilor, se constata coborârea pragului membrului (Noam,HigherThan(_)) la valoarea 3 lucru ce permite detectia exemplului pozitiv Card. Lucrul cel mai important însa este buna stabilizarea a valorii tuturor parametrilor de la o rulare la alta. Cel mai important este ca parametrul membrului (Woc,LowerThan(_)) devine mult mai stabil decât în cazurile anterior expuse. Aceasta stabilizare era de asteptat din moment ce contradictiile de care era raspunzator au fost acceptate. Celelalte exemple negative din setul de reglare reusesc sa constrânga suficient de bine acest parametru.

Cum în sistemul de validare a aparut o îmbunatatire a capacitatii de detectie a strategiei fata de situatia initiala putem spune ca am obtinut o solutie de parametrizare mai buna. Evident, putem spune acest lucru, doar daca acceptam indicarea ca suspecti de catre strategie si a exemplelor negative imposibil de satisfacut. Prin definitie, o strategie de detectie indica suspecti. Este necesara o analiza manuala a acestora în vederea identificarii carentelor reale. Totusi, suspectarea unor clase ca fiind DataClass-uri desi ele nu sunt, este mult mai putin grava decât scaparea detectiei unei carente de acest tip. În concluzie putem sa acceptam o solutie de parametrizare furnizata de masina de tuning la un balans de 0.9.

Înainte de a evalua îmbunatatirile trebuie sa mentionam urmatoarele. Ultimul fals-negativ din sistemele de tuning nu poate fi eliminat. El prezinta un Woc de valoare 1. S-ar putea spune ca avem un zgomot în setul de exemple. Totusi, o analiza manuala la nivel de cod a aratat ca nu avem o astfel de situatie, fiind într-adevar vorba de un DataClass. Problema este alta: exista o carenta de implementare a calculului metricii Woc si Noam. O metoda este considerata de tip accesor doar daca ea este si publica. Din pacate, în sistemele software scrise în Java, mai exista si specificatorul de acces package. Acesta permite accesul la metoda din cadrul pachetului. Totusi, desi metoda este de tip accesor din punctul de vedere al analistului sistemului, ea nu este tratata corespunzator de implementarea celor doua metrici. Aceasta carenta face imposibila detectia corespunzatoare a exemplului pozitiv. Un astfel de exemplu pozitiv exista si în cadrul sistemului de validare.

Ultimul fals-negativ din sistemul de validare, nu este detectat din alt motiv. El prezinta o valoarea 2 pentru Noam. Solutiile suboptimale furnizate la acest balans, arata ca o coborâre a pragului membrului (Noam,HigherThan(_)) la aceasta valoare ar introduce alte fals-pozitive, motiv pentru care masina nu mai coboara respectivul prag. Important de mentionat este si faptul ca o crestere a balansului nu mai poate conduce la o eventuala detectie a respectivului exemplu pozitiv din cadrul sistemului

de validare. In sistemele de reglare nu avem un exemplu pozitiv cu metrica Noam de valoare 2. S-ar putea spune ca setul de exemple utilizat la reglare este incomplet. Totusi, noi consideram ca problema este localizata tot la nivelul scheletului. Fals-negativul din sistemul de validare este un DataClass evident ce extinde un alt DataClass. Cel din urma prezinta un numar mare de metode de tip accesor, dar care nu sunt luate în considerare în momentul calculului Woc si Noam datorita modului de definire al acestor metrici. Prin urmare, ele nu pot surprinde faptul ca exemplul pozitiv mosteneste simptoame mult mai evidente de DataClass de la superclasa sa. Cu alte cuvinte, carenta de proiectare DataClass este o "boala" ereditara.

În final sa analizam performantele solutiilor de parametrizare obtinute la un balans de 0.9. Pentru sistemele utilizate la reglare, cum era de asteptat, avem o precizie de detectie a exemplelor pozitive ridicata: 94%. În acelasi timp 15% dintre exemplele negative sunt identificate ca pozitive. Procentul a crescut fata de situatia initiala pentru ca am acceptat deliberat introducerea unor fals-pozitive pentru o detectie mai buna a exemplelor pozitive. Este de mentionat totusi ca aceasta crestere nu e exagerata. În mod asemanator, pentru sistemul de validare, avem un procent de 50% respectiv 0%. Faptul ca acuratetea de detectie a exemplelor pozitive a crescut în timp ce clasificarea incorecta a exemplelor negative s-a mentinut este încurajatoare. Cu alte cuvinte, este posibil sa utilizam aceste solutii de parametrizare fara sa ne temem de aparitia unui numar impresionant de situatii fals-pozitive.

Din punctul de vedere al relevantei parametrilor trebuie sa mentionam urmatoarele. Era de asteptat ca parametrul lui (Woc,BottomValues(_)) sa fie ridicat. Motivul consta în faptul ca sistemele noastre sunt destul de reduse ca dimensiune. Toate prezinta si o carenta de proiectare GodClass ceea ce implica existenta unui numar mare de clase simple, fara functionalitate, ce apar doar ca suport de date (adica DataClass-uri). Valorile membrilor (Nopa,HigherThan(_)) si (Noam,HigherThan(_)) sunt si ele relevante din punctul de vedere al proiectantului strategiei de detectie, fiind niste valori aproximativ asteptate. Desi am aratat ca membrul (Woc,LowerThan(_)) nu surprinde suficient de bine aspectul de clasa usoara, pragul asociat lui are o valoare destul de relevanta, ceea ce arata ca el surprinde în parte aspectul vizat. Daca acceptam aparitiile situatilor fals-pozitive discutate în aceasta sectiune, considerarea unei clase usoare ca având metrica Woc mai mica decât 0.5 – 0.6 este acceptabila.

