Universitatea Politehnica din Timişoara
Facultatea de Automatică şi Calculatoare
Departamentul Calculatoare

SherlockJ

Unealtă de depanare statistică a programelor Java

Student:
Cristian-Alexandru Staicu

Conducător ştiinţific:
conf. dr. ing. Marius Minea

Iunie 2011

In an imperfect world with imperfect software, debugging does not end the day software is released.

Ben Liblit

Cuprins

Capitolul 1

Introducere

1.1 Contextul

Depanarea unei bucăţi de software este procesul de identificare şi reducere al numărului de erori conţinute de acel program. Erorile software sunt de cele mai multe ori imposibil de prezis, dar şi de evitat. Cele mai multe dintre ele sunt cauzate de erori umane la nivelul codului sursă sau al proiectării aplicaţiei. Conform unui studiu al International Data Corporation (IDC), efectuat pe 139 de organizaţii americane de dezvoltare de software, un programator consumă o treime din timpul său în procesul de reparare a erorilor [Bal08]. Depanarea este un proces format din doi paşi: P1 - determinarea naturii erorii şi localizarea acesteia, P2 - repararea defectului [MS04]. Cel de-al doilea pas este cu siguranţă o activitate ce trebuie efectuată de către un dezvoltator uman, deoarece implică capacitatea de rezolvare de probleme, de abstractizare şi de raţionare, capabilităţi de care nu dispune nici o maşină de calcul existentă. Cât despre primul pas, acela de localizare al erorilor, în ultimii ani se fac eforturi susţinute pentru automatizare lui folosind diferite tehnici inspirate de depanarea manuală, clasică.

Principalele tehnici de depanare, aşa cum sunt definite în [MS04], sunt: depanarea prin forţă brută, depanarea prin inducţie, depanarea prin deducţie, depanarea prin revenire şi depanarea prin teste. Prima dintre acestea este cea care prezintă interes pentru lucrarea de faţă, concentrându-ne pe aplicarea unei astfel de tehnici de către o maşină de calcul. Pentru un programator uman aceasta nu este recomandată, ea fiind mare consumatoare de timp.

Odată cu creşterea complexităţii softwareului a crescut şi complexitatea activităţii de depanare, fiind din ce în ce mai laborioasă. Aşa cum se precizează în [Zel01] tehnicile de depanare nu s-au schimbat prea mult în ultimii 50 de ani. Deşi s-au făcut eforturi susţinute pentru automatizarea acestora, de cele mai multe ori nu au reuşit să fie adoptate pe scară largă. Există două mari direcţii de cercetare în domeniu şi anume: automatizarea folosind *modele statice ale codului* şi automatizarea depanării prin *analiză dinamică* a sistemului.

Prima dintre ele presupune creeare de modele pentru codul existent şi analiza acestor modele prin diferite tehnici. Aplicaţii bazate pe această idee (a se vedea Findbugs [FBG11]) au reuşit să se impună în rândul dezvoltatorilor, deşi clasele de erori depistate sunt destul de limitate.

Depanarea prin analiză dinamică presupune rularea aplicaţiei existente şi monitorizarea anumitor parametrii din timpul rulării. Aceşti parametri sunt mai apoi corelaţi pentru a localiza eroarea. Marele beneficiu al tehnicilor de acest tip este penalizarea relativ mică de timp şi spaţiu, dar şi posibilitatea de a efectua o astfel de analiză fără a avea nevoie de modele complicate ale aplicaţiei. Toate acestea au însă un preţ şi

anume *acurateţea diagnozei* destul de scăzută (20% din program trebuie inspectat pentru a localiza eroarea după o astfel de analiză [AZvG07]).

Tehnicile de localizare de erori prin analiză dinamică au fost clasificate în [Pas10] în 6 categorii: analiză de fişiere jurnal, tehnici de depanare interactivă, tehnici de depanare automată, tehnici de tăiere, tehnici bazate pe spectrul program, analiza prin luarea de probe pe "teren" şi detecţia de anomalii.

1.2 Domeniul temei

Depanarea statistică, aşa cum este definită de inventatorul ei - Ben Liblit, presupune folosirea de modele statistice pentru a identifica erori prin corelarea comportamentului rulărilor cu succesul sau eşecul acestora [Lib11]. O astfel de tehnică presupune ca produsul care urmează să fie analizat să fie instrumentat mai întâi, adăugându-i astfel funcţionalitatea de a-şi monitoriza propriul comportament şi de a oferi rapoarte. În lucrarea de faţa, instrumentarea constă în observarea valorilor predicatelor existente în condiţiile din program.

După instrumentare, programul este rulat de un număr oarecare de ori, observând pentru fiecare rulare corectitudinea execuţiei sale, dar şi *spectrul program*. Acesta reprezintă partea de program care fost activă de-a lungul rulării [AZvG07]. Un raport R_i al unui program intrumentat, după o rulare, este format dintr-un bit care indică dacă rularea a fost cu succes sau nu şi un vector ce conţine un bit pentru fiecare predicat monitorizat. Dacă predicatul P este observat ca fiind adevărat măcar o dată de-a lungul rulării $R(P) = 1$, altfel $R(P) = 0$. Un algoritm de depanare statistică poate fi definită astfel:

$$f : \{R_i \mid i = \overline{1, N}\} \to [0, 1] \tag{1.1}$$

Unde N reprezintă numărul de teste, iar R_i reprezintă raportul testului i. Valoarea asociată de funcţie entităţii se numeşte *coeficient de suspiciune* şi reprezintă posibilitatea ca acea entitate să conţină erori.

1.3 Cerinţele proiectului

Lipsa unei aplicaţii de depanarea statistică care să se impună în rândul programatorilor Java m-au împins să dezvolt SherlockJ [SHK11] - un sistem care să integereze cât mai multe tehnici de localizare de erori prin analiză dinamică şi care să se integreze perfect în Eclipse (a se vedea Figura 1.1), cel mai folosit mediu de dezvoltare pentru limbajul Java. Sistemul trebuie să poată fi folosit şi ca o aplicaţie de sine stătătoare, independentă de Eclipse. Trebuie să dispună totodată de o interfaţă cu utilizatorul cât mai intuitivă.

Principala funcţionalitate pe care o va oferi este asocierea unui coeficient de suspiciune fiecărei entităţi din program şi prezentarea acestor coeficienţi utilizatorului într-o manieră în care să permită inspectarea valorilor mai speciale. Sistemul trebuie să preia setările programului analizat definite în mediul de dezvoltare. Astfel, dezvoltatorul nu va trebui să definească dependenţele de bibliotecile externe, căile spre surse sau codul binar al proiectului.

Aplicaţia trebuie să dispună de un modul de intrumentare care să ofere programului capabilitatea de a genera rapoarte în timpul rulării, fără a altera în vreun fel starea proiectului din Eclipse. Instrumentarea trebuie să se facă selectiv, putând configura pachetele / clasele / metodele care să fie monitorizate. De asemenea trebuie să fie configurabil ce se monitorizează din aceste entităţi (comparaţiile, apelurile de metode,

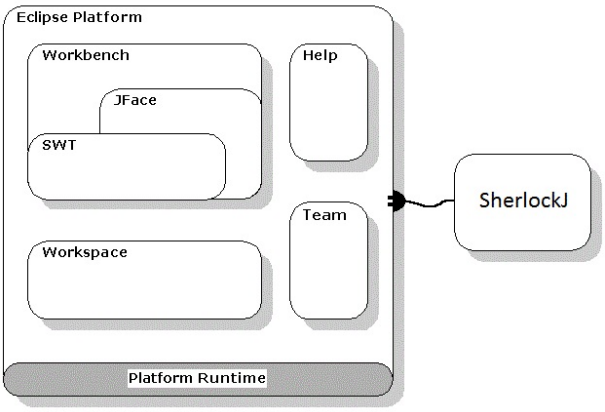


Figura 1.1: Integrare SherlockJ în Eclipse

atribuirile, instanţierile). Intrumentarea trebuie să introducă un balast cât mai mic şi să fie transparentă pentru utilizator. Totodată instrumentarea unei clase trebuie să se facă o singură dată în timpul rulării unei analize.

Testele unitare trebuie să reprezinte scenarile de rulare a analizei dinamice pentru SherlockJ. El va oferi un mecanism de rulare a acestor teste existente deja în proiectul analizat. Trebuie să suporte cadrele de tesare consacrate JUnit [JNT11], TestNG [TNG11] cu toate versiunile majore ale acestora. Rularea testelor trebuie să se facă utilizând programul deja instrumentat, înregistrând rezultatul şi spectrul program pentru fiecare test (R_i). De asemenea trebuie să ofere un mecanism de detecţie automată a testelor şi filtrarea acestora înainte de a începe rularea.

Având în vedere numărul de tehnici de depanare statistică în continuă creştere, SherlockJ trebuie să permită integrarea cât mai facilă a acestora, şi compararea performanţelor diferitelor strategii.

După implementarea sistemului trebuie construit un sistem de evaluare cu ajutorul căruia să se injecteze erori într-un program pentru care dispunem de o suită de teste destul de variată şi de bogată. După injectare i se va cere sistemului să localizeze erorile injectate şi se vor compara rezultatele analizei cu locaţia reală a defectelor injectate. În felul acesta se va demonstra validitatea implementării.

Versiunea de linie de comandă SherlockJ ar trebui să poată accepta fişiere de configurare în care să se seteze parametrii proiectului pentru o mai uşoară utilizare. Proiectul trebuie să nu conţină nici o dependenţă de sistemul de operare pe care a fost dezvoltat, putând fi utilizat pe orice platformă.

Aplicaţia nu trebuie să presupună nimic despre proiectul analizat (număr de fire de execuţie, memorie folosită, conexiune la reţea, apeluri la biblioteci externe). Deci, SherlockJ trebuie să rezolve intern eventualele condiţii de cursă datorate analizării unui proiect ce conţine mai multe fire de execuţie.

Capitolul 2

Fundamentarea teoretică

În acest capitol vom formaliza problema localizării erorilor folosind spectrul program. Vom folosi notaţiile introduse în [LFY+06].

Considerăm o suită de teste $T = \{t_1, t_2, \ldots, t_n\}$ pentru programul analizat P. Fiecare test $t_i = (d_i, o_i)$ este caracterizat de intrarea sa, d_i şi de rezultatul aşteptat, o_i. Rezultatul obţinut efectiv în urma execuţiei pentru testul i este notat cu o_i'. Spunem că programul P a trecut testul t_i dacă şi numai dacă $o_i' = o_i$, altfel spunem că P a picat respectivul test. Astfel suita de teste T este partiţionată în două submulţimi disjuncte:

$$T_p : \{t_i | P(i) = o_i' \text{ şi } o_i = o_i'\} \tag{2.1}$$

$$T_f : \{t_i | P(i) = o_i' \text{ şi } o_i \neq o_i'\} \tag{2.2}$$

Obiectivul principal al unui algoritm de depanare statistică (ADS) este ca având un program P şi o suită de teste $T = T_p \bigcup T_f$ să localizeze regiunile programului ce conţin cel mai probabil erori, prin contrastarea comportamentului lui P din timpul rulării testelor din T_p şi respectiv T_f. Un astfel de algoritm produce un clasament $\tau = ADS(T_p, T_f)$ [Liu06]. Denumim rezultatul funcţiei τ clasament global, deoarece în calcularea funcţiei se iau în considerare toate testele eşuate disponibile pentru programul P. În general, un predicat aflat în partea superioară a clasamentului este foarte probabil să indice o eroare sau vecinătatea acesteia. Fie $\tau(i)$ poziţia în clasament a predicatului P_i. Predicatul P_j se află înainte acestuia în clasament dacă şi numai dacă $\tau(i) \leq \tau(j)$.

Pentru a obţine un clasament τ nu este nevoie să dispunem de T_p şi de T_f în întregime. Orice subseturi ale acestor mulţimi pot produce astfel de clasamente. În cazul extrem, putem contrasta orice test eşuat cu întreaga mulţime de teste corecte:

$$\tau_i = ADS(\{f_i\}, P),\ i = \overline{1, card(T_f)} \tag{2.3}$$

Un astfel de clasament este denumit clasament individual, deoarece consideră o singură execuţie nereuşită la un moment dat.

Observând comportamentul lui P de-a lungul tuturor testelor se poate construi aşa numitul *spectrul program* (SP). Acesta reprezintă un set de date ce caracterizează comportamentul dinamic al rulărilor. El este reprezentata ca o matrice de forma:

$$SP(T) = \left[SP(T_1)|SP(T_2)|\ldots|SP(T_n)\right]^T = \begin{bmatrix} s_{11} & s_{12} & \ldots & s_{1m} \\ s_{21} & s_{22} & \ldots & s_{2m} \\ \ldots & \ldots & \ldots & \ldots \\ s_{n1} & s_{n2} & \ldots & s_{nm} \end{bmatrix} \tag{2.4}$$

n este numărul de teste considerate, iar m este numărul de entităţi luate în calcul. Matricea 2.4 are proprietatea că s_{ij} este 0 atunci când în timpul execuţiei i componenta j nu a fost executată, iar 1 în cazul în care a fost executată.

Mai considerăm un vector de forma:

$$e = \left[e_1|e_2|\dots|e_m\right]^T \tag{2.5}$$

$e_i = 1 \iff T_i \in T_f$. Acesta indică dacă în timpul rulării unui anumit test a apărut o eroare. Putem defini mai exact funcţia τ ca fiind $\tau(SP(T), e)$. Conform [AZvG07], această funcţie se reduce la a calcula similititudinea dintre vectorul e şi fiecare coloană a matricii $SP(T)$.