Propuneri de îmbunatatire a scheletului strategiei

În continuare vom prezenta câteva propuneri de îmbunatatire a scheletului strategiei de detectie DataClasses care, din punct de vedere intuitiv, ar trebui sa rezolve situatiile contradictorii întâlnite pe parcursul acestei expuneri si sa asigure în ansamblu o capacitate de detectie mai buna prin reducerea situatiilor fals-pozitive întâlnite.

În general, scheletul strategiei de detectie trebuie sa surprinda mai bine aspectul de clasa usoara întâlnit în contextul carentei de proiectare DataClass. Dupa cum am aratat, metrica Woc (desi se numeste sugestiv greutatea clasei) nu poate cuantifica singura acest aspect. Un prim motiv consta în existenta unor clase care au multe atribute publice sau multe metode de tip accesor în interfetele lor, dar în acelasi timp si un numar de metode obisnuite, non-accesor, ce confera clasei multa functionalitate. Astfel de clase nu trebuie considerate usoare. Metrica Woc este definita ca numarul de metode non-accesor din interfata clasei raportat la numarul total de membrii din interfata. Daca numarul total de membrii din interfata e mare (în cazul nostru multe metode de tip accesor si/sau multe atribute publice) si numarul de

metode non-accesor e mic, atunci, oricât de complexe ar fi aceste metode, Woc va avea o valoare mica. Prin urmare, membrul (Woc,LowerThan(_)) trebuie ajutat cumva pentru a cuantifica într-un mod corect aspectul de clasa usoara.

Pentru evitarea situatiei mai sus prezentate, propunem utilizarea în conjunctie cu membrul (Woc,LowerThan(_)) a unui alt membru: (Amw,LowerThan(_)). Amw (Average method weight) cuantifica complexitatea medie a metodelor unei clase. Ca metrica de complexitate a unei metode s-ar putea utiliza fie complexitatea ciclomatica fie numarul de linii de cod asociate metodei masurate. În contextul situatiei mai sus amintite, functionalitatea adusa clasei de un numar mic de metode ar fi neglijata doar daca functionalitatea medie a unei metode din clasa respective este redusa.

O problema posibila a acestei abordari consta în avantajarea detectiei DataClass-urilor ce au metode de tip accesor, în timp ce DataClass-urile ce au câmpuri publice ar putea sa nu fie detectate. Strategia ar putea fi pacalita prin utilizarea câmpurilor publice si renuntarea la metodele accesor asociate lor, crescând astfel complexitatea medie a metodelor. Totusi, aceasta situatie poate fi evitata prin definirea unei metrici noi AMWNAM (Average method weight on non-accesor methods) ce sa fie utilizata în locul lui Amw. În aceasta situatie, functionalitatea adusa clasei de un numar mic de metode non-accesor ar fi neglijata doar daca functionalitatea medie a acestor metode este redusa.

O analiza asupra codului situatiilor fals-pozitive întâlnite pe parcursul acestui experiment arata o alta carenta a scheletului strategiei DataClasses: multe exemple negative reprezinta clase ce au doar metode de tip accesor, parând astfel a fi clase de date, dar care au multa "functionalitate" mostenita de la superclasa lor. Observatiile noastre arata ca aceasta situatie apare în principiu la interfata modulului "business logic" cu modulul "presentation" (user interface). Generalizând, o astfel de situatie poate apare si la interfata modulului "business logic" cu modulul "data base". De exemplu, în tehnologia EJB, superclasa corespunzatoare unui bean dicteaza interfata subclasei impunând implementarea unor metode de tip accesor. Astfel de clase nu ar trebui detectate ca clase de date. Scheletul actual al strategiei nu poate însa elimina astfel de situatii. O varianta simpla de rafinare a scheletului ar consta în introducerea unui membru bazat pe metrica DIT (Depth in inheritance tree) prin care sa se impuna ideea ca un DataClass nu poate extinde o alta clasa. Aceasta abordare este însa mult prea specifica. Dupa cum am vazut, exista situatii în care un DataClass extinde un alt DataClass (vezi discutia de la solutiile de parametrizare obtinute la un balans de 0.9). O rafinare mai sigura ar fi sa spunem ca un DataClass nu extinde niciodata o clasa dintr-o librarie. Abstract, o solutie generala mult mai eleganta ar fi o abordare recursiva a problemei: o clasa poate fi un DataClass numai daca nu extinde nici o clasa, iar daca extinde totusi una sau mai multe, toate acestea trebuie sa fie la rândul lor DataClass-uri. Realizarea unui astfel de schelet este însa extrem de dificila, poate chiar imposibila, în special datorita faptului ca limbajul de specificare al strategiilor de detectie nu permite actualmente o astfel de reprezentare. O alta dificultate este aceea ca noi nu stim ce clase din librarie sunt clase de date.

Totusi, ideea merita a fi analizata în continuare deoarece, dupa cum am vazut, noi am gasit exemple pozitive de clase afectate de carenta DataClass care extind alte clase afectate de aceeasi carenta. În astfel de situatii, subclasa are un numar foarte mic de metode de tip accesor. Pentru detectia lor, ar trebui, în principiu, ca pragul membrului (Noam,HigherThan(_)) sa fie si el mic. Acest lucru ar putea cauza aparitia situatiilor fals-pozitiv, membrul devenind prea general. Tinând însa cont si de metodele mostenite de la superclasa constatam ca subclasa prezinta în realitate multe metode accesor, care nu sunt luate în considerare de nici o metrica utilizata în

scheletul actual. În mod asemanator, aceeasi problema poate apare si pentru atributele publice. În vederea solutionarii acestei deficiente ar fi posibil sa redefinim anumite metrici astfel încât ele sa surprinda si aspectul mostenirii. Totusi, daca am tine cont în mod direct de faptul ca superclasa este un DataClass, am putea obtine un schelet mai elegant în care valorile pragurilor asociate metricilor Noam si Nopa nu ar generaliza prea mult strategia de detectie.

Capitolul VI

Concluzii si perspective

6.1 Concluzii

Stabilirea adecvata a valorilor pragurilor filtrelor de date este o problema sensibila în contextul definirii unei strategii de detectie. Aceste valori reprezinta inima mecanismului de detectie întrucât pe baza acestor valori se stabileste daca o entitate de design prezinta sau nu simptoamele carentei de proiectare vizata de strategia de detectie aplicata. Utilizarea unor valori inadecvate pentru aceste praguri poate conduce atât la ratarea detectie unor carente reale cât si la introducerea unui numar inacceptabil de situatii în care sunt suspectate fragmente de design ce nu sunt conforme cu descrierea informala a carentei de proiectare vizata de strategia de detectie utilizata. Cu alte cuvinte, asa cum se recunoaste si în [14], calitatea unei strategii de detectie depinde puternic de stabilirea corespunzatoare a valorilor pragurilor filtrelor de date utilizate în respectiva strategie.

Masina de tuning a strategiilor de detectie s-a dovedit un instrument util în vederea rezolvarii problemei pragurilor. Experimentele au aratat ca este posibila o îmbunatatire a capacitati de detectie a strategiilor curente, daca pragurile valorice asociate filtrelor lor de date sunt stabilite de masina de tuning. În cadrul experimentului de parametrizare a strategiei de detectie GodClasses s-a demonstrat ca valorile pragurilor furnizate de masina asigura o mult mai buna detectie a carentelor reale cât si o reducere de ansamblu numarului de entitati de design suspectate în mod eronat.

Pe de alta parte, utilizarea masinii de tuning a facut posibila identificarea usoara a unor situatii de incompletitudine a scheletului strategiei de detectie supusa procesului de parametrizare. Astfel de situatii au aparut în cadrul experimentului de parametrizare a strategiei de detectie DataClasses. Utilizând facilitatea de balans a masinii de tuning a fost posibila determinarea rapida a unor exemple pozitive si negative corespunzatoare carentei de proiectare DataClass care nu pot fi satisfacute simultan daca utilizam scheletul de strategie prezentat în [14]. O analiza manuala la nivelul codului sursa asociat acestor exemple a condus la identificarea rapida a carentelor scheletului strategiei de detectie si la gasirea simpla a posibilitatilor de corectare a acestor carente.

Este important de remarcat ca experimentul de parametrizare a strategiei de detectie DataClasses a permis nu doar identificarea unor probleme de cuantificare a descrierii informale corespunzatoare acestei carente ci si obtinerea unor praguri valorice pentru filtrele de date ce asigura o capacitate de detectie mai buna. Astfel, masina de tuning a reusit sa gaseasca solutii de parametrizare care asigura detectia multor carente reale de tip DataClass. Dupa cum am precizat de nenumarate ori pe parcursul acestei lucrari, din punctul de vedere al unei persoane ce cauta carentele de proiectare dintr-un sistem software orientat pe obiecte de mari dimensiuni, este foarte important ca strategia sa gaseasca toate carentele de proiectare pe care spune ca le detecteaza. Suspectarea eronata a unui numar redus de entitati de design nu reprezinta o problema prea grava în momentul în care toate carentele reale sunt detectate. În acest sens a fost necesara acceptarea unui compromis: dorim detectia unui numar mai

mare de carente reale chiar daca acest lucru conduce la cresterea numarului de entitati de design suspectate eronat. Masina de tuning a fost informata de acest compromis prin facilitatea de balans. Solutiile furnizate în aceasta situatie conduc la o crestere puternica a numarului de carente reale detectate. În schimb, apar si destul de multe entitati suspectate în mod fals. În conditiile în care acceptam aparitia situatiilor fals-pozitive, care nu pot fi eliminate datorita incompletitudinii scheletului strategiei de detectie, putem spune ca în principiu avem o strategie mai buna.

6.2 Sumarul contributiei

În aceasta lucrare am propus o metoda clara de stabilire a pragurilor filtrelor de date din cadrul unei strategii de detectie si am demonstrat utilitatea ei. Metoda este una generala, care nu depinde de o anume strategie de detectie sau de o anumita carenta de proiectare. Conceptul de masina de tuning este destinat unei implementari automate si ce este foarte important, asigura repetabilitatea operatiei de parametrizare.
Alte caracteristici importante ale metodei sunt:

- Putem utiliza un numar foarte mare de exemple corespunzatoare unei carente datorita posibilitatii de automatizare a metodei de parametrizare. Daca metoda nu s-ar preta implementarii automate setul de exemple analizate în vederea stabilirii pragurilor valorice nu ar putea fi unul prea mare, iar metoda ar fi costisitoare ca timp.

- La construirea unui set mare de exemple pot participa multe persoane. În consecinta, exemplele vor reflecta perceptia corespunzatoare mai multor persoane asupra carentei ce se doreste a fi detectata. Este posibila astfel reducerea influentei subiectivismului în selectia exemplelor. Subiectivismul apare inevitabil la nivelul interpretarii descrierii informale corespunzatoare unei carente. Mai grav este atunci când persoana ce selecteaza un exemplu corespunzator unei carente cunoaste modul concret de detectie a respectivei carente. Ea poate fi influentata sa clasifice gresit acel exemplu pentru simplul motiv ca el nu se potriveste regulii cuantificate corespunzatoare acelei carente.