În domeniul grupării datelor (data clustering) asemănarea a doi vectori binari se face cu ajutorul *coeficienţilor de similaritate*. Aceştia sunt cei care diferenţează tehnicile de depanare statistică existente. În lucrarea de faţă am implementat doi astfel de coeficienţi: Tarantula ([JH05]) şi Jaccard ([CKF+02]). Folosind notaţiile din [AZvG07] aceştia pot fi definiţi astfel:

- **Coeficientul Jaccard** - introdus în 1908, unul dintre cei mai utilizaţi astfel de coeficienţi.

$$\tau_{Jaccard}(i) = \tau_{Jaccard}(col_i(SP(T)), e) = \frac{a_{11}(i)}{a_{11}(i) + a_{10}(i) + a_{01}(i)} \tag{2.6}$$

- **Coeficientul Tarantula** - marele său beneficiu este că permite ca ocazional entităţile ce conţin erori să fie evaluate într-un test corect.

$$\tau_{Tarantula}(i) = \tau_{Tarantula}(col_i(SP(T)), e) = \frac{\frac{a_{11}(i)}{a_{11}(i)+a_{01}(i)}}{\frac{a_{11}(i)}{a_{11}(i)+a_{01}(i)} + \frac{a_{10}(i)}{a_{10}(i)+a_{00}(i)}} \tag{2.7}$$

Coeficienţii de mai sus se definesc pentru o entitate $i \in \{1 \dots m\}$, iar pentru a fi calculaţi avem nevoie de coloana corespunzătoare respectivei entităţi din $SP(T)$ şi de vectorul e. Elementele folosite în definirea coeficienţilor sunt de forma:

$a_{pq}(i) = card\{1 \le j \le n \mid s_{ji} = p \wedge e_j = q\}$

$p, q \in \{0, 1\}$ cu semnificaţia: $s_{ji} = p$ indică dacă entitatea j a fost executată sau nu în testul i, iar $e_j = q$ dacă testul j a fost corect sau eronat.

Capitolul 3

Specificaţiile aplicaţiei

3.1 Schema bloc a sistemului

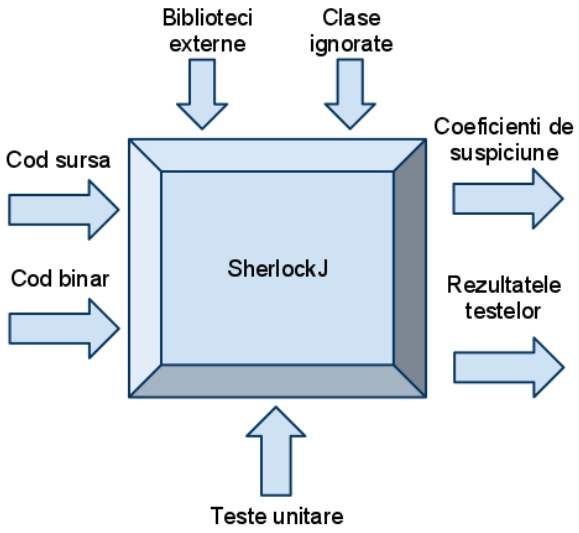


Figura 3.1: Schema bloc a nucleului

Sistemul de faţă analizează rularea anumitor teste unitare încercând să localizeze erorile apărute. Nu are nevoie de un model al codului sursă existent. În figura 3.1 este reprezentată schema bloc a aplicaţiei, iar în continuare voi descrie parametrii de intrare şi de ieşire ai sistemului.

Codul sursă este reprezentat printr-o colecţie de fişiere / biblioteci în care se va căuta eroarea. Acesta este folosit pentru a mapa informaţiile obţinute în urma instrumentării. Pentru o rulare corectă orice clasă a cărei cod binar a fost rulat trebuie să aibă un coresponent în această colecţie. *Codul binar* este furnizat sub forma unui dosar ce conţine totalitatea claselor compilate. Este foarte important faptul că o clasă nu va fi instrumentată decât dacă se află conţinută aici. *Bibliotecile externe* sunt o colecţie de arhive şi dosare ce conţin clase adiacente proiectului, dar care nu vor fi analizate şi instrumentate. Între acestea trebuie neapărat să se afle şi cadrul de testare. *Clasele ignorate* sunt specificate cu numele lor complet şi reprezintă acele clase care au fost anterior specificate ca făcând parte din proiect, dar care nu se doresc a fi analizate şi instrumentate. Astfel de clase sunt de obicei clasele ce conţin teste, care de obicei sunt compilate de către mediul de dezvoltare în acelaşi loc cu restul claselor proiectului, deşi nu reprezintă cod util, de producţie. *Testele unitare* reprezintă

numele complet al unor clase care vor fi pasate cadrului de testare pentru rulare.

Rezultatele testelor reprezezintă o serie de informaţii legate de fiecare test rulat de cadrul de testare: numele clasei de test, numele metodei de test, rezultatul testului, cauza erori în cazul unui test eronat. *Coeficientii de suspiciune* sunt furnizaţi împreună cu o listă de entităţi detectate în timpul instrumentării. Lista de entităţi este ordonată în ordine descrescătoare a suspiciunii.

De precizat ar fi faptul că partea prezentată în Figura 3.1 anterioară este doar nucleul sistemului, care poate fi folosit fără a dispune de o versiune de eclipse instalată. Problema cu aceast mod de utilizare este faptul că trebuie furnizaţi o mulţime de parametri, manual. Trebuie specificate toate intrările, iar ieşirile sunt o listă de obiecte cu date de a căror format utilizatorul trebuie să fie conştient. Pentru o mai uşoară utilizare a fost conceput un plugin de Eclipse care să interfaţeze nucleul cu platforma Eclipse 3.5+. Structura plugin-ului se poate vedea în Figura 3.4.

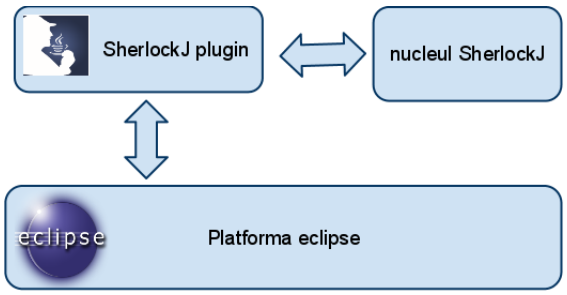


Figura 3.2: Schema bloc a întregului sistem

După cum se observă, parametrii necesari analizei se iau direct din mediul de dezvoltare de această dată, făcând mult mai uşoară utilizare aplicaţiei. Declanşarea analizei, deci comunicarea platformă - plugin - nucleu se face prin selectarea unui proiect java din eclipse şi alegerea din meniul contextual SherlockJ - Find bugs. În acel moment platforma eclipse apelează plugin-ul care mai departe apelează un serviciu din nucleu. După terminarea execuţiei rezultatele sunt preluate de către plugin şi afişate în spaţiul de lucru prin intermediul platformei.

3.2 Funcţiile sistemului

Sistemul prezentat dispune de o singură funcţie principală şi anume localizarea de erori într-un proiect Java, având disponibilă o suită de teste. Această funcţie însă, poate fi customizată în funcţie de dorinţele utilizatorului. El poate alege suita de teste, clasele ignorate, cadrul de testare folosit, metoda de localizare. Aplicaţia mai oferă o funcţionalitate destul de importantă şi anume recalcularea coeficienţilor considerând doar un anumit test eşuat şi toate celelalte corecte.