- Datorita posibilitatii utilizarii unui set de exemple de mari dimensiuni este mult mai probabil ca acesta sa surprinda marea majoritate a particularitatilor carentei careia îi este destinat.

6.3 Perspective

În perioada imediat urmatoare, eforturile noastre se vor concentra în doua directii esentiale. Pe de-o parte dorim sa efectuam experimente de parametrizare mai puternice atât pentru strategiile GodClasses si DataClasses cât si pentru alte strategii de detectie. Pe de alta parte vom trece la rafinarea implementarii conceptului de masina de tuning, a prototipului realizat si prezentat în aceasta lucrare. În paralel cu cele doua linii de dezvoltare ale ideii noastre vom efectua si rafinari ale conceptului de masina de tuning daca vom considera ca este necesar.

6.3.1 Alte experimente de evaluare

Scopul acestor noi experimente va urmarii doua aspecte:

- Optimizarea detectiei carentelor de proiectare. În acest context, dorim sa parametrizam diferite strategii de detectie utilizând masina de tuning. Acolo unde situatia o va impune, vom determina carente aparute la nivelul scheletului unei strategii si vom cauta solutii de corectie pe care le vom realiza efectiv.

- Problema dependentei pragurilor. În cadrul experimentelor prezentate în lucrare au fost folosite exemple provenite din sisteme de dimensiuni similare si reduse. Este aproape sigur ca pragurile filtrelor de date depind de dimensiunea acestor sisteme. Pentru a analiza impactul pe care îl are dimensiunea sistemelor asupra valorii pragurilor vom utiliza la reglarea parametrica exemple provenite din sisteme de dimensiuni similare si în acelasi timp mai mari. Mai nou, a aparut ideea dependentei pragurilor si de limbajul de programare corespunzator exemplelor. Dorim sa analizam si acest aspect folosind doar exemple scrise în acelasi limbaj. În final am putea obtine un catalog de strategii de detectie în cadrul caruia fiecare "familie" de sisteme software orientate pe obiecte sa aiba praguri valorice specifice si de ce nu si schelete de strategie usor modificate.

În vederea executiei unei reglari parametrice pentru o strategie de detectie trebuie construit mai întâi un set mare de exemple pozitive si negative corespunzatoare. Pentru colectarea de exemple provenite din sisteme software de dimensiuni reduse procedura de selectie va fi similara celei descrise în capitolul anterior. Detinem codul unui numar mare de sisteme software care vor putea fi analizate manual de diferite persoane în vederea identificarii carentelor reale. Problema esentiala cu care ne confruntam consta în faptul ca nu dispunem de sisteme software de mari dimensiuni din care sa preluam exemple. Astfel de sisteme putem gasi în special în industrie. Problema nu rezida în identificarea carentelor dintr-un astfel de sistem, deoarece persoanele ce-l întretin pot identifica usor diverse carente. Mai mult, daca sistemul a fost supus unui proces de reengineering carentele reale sunt deja cunoscute. Astfel, versiunile mai vechi ale sistemului pot fi surse de exemple de carente. Singurul obstacol ce trebuie depasit pentru a le putea folosi consta în obtinerea codului sursa din partea proprietarului. În principiu, pe noi ne-ar interesa doar meta-modelul sistemului ce permite calcularea diverselor metrici software si nu contine cod sursa. Din pacate, în situatii de incompletitudine a scheletului unei strategii ar putea fi necesara si inspectarea codului. Obtinerea lui ar putea fi foarte dificila. Pentru traversarea acestui impas vom înzestra implementarea conceptului de masina de tuning cu o facilitate speciala despre care vom vorbi mai târziu. Evident, astfel de probleme nu apar daca codul sursa este public (si exista suficient cod public pe internet). Mai mult, nu excludem nici posibilitatea colaborarii cu universitati din strainatate în vederea obtinerii de exemple.

6.3.2 Rafinarea implementarii

În ceea ce priveste rafinarea implementarii conceptului de masina de tuning dorim ca în perioada urmatoare sa analizam si alte alternative referitoare la algoritmul de tuning. Trebuie observat ca ideea de "tuning machine" nu este dependenta de utilizarea algoritmului genetic folosit în prototipul LRG-DSTM. Din acest motiv dorim sa analizam impactul pe care îl are asupra rezultatelor utilizarea altor posibili algoritmi de tuning. Evident, putem utiliza diverse variatii de algoritmi genetici [23] sau ideea de strategie evolutiva [23]. O alta alternativa ar putea consta în utilizarea unei retele neuronale care sa "învete" valorile pragurilor filtrelor de date.

O alta rafinare necesara relativ la prototipul nostru consta în scaderea timpului de executie al algoritmului de tuning. Momentan, în vederea evaluarii la nivel conceptual a masinii de tuning, timpii de executie sunt perfect rezonabili. În timp, pe masura ce acest concept va ajunge la maturitate, scaderea timpului de executie va deveni o necesitate. Unii ar putea spune ca simpla eliminare a utilizarii serverului de baze de date de la calcularea calitatii unei solutii de parametrizare potentiale va conduce la timpi de executie mai buni. Totusi, amintim ca masina de tuning va utiliza multe exemple în vederea stabilirii pragurilor valorice pentru o strategie de detectie. Prin urmare pot apare atât probleme cauzate de spatiul de memorie limitat cât si probleme de timp datorate numarului mare de exemple ce trebuie analizate. În vederea reducerii timpului de executie va fi necesara utilizarea unor structuri de date puternice care sa permita analiza rapida a setului de exemple. Analiza rapida este necesara pentru ca întregul set de exemple este analizat în momentul calcularii calitatii unei solutii potentiale de parametrizare (functia de fitness). Evident, aceasta discutie are sens numai daca prototipul LRG-DSTM va pastra algoritmul de tuning curent. Daca acest algoritm se va schimba este posibil ca modul de îmbunatatire a timpului de executie sa difere.