În jurul aplicaţiei am construit o serie de mini-aplicaţii pentru evaluarea lui SherlockJ care pot fi folosite independent. În Figura 3.3 se pot distinge aceste sisteme şi utilizarea lor. Custom mutator are rolul de a injecta erori în fişiere .class prin modificarea anumitor predicate. Cel de-al doilea sistem auxiliar, Evaluator, are scopul de a rula câte o analiză SherlockJ pentru fiecare clasă mutant dintr-un dosar dat. De remarcat ar fi faptul că aplicaţia Evaluator apare atât ca sistem având propriul scenariu de utilizare, cât şi ca actor.

Deoarece întregul sistem este format din componente foarte bine decuplate ne putem imagina cazuri când bucăţi din cadrul sistemului sunt refolosite în alte proiecte,

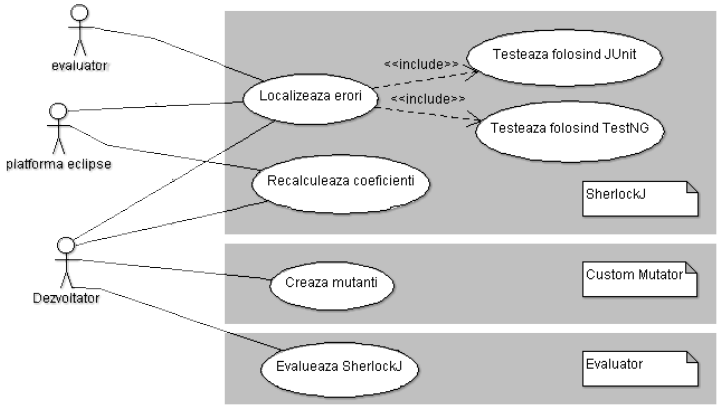


Figura 3.3: Cazuri de utilizare ale sistemului

astfel că fiecare modul expune o interfaţă publică conţinând serviciile pe care le oferă fără a depinde de celelalte module. Există un modul care poate rula o suită de teste, unul care poate instrumenta o clasă dată adăugându-i facilităţi de logging şi altul ce poate analiza o serie de rezultate, prezicând locaţia erorilor.

3.3 Interfaţa cu utilizatorul

3.3.1 Linia de comandă

Nucleul SherlockJ poate fi utilizat din linia de comandă, acesta acceptând argumente de tipul celor folosite de utilitarele Linux. Aplicaţia este distribuită într-o arhivă cu numele SherlockJ_alpha_0.01.jar. Clasa principală este ro.upt.ac.cstaicu.ui.Main şi acceptă următorii parametri:

- -cp = căile spre arhive şi dosare ce conţin clase compilate adiacente proiectului. Dacă suunt mai multe intrări se separă cu ":"
- -pp = calea spre proiect
- -bp = calea spre clasele compilate din proiect
- -tcf = calea spre un fişier de configurare ce conţine pe fiecare rând câte un nume de clasă de test ce va fi rulată în cadrul analizei
- -tf = cadrul de testare (TestNG sau JUnit)
- -flt = tehnica de localizare (Jaccard, Tarantula)
- -of = fişierul de ieşire

În fişierul de ieşire se va afişa tabelul entităţilor suspicioase în formatul csv.

3.3.2 Plugin de eclipse

Interfaţa cu utilizatorul pentru plugin-ul de eclipse trebuie realizată folosind biblioteca SWT şi va adăuga funcţionalităţi în cadrul platformei folosind aşa numitele

puncte de extensie ale eclipse. În documentaţia oficială eclipse[ECH11] acestea sunt definite astfel:

> Punctele de extensie sunt un mecanism prin care se asigură cuplarea slabă în cadrul eclipse. Când un plugin vrea ca anumite bucăţi din funcţionalitatea sa să fie extinse sau personalizate va declara astfel de puncte de extensie. Pentru aceasta se va declara un contract folosind fişiere XML şi interfeţe Java. Contractul trebuie implementat de toate plugin-urile care doresc să se conecteze la acest punct de extensie.

SherlockJ va folosi două astfel de puncte de extensie:

- o intrare într-un **meniu contextual** pentru rularea plugin-ului. Ea va fi afişată doar atunci când se dă click drepta pe un proiect Java
- o **vedere** eclipse pentru afişarea rezultatelor şi inspectarea lor

Tot folosind SWT trebuie implementat şi o fereastră care să îi permită utilizatorului să selecteze testele care vor fi rulate. Acestea vor fi structurate într-o manieră arborescentă, pe pachete şi subpachete.

3.4 Structurarea informaţiei

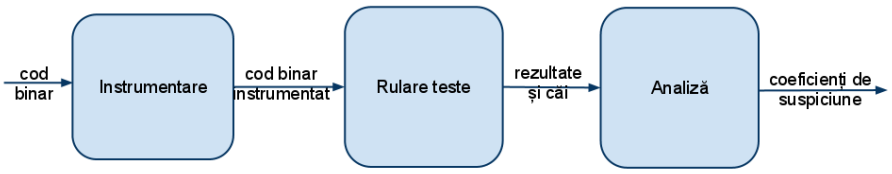


Figura 3.4: Fluxul de informaţii în cadrul sistemului

Fiind o arhitectură de tipul “Pipes & Filters”, fiecare modul face asupra datelor anumite modificări adăugând informaţii despre prelucrarea făcută în interiorul său. Astfel, la instrumentare se creează aşa numitele entităţi. Ele reprezintă abstracţiune principală a aplicaţiei şi modelează o porţiune de cod. O entitate poate fi un predicat dintr-o condiţie, o funcţie, o clasă sau un bloc de cod fără instrucţiuni condiţionate. În interiorul obiectelor de tip entitate este memorat un obiect de tip IContext ce conţine informaţii de localizare în cod (nume clasă, pachet, proiect, linia de cod).

În cadrul modulului de execuţie, fiecărei entităţi i se adaugă informaţii legate de testele ce au evaluat-o. De fiecare dată când un predicat de control corespunzător unei entităţi este evaluat într-un test, acel test este adăugat la lista din interiorul entităţii.

Cu ajutorul acestor informaţii în modulul de analiză se calculează coeficienţii de suspiciune care alături de rezultatelor testelor reprezintă rezultatul unei analize SherlockJ. Coeficienţii de suspiciune se memoreazua alături de entitatea pe care o caracterizează într-un obiect de tipul FaultLocalizationEntity.

În figura 3.5 am structurat pe nivele tipurile de date introduse de fiecare modul. Cel mai de sus nivel este corespunzător modulului de analiză, cel din mijloc modulului de execuţie, iar jos este reprezentat modulul de instrumentare.

Ieşirea unei analize SherlockJ este un obiect SherlockJResult ce conţine o colecţie de obiecte TestResult, câte unul pentru fiecare test şi o listă de obiecte FaultLocalizationEntity ordonate descrescător după coeficientul de suspiciune.