O alta rafinare importanta a implementarii conceptului va fi descrisa în continuare. Actualmente, un exemplu este reprezentat din numele entitatii de design (metoda, clasa sau pachet) si valorile diferitelor metrici software calculate peste ea. Din motivele prezentate în capitolul anterior (calcularea parametrilor filtrelor de tip TopValues si BottomValues), este necesar ca în depozitul de exemple al prototipului sa fie prezente toate entitatile de design similare dintr-un sistem, ca exemple negative ori pozitive dupa cum respectivele entitati sunt afectate sau nu de carenta de proiectare vizata. Aceasta cerinta nu este nesatisfacatoare din punctul de vedere al utilizatorului masinii. Încarcarea exemplelor se poate face complet automat si rapid. Mai mult, furnizorul de exemple (proprietarul sistemului din care provin exemplele) poate accepta sa furnizeze meta-modelul sistemului în vederea calcularii tuturor metricilor necesare si introducerea exemplelor în depozit. Problema este însa alta: în cazul unor carente la nivelul scheletului unei strategii va fi necesar sa analizam cod sursa pentru a identifica respectivele carente. Simpla analiza a valorilor metricilor asociate exemplelor contradictorii în raport cu respectivul schelet nu ne poate informa despre modul concret în care este construit acel exemplu. Prin urmare avem nevoie de codul sursa. Este putin probabil ca proprietarul sistemului sa ne furnizeze întregul cod al sistemului. Este adevarat ca la cerere el ne-ar putea da doar fragmentul interesant din punctul nostru de vedere. Pentru a nu cere în mod repetat codul diferitelor fragmente noi propunem o alta abordare. Un furnizor de exemple oarecare ne va da direct codul asociat unui fragment de design corespunzator unei exemplu de carenta de proiectare. În acelasi timp va furniza un descriptor ce va contine toate informatiile necesare utilizarii lui de catre masina de tuning. Acesta este motivul pentru care la

nivel de meta-arhitectura au fost definite conceptele de mostra de date si descriptor de exemplu. Mai multe mostre de date reconstituie fragmentul de design furnizat (care ar putea fi spre exemplu o structura de clase necesara întelegerii unei carente si calcularii metricilor software necesare si nu doar o clasa), iar descriptorul contine alte informatii necesare. Aceste notiuni vor fi necesare si în alte situatii. Este foarte posibil sa apara strategii de detectie pentru carente de proiectare ce apar datorita colaborarii defectoase dintre diferite entitati de design. În acest caz reprezentarea unui exemplu este mai dificila pentru ca el nu reprezinta doar o clasa, metoda ori pachet ca în prezent. Notiunea de mostra de date si descriptorul de exemplu vor putea fi utilizati pentru rezolvarea acestei probleme.

O ultima rafinare pe care dorim sa o realizam consta în modificarea prototipului astfel încât acesta sa primeasca la intrare un fisier ce sa contina scheletul strategiei de detectie ce trebuie parametrizat specificat într-un anumit limbaj. Limbajul curent de specificare a unei strategii este SOD. Noi nu am utilizat acest limbaj în vederea specificarii scheletului pentru simplul motiv ca într-un viitor foarte apropiat limbajul SOD va fi înlocuit de limbajul SAIL. Acesta va permite specificarea unor strategii de detectie mai complexe ce nu pot fi reprezentate prin SOD. Prin urmare, masina de tuning va trebui sa accepte limbajul SAIL ca limbaj de specificare a scheletului strategiei de detectie ce urmeaza a fi parametrizata.

6.3.3 Alte idei

Dupa cum se specifica în [14], conceptul de strategie de detectie ar putea fi utilizat si în alte scopuri, de exemplu detectia utilizarii anumitor tipare de proiectare. Personal sunt de parere ca acest concept poate fi utilizat si în afara domeniului ingineriei software. Generalizând, conceptul ar putea fi definit astfel: o strategie de detectie reprezinta o expresie cuantificata a unei reguli prin care pot fi detectate dintr-o multime de entitati acelea ce sunt conforme cu respective regula. Pentru a putea utiliza strategiile de detectie si în alte domenii de activitate este necesara în primul rând existenta unei metode clare de parametrizare a filtrelor de date ce sa poata fi utilizata acolo unde nu exista alte metode de stabilire a pragurilor specifice domeniului. Metoda propusa în aceasta lucrare este generala si independenta de domeniul de activitate propriu-zis lucru ce subliniaza înca o data, daca mai era necesar, importanta conceptului de masina de tuning a strategiilor de detectie.

Bibliografie

1. E.V. Berard. ***Abstraction, Encapsulation and Information Hiding.*** http://www.toa.com/pub/abstraction.txt, 2002.

2. G. Booch.***Object-Oriented Analysis and Design with Applications.*** Benjamin Cummings, Redwood City, 2nd edition, 1994.

3. F.P. Brooks Jr. ***No Silver Bullet. Essence and Accidents of Software Engineering.*** Computer Magazine, April 1987.

4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. ***Pattern Oriented Software Architecture. A System of Patterns.*** John Wiley & Sons, 1996.