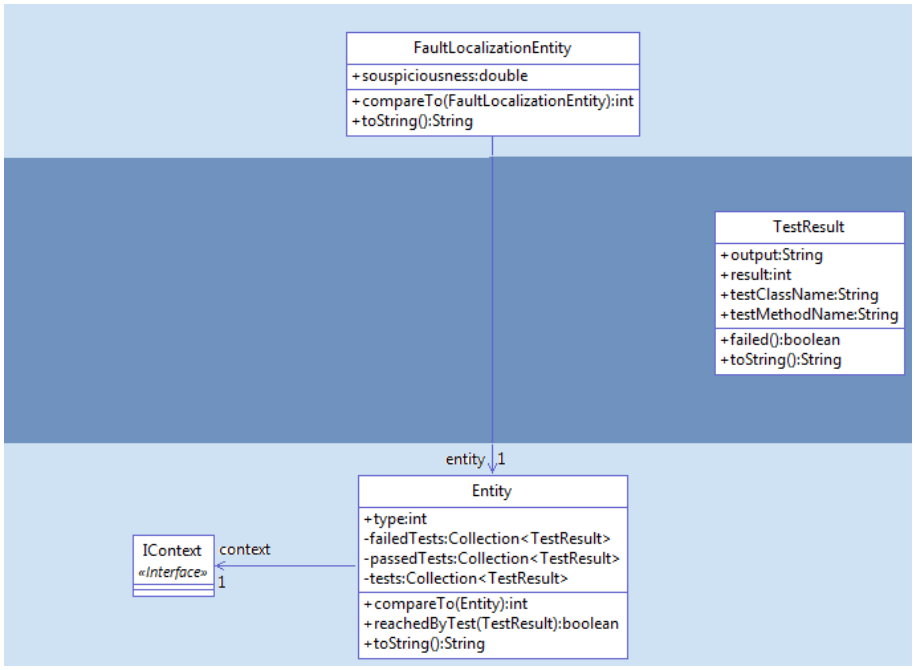


Figura 3.5: Structurile de date utilizate

3.5 Comunicarea cu alte sisteme

SherlockJ comunică intensiv cu platforma eclipse prin mecanismul descris anterior de puncte de extensie, dar şi prin intermediul interfeţei Platform care oferă în special servicii de localizare a altor pluginuri. Comunicarea cu bibliotecile ASM, JUnit şi TestNG va fi descrisă în capitolul legat de implementare. Un alt aspect important ar fi cel legat de sistemul pe care se face analiza folosind SherlockJ. Comunicarea cu acesta este unidirecţională: nu se modifică nimic din starea proiectului, doar se observă rularea codului binar existent.

Capitolul 4

Proiectarea aplicaţiei

4.1 Arhitectura sistemului

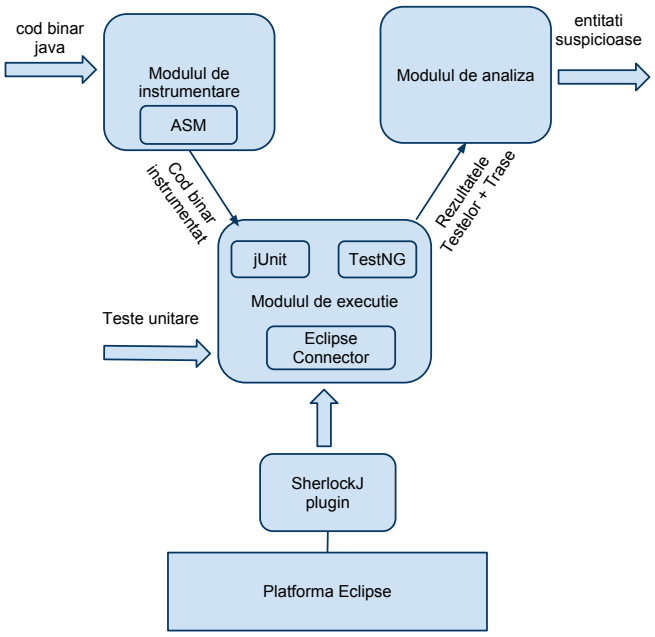


Figura 4.1: Arhitectura nucleului SherlockJ

La proiectarea aplicaţiei am avut ca obiectiv principal flexibilitatea. Modulele sunt foarte slab cuplate, existând câte o interfaţă pentru fiecare modul. Arhitectura nucleului este, după cum se poate observa în Figura 4.1, una de tipul “Pipes and Filters”. Peste nuceleu s-a construit plugin-ul de eclipse care preia direct din eclipse parametrii necesari analizei şi îi pasează nucleului. Legătura dintre plugin şi nucleu se face prin intermediul clasei EclipseConnector din modulul de execuţie care oferă servicii de analiză a unei suite de teste. Aşadar aceasta este o clasă de tip faţadă a nucleului. Modulul de execuţie foloseşte modulul de instrumentare oridecâte ori are nevoie să încarce o clasă în maşina virtuală Java. La terminarea rulării testelor modulul de execuţie îi cere celui de analiză să calculeze coeficienţii de suspiciune după care returnează aceste informaţii plugin-ului care le afişează în mediul de lucru.

4.2 Descrierea componentelor

4.2.1 Modulul de instrumentare

Acest modul are ca principal scop injectarea de predicate de control în interiorul unei clase şi returnarea codului binar al acesteia. Interfaţa Instrumentator din Figura 4.2 este punctul de legătură dintre acest modul şi restul aplicaţiei. După cum se observă, se poate instrumenta o clasă întreagă sau doar anumite metode din aceasta.

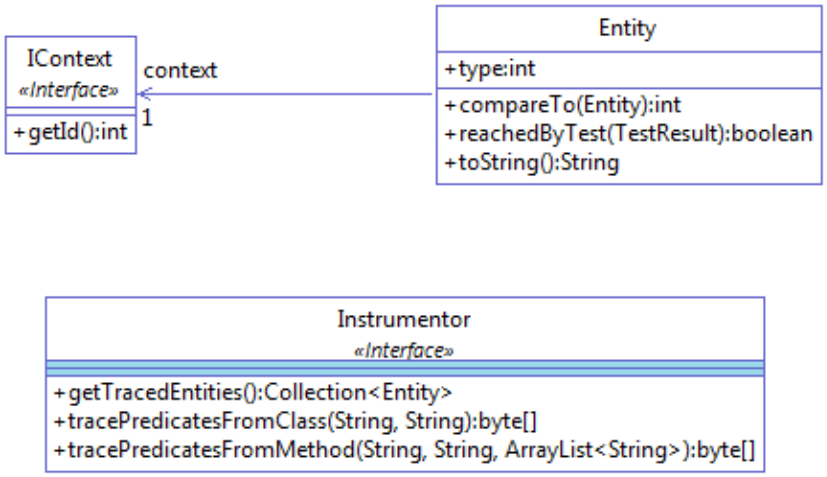


Figura 4.2: Modulul de instrumentare

În procesul de instrumentare sunt identificate şi create obiectele de tipul Entity care reprezintă porţiuni de cod ce pot conţine erori. În funcţie de granularitatea aleasă acestea pot fi predicate, funcţii, clase sau blocuri de bază. Aceste entităţi se stochează, iar la apelarea seriviciului getTracedEntities se returnează toate entităţile identificate. În cadrul acestui modul se injectează aşa numite predicate de control în codul binar al clasei instrumentate. Fiecare astfel de predicat are asociat o entitate. Prin intermediul acestui mecanism se extrage spectrul program în timpul rulării.

4.2.2 Modulul de execuţie

Modulul de execuţie oferă facilitatea de a rula o suită de teste şi de a returna informaţiile referitoare la rezultatul acestora. În interiorul acestui modul se construiesc obiecte de tipul TestResult care se asociază cu entităţiile care au fost evaluate în timpul rulării respectivului test. În astfel de obiecte se reţin informaţii de identificare a testelor, rezultatul lor şi în caz de eroare cauza erorii. Asocierea testelor cu entităţiile se face folosind un GlobalLogger care este apelat de codul instrumentat atunci când apare un predicat de control şi de Executor atunci când începe sau se termină un test.

Modulul de execuţie foloseşte serviciile modulului de instrumentare atunci când doreşte încărcarea unei clase în maşina virtuală. Instrumentarea este comandată folosind CustomClassLoader, în cazul în care clasa este în dosarul de clase binare şi dacă nu face parte din lista claselor ignorate la execuţie. Pentru a folosi serviciile de instrumentare orice instanţă a clasei Executor trebuie să primească în constructor o referinţă spre un Instrumentator.

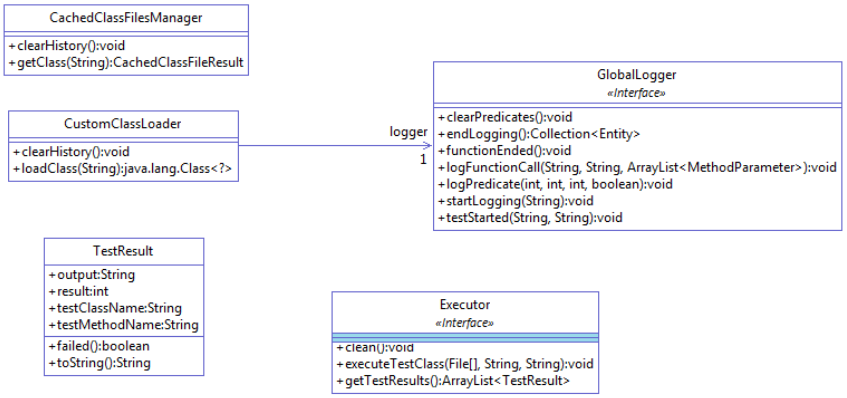


Figura 4.3: Modulul de execuţie

Evitarea instrumentării multiple se face printr-un mecanism de memorare a claselor instrumentate deja. În figura 4.3 se poate vedea marcată cu albastru interfaţa modulului ce este folosită de clienţi pentru a accesa serviciul oferit.

4.2.3 Modulul de analiză

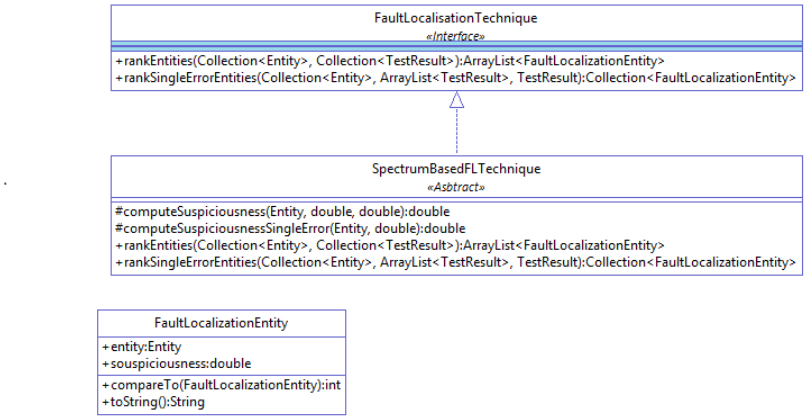


Figura 4.4: Modulul de analiză

Acest modul este cel unde pe viitor se aşteaptă cele mai multe schimbări. Aici se vor implementa toţi algoritmii de localizare de erori având ca intrări valorile predicatelor de control şi rezultatele testelor. FaultLocalizationEntity este o clasă care înglobează o entitate şi coeficientul său de suspiciune. Astfel de obiecte sunt create în acest modul pentru fiecare entitate, după care se ordonează în ordine descrescătoare a suspiciunii.

4.2.4 Pluginul de eclipse

Componentele principale ale acestei părţi din aplicaţie sunt cele din figura 4.5. Cea mai importantă componentă este ContextMenuListener care este apelată oridecâte ori utilizatorul selectează SherlockJ - Find bugs din meniul contextual. Aceasta la rândul ei detectează testele din proiect şi le afişează într-o fereastră (TestFilter) pentru a fi selectate cele care vor intra în procesul de analiză dinamică. Mai apoi se rulează nucleul, iar rezulatele sunt afişate în vederea de eclipse (SherlockView) apelând drawNewRankingTable().

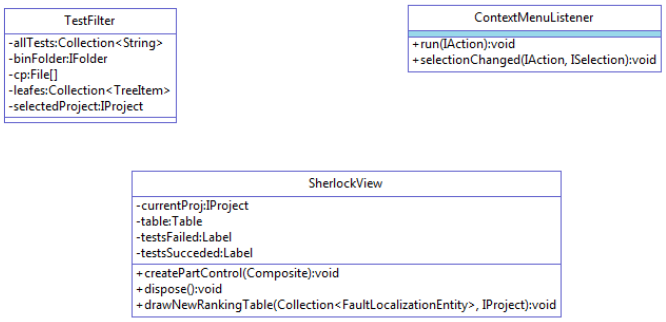


Figura 4.5: Componentele principale ale pluginului

4.3 Descrierea comunicării între module

Comunicarea dintre plugin şi nucleu se face prin intermediul clasei EclipsePlugin care mai apoi are rolul de a construi modulele de care este nevoie pentru analiză, ţinând cont de particularităţile solicitării. Tot clasa EclipsePlugin este cea care preia rezultatele testelor şi spectrul program şi le trimite modulului de analiză. Rezultatele acestuia sunt preluate şi returnate plugin-ului. Comunicarea efectivă se face prin intermediul claselor marcate cu albastru în figurile 4.2, 4.3 şi 4.4. O secvenţă tipică de apeluri derulată în timpul analizei este prezentată în figura 4.6.