5. M. Crisan. ***Învatare automata.*** Note de curs, Universitatea "Politehnica" din Timisoara, 2002-2003.

6. P.J. Denning. ***Genetic Alghorithms.*** American Scientist, Volume 80, January-February, 1992.

7. M.Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts. ***Refactoring. Improving the design of existing code.*** Addison-Wesley, 1999.

8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. ***Design Patterns. Elements of Reusable Object-Oriented Software***. Addison-Wesley, 1994.

9. R. Marinescu. ***A Survey on Object-Oriented Measurement in the Context of Reengineering.*** Phd Report #1, February 2000.

10. R. Marinescu. ***Design Flaws and Detection Strategies in Object-Oriented Software Systems.*** Phd Report #2, June 2001.

11. R. Marinescu. ***Detection Strategies.*** Tehnical Report at FZI Karlsruhe, September 2001.

12. R. Marinescu. ***Ingineria programarii II.*** Note de curs, Universitatea "Politehnica" din Timisoara, 2002-2003.

13. R. Marinescu. ***Measurements and Quality in Object-Oriented Design.*** Phd Report #3, 2002.

14. R. Marinescu. ***Measurements and Quality in Object-Oriented Design.*** Phd Thesis, October 2002.

15. R.C. Martin. ***Agile Software Development***. ***Principles, Patterns and Practices***. Prentice Hall, 2003.

16. R.C. Martin. ***Granularity.*** C++ Report, 1997.

17. R.C. Martin. ***Interface segregation principle.*** C++ Report, 1996.

18. R.C. Martin. ***Open-Close Principle.*** C++ Report, 1996.

19. R.C. Martin. ***Patterns of Learning.*** http://www.objectmentor.com, 2000

20. R.C. Martin. ***Stability.*** C++ Report, 1997.

21. R.C. Martin. ***The dependency inversion principle***, C++ Report,1996.

22. R.C. Martin. ***The Liskov substitution principle.*** C++ Report, 1996.

23. Z. Michalewicz. ***Genetic Algorithms + Data Structures = Evolution Programs.*** Springer, 1996.

24. P. Mihancea. ***The Meta-Architecture of the Detection Strategy Tuning Machine.*** Tehnical Report at "Politehnica" University of Timisoara, 2002.

25. D.L. Parnas. ***On the Criteria To Be Used in Decomposing Systems into Modules***. Communications of the ACM, 15(12), December 1972.

26. D.L. Parnas. ***Software Aging.*** IEEE, 0270-5257/94, 1994.

27. I. A. Rivis. ***Semiologie medicala.*** Volumul I, Ed. Mirton, Timisoara, 1997

28. S.J. Russell, P. Norvig. ***Artificial Intelligence. A Modern Approach.*** Prentice Hall, 1995.

29. M. Shaw, D. Garlan. ***Software Architecture. Perspective on an Emerging Discipline***. Prentice Hall, 1996.

30. I. Sommerville. ***Software engineering.*** Addison-Wesley, 1996.

Anexa A. Catalog cu descrieri informale ale carentelor de proiectare ce apar în sistemele software orientate pe obiecte

Spunem despre o entitate de design ce violeaza un principiu, regula sau euristica de proiectare ca este afectata de o carenta de proiectare. M. Fowler denumeste aceasta situatie "bad-smell" iar R. Martin, într-un mod mult mai general, o denumeste "rotting design". Scopul acestui catalog este de a aduna un set de descrieri informale ce pot fi utilizate la identificarea carentelor de proiectare cu care ne confruntam într-un program. Utilizarea catalogului este simpla: când veti întâlni o entitate de design pe care o banuiti ca e afectata de o carenta de proiectare, începeti sa cautati în acest catalog o descriere informala ce se potriveste cu entitatea de design analizata. Fiecare descriere are asociat un nume corespunzator carentei de proiectare descrise. Numele asociat descrierii informale gasite, identifica problema de proiectare cu care va confruntati.

Înainte de a prezenta catalogul, trebuie specificat ce întelegem printr-o descriere informala. Aceasta reprezinta o descriere necuantificata a unei carente de proiectare care enumera simptoamele respectivei carente sau altfel spus, expune tabloul clinic asociat unei entitati de design afectate de respectiva carenta. Trebuie amintit ca un simptom reprezinta o perceptie subiectiva a unei caracteristici asociate respectivei carente. Astfel, interpretarea descrierii informale poate varia de la o persoana la alta, în special atunci când frontiera dintre "bine" si "rau" nu este foarte clara.

Catalog

A. Carente de proiectare la nivel de subsistem

1. God package

Aplicând în mod strict principiul Common Closure, un pachet tinde sa devina foarte mare si necoeziv. Un alt simptom asociat acestei carente este numarul mare de clienti ai pachetului (ex. clase din alte pachete) ce utilizeaza excesiv pachetul suspectat. De obicei clientii sunt distribuiti în multe alte pachete.

2. Misplaced class

O clasa amplasata într-un alt pachet decât cel în care ar fi normal depinde mai mult de clasele altui pachet decât de cele din pachetul în care se gaseste.

3. Lack of Façade

Interfata unui subsistem consta din clasele accesibile din afara pachetului. Carenta se refera la situatiile în care aceasta interfata este foarte mare, cu multe clase apartinând acestui pachet accesate din afara sa, lucru ce cauzeaza un cuplaj foarte mare între respectivul pachet si restul sistemului software.