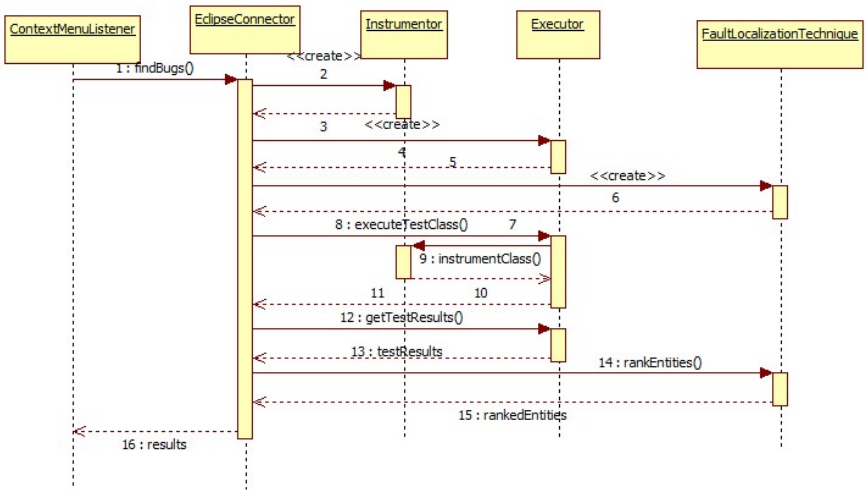


Figura 4.6: Comunicarea în cadrul analizei

Capitolul 5

Implementarea aplicaţiei

5.1 Mediul de lucru.

SherlockJ a fost dezvoltat folosind "Eclipse 3.6 Helios for Java Developers". S-a lucrat la dezvoltarea lui atât pe Linux (Arch Linux) cât şi pe Windows (Windows 7). Pentru versionare am folosit sistemul mercurial (pe Windows - TortoiseHg şi MercurialEclipse, pe Linux - utilitarul hg). Mercurial e un sistem distribuit de versionare uşor de învăţat, dar totodată puternic şi rapid [MER11].

Pentru stocarea sistemului de versionare, dar şi pentru ca proiectul să fie mai vizibil am ales Google Code [SHK11]. Aici se poate vedea evoluţia proiectului şi se poate descărca ultima versiune atât a plugin-ului cât şi a aplicaţiei de linie de comandă. Tot aici se poate găsi documentaţia aplicaţiei, dar şi codul sursă al nucleului.

5.2 Descrierea generală a implementării

În acest capitol se descrie implementarea fiecărui modul în detaliu insistând pe problemele mai sensibilie şi pe motivaţia alegerii implementării curente.

5.2.1 Modulul de instrumentare

Principalele capabilităţi ale acestui modul sunt identificarea entităţilor din program şi inserarea de predicate de control pentru fiecare astfel de entitate. Implementarea actuală permite doar identificarea predicatelor din cadrul instrucţiunilor condiţionale.

Modulul lucrează doar cu codul binar java pe care îl interpretează şi îl instrumentează folosind biblioteca ASM [ASM11]. Am ales această variantă în defavoarea BCEL ([BCE11]) sau SERP ([SER11]), alte biblioteci de acest tip, în principal datorită eficienţei crescute. Alte avantaje sunt simplitatea ASM, lipsa constrângerilor legate de utilizare şi dimensiunea scăzută. Testele efectuate în [EB03] arată că la instrumentarea a 1155 de clase ASM a introdus un balast de 60%, BCEL de 700%, iar SERP de 1100%. Această superioritate este datorată mai ales utilizării tiparului visitor, descris în [GHJV95], într-o manieră ingenioasă.

Biblioteca ASM este organizată în jurul noţiunilor de producători de eventimente (cititoarele de clase), consumatorii de evenimente (obiecte care scriu clasele transformate într-un fişier spre exemplu) şi filtrele de evenimente (obiecte care fac transformări ale codului binar). Ordinea procesării codului binar este dată de modul de legare al componentelor. De fiecare dată când un producător citeşte antetul unei metode, spre exemplu, va apela metoda visitMethod() din cel mai apropiat filtru, care la rândul său după procesare va apela următorul filtru şi tot aşa până când se apelează

un consumator de evenimente care va opri lanţul de apeluri. Aşadar, folosirea ASM implică, conform [Bru07], următorii paşi:

- ansamlarea unei arhitecturi de producători, consumatori şi filtre de evenimente ca cea din figura 5.1
- pornirea producătorilor de evenimente

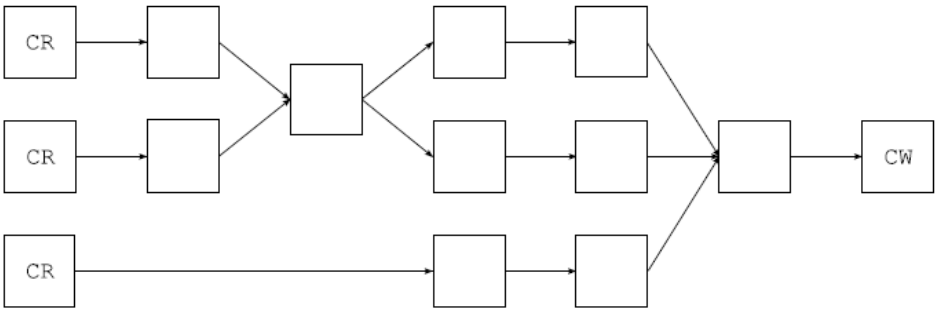


Figura 5.1: Exemplu de arhitectură bazată pe ASM

În exemplul de mai jos se poate vedea care este structura unei prelucrări ce foloseşte biblioteca ASM. ClassReader şi ClassWriter sunt obiecte care interfaţează efectiv cu codul binar Java de care utilizatorul este izolat. ClassReader-ul apelează metode din class adapter oridecâte ori citeşte o informaţie din fişierul binar. Astfel că există metode de tipul visitField(), visitMethod(), visitInnerClass(), visitEnd(). Prelucrările sunt astfel foarte simplu de efectuat asupra codului java binar deoarece suntem parţial scutiţi de structura sa. Tot ce trebuie făcut este să se creeze obiecte personalizate de tipul ClassAdapter care să efectueze acţiunile corespunzătoare atunci când apare un eveniment de acel tip.

Listing 5.1: Exemplu de prelucrare folosind ASM

```
ClassWriter writer = new ClassWriter(0);
ClassAdapter ca = new ClassAdapter(writer);
ClassReader reader = new ClassReader(new FileInputStream("in.class"));
reader.accept(adapter, 0);
FileOutputStream fileWriter = new FileOutputStream("out.class");

fileWriter.write(writer.toByteArray());
fileWriter.close();
```

Pentru o prelucrare la un nivel mai scăzut (atribuiri, comparaţii, operaţii aritmetice, tratare de excepţii) trebuie construit un MethodAdapter personalizat în interiorul ClassAdapter-ului. Acesta va fi la rândul său apelat tot de ClassReader de câte ori apar evenimente legate de citirea instrucţiunilor din interiorul unei metode. Pentru a putea defini structura MethodAdapter folosită trebuie explicate mai întâi câteva noţiuni de bază legate de codul binar Java şi de maşina virtuală. Aceste noţiuni vor fi descrise în continuare, dar pentru început să vedem cum sunt create obiectele de tip filtru în interiorul modulului şi care este rolul lor.