B. Carente de proiectare la nivel de clasa

4. Data class

O astfel de carenta apare când o clasa are câmpuri, metode de tip get / set pentru respectivele câmpuri si nimic altceva. Astfel de clase reprezinta containere de date si în mod sigur sunt manipulate prea în detaliu de alte clase. Câmpurile publice reprezinta un alt simptom al carentei.

5. God class

Aceasta carenta este asociata acelor clase care tind sa centralizeze inteligenta sistemului software. Ele executa cea mai mare parte din sarcini, delegând doar detalii minore catre un set de clase triviale si accesând date ce apartin altor clase.

6. Shotgun surgery

Aceasta carenta îsi face simtita prezenta în situatia în care de fiecare data când realizam o modificare într-o clasa, trebuie sa realizam multe modificari marunte într-un numar relativ mare de alte clase. Cu alte cuvinte, carenta apare datorita cuplajului puternic si datorita dispersiei acestuia.

Observatie: Este important de mentionat ca trebuie sa ne concentram asupra schimbarilor datorate instabilitatii clasei.

7. Refused bequest

Carenta se refera la situatia în care o subclasa nu utilizeaza membrii apartinând ascendentilor sai desi acestia au fost special proiectati pentru a fi utilizati de descendenti. Daca un descendent refuza doar implementarea mostenita atunci lucrurile ar putea sa nu fie chiar atât de grave. Problema e mai mare atunci când subclasa reutilizeaza comportament mostenit, dar nu vrea sa suporte interfata superclasei.

8. ISP violation

Violarea principiului segregarii interfetei apare atunci când o clasa are o interfata mare dedicata tuturor clientilor sai (numerosi) în loc de mai multe interfete specifice diferitelor categorii de clienti.

C. Carente de proiectare la nivel de metoda

9. Feature envy

O metoda sufera de aceasta carenta de proiectare când ea pare a fi mai interesata de o alta clasa decât de cea careia îi apartine. Interesul se refera de cele mai multe ori la date. Astfel de metode acceseaza în mod direct ori prin intermediul metodelor de tip accesor multe date apartinând altor clase si nu celei de care apartine metoda.

10. God method

O metoda afectata de aceasta carenta de proiectare tinde sa centralizeze inteligenta clasei de care apartine. Simptoamele tipice sunt: dimensiune mare a metodei, lista lunga de parametrii precum si prezenta numeroasa a instructiunilor de tip if / else.

D. Carente de proiectare la nivel de micro-arhitectura

11. Lack of bridge

Carenta apare atunci când o abstractiune si implementarile sale trebuie sa fie capabile sa varieze independent, dar se utilizeaza mecanismul de mostenire pentru a crea diferite implementari ale aceleiasi abstractiuni: o clasa abstracta defineste interfata iar subclasele concrete furnizeaza implementare. Un simptom tipic al acestei carente consta într-o inflatie de clase, o ierarhie de clase foarte bogata în care subclasele se împart în doua categorii: subclase de implementare (clase concrete ce implementeaza interfata superclasei lor) si subclase de specializare (clase abstracte ce reprezinta specializari abstracte ale superclasei lor). Fiecare subclasa de specializare va avea un propriu subset de subclase de implementare.

12. Lack of strategy

Exista doua situatii tipice ce pot apare în situatia în care se ignora utilizarea tiparului de proiectare Strategy. În prima situatie toti algoritmii sunt încapsulati într-o singura clasa mare. În a doua situatie apare o ierarhie de clase de adâncime mica si latime mare, în care fiecare subclasa ofera o alta implementare a algoritmului. Este foarte posibil ca singura diferenta dintre clasa de baza si subclasele sale sa constea în suprascrierea metodelor ce implementeaza algoritmul.

13. Lack of state

Uzual, starea unui obiect este reprezentata prin atributele sale private. Pentru schimbarea comportamentului sau în raport cu starea sa curenta, se utilizeaza instructiuni conditionale. Când schimbarile comportamentale sunt complexe si apar în multe locuri, dimensiunile si complexitatea metodelor clasei va creste foarte mult, iar lipsa aplicarii tiparului de proiectare State îsi face simtita prezenta.

14. Lack of visitor

Exista situatii în care dorim sa adaugam noi operatii ce sa se execute peste elementele unei structuri de obiecte. Ignorând solutia furnizata de tiparul de proiectare Visitor, noile operatii vor fi adaugate prin introducerea de metode noi în ierarhia de clase asociate obiectelor. Carenta de proiectare îsi face simtita prezenta când actionam în aceasta maniera pentru o structura de obiecte a caror clase apartin unei ierarhii bogate si stabile.

Anexa B. Solutiile de parametrizare pentru scheletul strategiei de detectie GodClasses

Balans	Nr. rulare	Praguri stabilite de masina de tuning				Fitness	Sisteme de tuning		Sisteme de validare	
		Procent	Atfd	Wmc	Tcc		Fals-negative	Fals-pozitive	Fals-negative	Fals-pozitive
0.5	1	20	2	32	0.13	151	1	2	0	1
		20	2	32	0.15	151	1	2	0	1
	2	19	2	41	0.32	151	1	2	0	1
	3	18	2	34	0.31	151	1	2	0	1
	4	19	2	31	0.11	151	1	2	0	1
		19	2	31	0.02	151	1	2	0	1
		19	2	44	0.32	151	1	2	0	1
		16	2	44	0.28	151	1	2	0	1
	5	17	2	40	0.28	151	1	2	0	1
		17	2	40	0.3	151	1	2	0	1
		17	2	40	0.32	151	1	2	0	1
	6	17	2	35	0.31	151	1	2	0	1
		17	2	32	0.19	151	1	2	0	1
	7	18	2	42	0.29	151	1	2	0	1
		15	2	42	0.29	151	1	2	0	1
	8	18	2	40	0.3	151	1	2	0	1
	9	5	2	149	0.55	201	4	0	0	0
		5	3	93	0.43	201	4	0	0	0
		5	2	74	0.55	201	4	0	0	0
		5	2	127	0.55	201	4	0	0	0
	10	18	2	37	0.29	151	1	2	0	1

Anexa C. Solutiile de parametrizare pentru scheletul strategiei de detectie DataClasses

		Praguri stabilite de masina de tuning					Sisteme de tuning		Sisteme de validare	
Balans	Nr. rulare	Procent	Woc	Nopa	Noam	Fitness	Fals-negative	Fals-pozitive	Fals-negative	Fals-pozitive
0.5	1	53	0.97	5	7	401	6	2	3	0
		53	0.65	5	7	401	6	2	3	0
		53	0.5	5	7	401	6	2	3	0
	2	55	0.59	8	7	401	7	1	3	0
		55	0.92	8	7	401	7	1	3	0
		80	0.36	8	8	401	7	1	3	0
		55	0.88	9	7	401	7	1	3	0
		55	0.95	8	7	401	7	1	3	0
		55	0.95	8	8	401	7	1	3	0
		55	0.89	8	7	401	7	1	3	0
		54	0.42	9	7	401	7	1	3	0
	3	55	0.88	9	8	401	7	1	3	0
		55	0.72	9	8	401	7	1	3	0
		55	0.47	9	8	401	7	1	3	0
	4	55	0.35	9	8	401	7	1	3	0
		55	0.41	9	8	401	7	1	3	0
	5	82	0.35	9	8	401	7	1	3	0
		94	0.35	9	8	401	7	1	3	0
		75	0.35	9	8	401	7	1	3	0
		99	0.35	9	8	401	7	1	3	0
0.55	1	55	0.45	5	7	421	6	2	3	0
		55	0.63	5	7	421	6	2	3	0
		55	0.4	5	7	421	6	2	3	0
		55	0.9	5	7	421	6	2	3	0
		55	0.9	5	8	421	6	2	3	0

		55	0.88	5	8	421	6	2	3	0
		55	0.88	5	7	421	6	2	3	0
	2	53	0.65	5	8	421	6	2	3	0
		53	0.32	5	8	421	6	2	3	0
	3	55	0.56	5	7	421	6	2	3	0
		55	0.73	5	8	421	6	2	3	0
		55	0.95	5	8	421	6	2	3	0
		55	0.44	5	8	421	6	2	3	0
	4	55	0.8	5	8	421	6	2	3	0
		55	0.44	5	7	421	6	2	3	0
	5	54	0.95	5	7	421	6	2	3	0
		54	0.38	5	8	421	6	2	3	0
		54	0.57	5	7	421	6	2	3	0
		54	0.74	5	7	421	6	2	3	0
		54	0.56	5	7	421	6	2	3	0
		54	0.88	5	7	421	6	2	3	0
		54	0.88	5	8	421	6	2	3	0
0.7	1	92	0.49	5	5	451	3	8	3	0
		72	0.49	5	5	451	3	8	3	0
	2	96	0.49	5	5	451	3	8	3	0
		71	0.49	5	5	451	3	8	3	0
		96	0.48	5	5	451	3	8	3	0
		86	0.49	5	5	451	3	8	3	0
		93	0.49	5	5	451	3	8	3	0
		75	0.49	5	5	451	3	8	3	0
	3	95	0.49	5	5	451	3	8	3	0
		92	0.49	5	5	451	3	8	3	0
	4	71	0.95	5	5	451	3	8	3	0
		71	0.84	5	5	451	3	8	3	0
	5	70	0.68	5	5	451	3	8	3	0
0.8	1	85	0.47	5	4	381	2	11	2	0

		88	0.49	5	4	381	2	11	2	0
	2	72	0.48	5	4	381	2	11	2	0
		72	0.71	5	4	381	2	11	2	0
		72	0.88	5	4	381	2	11	2	0
		70	0.94	5	4	381	2	11	2	0
	3	79	0.48	5	4	381	2	11	2	0
		71	0.48	5	4	381	2	11	2	0
	4	69	0.87	5	4	381	2	11	2	0
		70	0.59	5	4	381	2	11	2	0
		72	0.5	5	4	381	2	11	2	0
		70	0.66	5	4	381	2	11	2	0
	5	73	0.91	5	4	381	2	11	2	0
		73	0.65	5	4	381	2	11	2	0
		70	0.79	5	4	381	2	11	2	0
		71	0.57	5	4	381	2	11	2	0
		71	0.67	5	4	381	2	11	2	0
		71	0.6	5	4	381	2	11	2	0
		71	0.92	5	4	381	2	11	2	0
0.9	1	89	0.48	5	3	281	1	19	2	0
		79	0.49	5	3	281	1	19	2	0
		79	0.48	5	3	281	1	19	2	0
	2	96	0.48	5	3	281	1	19	2	0
		80	0.48	5	3	281	1	19	2	0
		76	0.48	5	3	281	1	19	2	0
	3	71	0.51	5	3	281	1	19	2	0
		73	0.48	5	3	281	1	19	2	0
		71	0.5	5	3	281	1	19	2	0
	4	71	0.57	5	3	281	1	19	2	0
		72	0.52	5	3	281	1	19	2	0
	5	71	0.52	5	3	281	1	19	2	0
		71	0.65	5	3	281	1	19	2	0