Ideea de bază a implementării modulului curent este să existe un obiect fabrică de tip singleton (AdapterFactory) care în funcţie de tipul cererii făcută modulului de instrumentare să creeze filtrele necesare. Spre examplu dacă se doreşte monitorizare apelurilor de funcţii, dar şi evaluarea instrucţiunilor condiţionate se vor crea două filtre. Unul va introduce predicate de control înainte de fiecare apel de funcţie, iar celălalt va introduce predicate care vor verifica fiecare evaluare de condiţie. Crearea unui filtru este prezentată în diagrama de secvenţă din figura 5.2

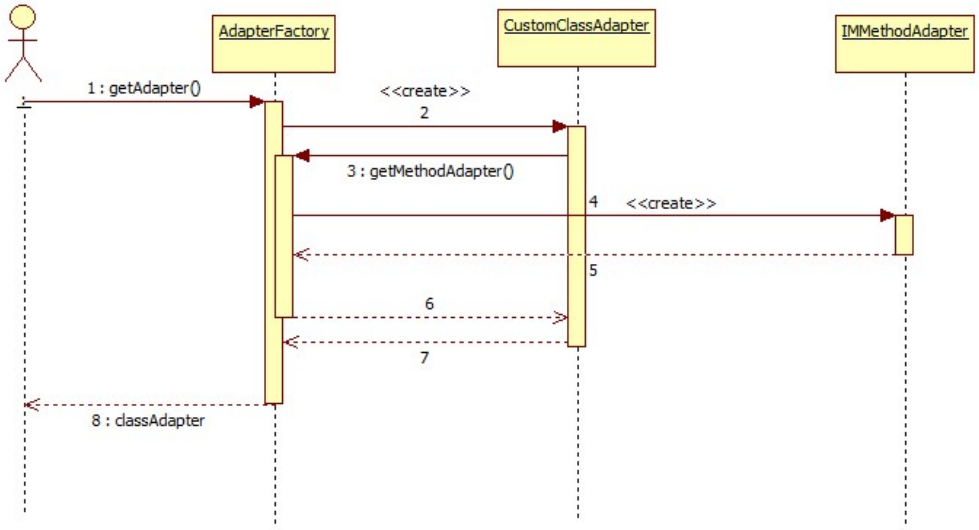


Figura 5.2: Crearea unui filtru folosind AdapterFactory

Conform [LY99], în maşina virtuală java, codul rulează în fire de execuţie. Fiecare astfel de fir are o aşa numită stivă de execuţii care este formată din cadre. Fiecare cadru reprezintă un apel de metodă. Un cadru este format din două părţi: una pentru variabilele locale şi încă una care poartă numele de stivă de operanzi. Porţiunea variabilelor locale poate fi accesată indexat folosind identificatorul variabilei locale. Cât despre stiva de operanzi, aceasta conţine valori ce vor fi utilizate de instrucţiunile JVM. Dimensiunea unui cadru depinde de codul sursă al metodei apelate.

În cadrul JVM sunt definite 256 de instrucţiuni care se împart în două mari categori: cele care fac transferul între stiva de operanzi şi memoria şi cele care iau operanzi de pe stivă, efectuează anumite operaţii şi mai apoi pun rezultatul pe stivă.

MethodVisitor este o interfaţă folosită de ASM pentru a parcurge instrucţiunile dintr-o metodă oarecare. La apelarea metodei visitMethod() din ClassAdapter un obiect de tipul MethodVisitor este creat. Dacă creăm un obiect MethodAdapter şi îl legăm de MethodVisitor-ul abia creat oridecâte ori se va petrece un eveniment legat de metoda respectivă obiectul creat de noi va fi notificat. Clasa MethodAdapter implementează MethodVisitor, având o mulţime de metode ce corespund diferitelor tipuri de instrucţiuni, dar şi altor evenimente cum ar fi calcularea dimensiunilor stivei sau parcurgerea unei etichete. Pentru a face anumite prelucrări la apariţia unui tip de eveniment trebuie suprascisă metoda ce tratează acel eveniment şi adăugat comportamentul dorit.

În continuare se va descrie cum anume se face injectarea predicatelor de control în codul binar al unei clase folosind un MethodAdapter propriu ce are ca scop monitorizarea predicatelor din cadrul instrucţiunilor de control.

Tehnica aleasă este una destul de intuitivă şi anume:

Listing 5.2: Monitorizarea predicatelor din cadrul instrucţiunilor de control

```
public void visitJumpInsn(int opcode, Label label) {
  if (opcode != Opcodes.GOTO) {
    if (oneOperandJump(opcode))
      duplicateTopElements(1);
    else
      duplicateTopElements(2);
    insertPredicate(opcode, label);
  }
  super.visitJumpInsn(opcode, label);
}
```

De fiecare dată când se întâlneşte o instrucţiune de salt condiţionat se introduce un predicat de control. Pentru a nu modifica fluxul programului, înainte de inserţia predicatului se recrează contextul evaluării. Adică dacă avem de-a face cu o instrucţiune de salt cu doi operanzi ultimii doi operanzi de pe stivă se duplică, iar dacă e o instrucţiune cu un singur operand se duplică doar ultimul element al stivei.

Listing 5.3: Inserţia efectivă a predicatelor

```
private void insertPredicate(int opcode, Label label) {

  Label l = new Label();
  Label l2 = new Label();

  super.visitJumpInsn(opcode, l);
  mv.visitMethodInsn(Opcodes.INVOKESTATIC, "ro/upt/ac/cstaicu/execution/logging/"
       + loggerClass, "getInstance", "()Lro/upt/ac/cstaicu/execution/logging/" +
      loggerClass + ";");
  mv.visitLdcInsn(predicateIndex);
  mv.visitLdcInsn(opcode);
  mv.visitLdcInsn(currentLineNumber);
  mv.visitLdcInsn(false);
  super.visitJumpInsn(Opcodes.GOTO, l2);

  super.visitLabel(l);
  mv.visitMethodInsn(Opcodes.INVOKESTATIC, "ro/upt/ac/cstaicu/execution/logging/"
       + loggerClass, "getInstance", "()Lro/upt/ac/cstaicu/execution/logging/" +
      loggerClass + ";");
  mv.visitLdcInsn(predicateIndex);
  mv.visitLdcInsn(opcode);
  mv.visitLdcInsn(currentLineNumber);
  mv.visitLdcInsn(true);

  super.visitLabel(l2);
  mv.visitMethodInsn(Opcodes.INVOKEVIRTUAL, "ro/upt/ac/cstaicu/execution/logging/
      " + loggerClass, "logPredicate", "(IIIZ)V");
  Entity predEnt =
    new Entity(new PredicateContext(predicateIndex, className, methodName,
        currentLineNumber, className + "_" + predicateIndexInsideClass), Entity.
        PREDICATE);
  predicates.add(predEnt);
  predicateIndex++;
}
```

După cum se poate observa, la fiecare identificare a unei astfel de instrucţiuni se creează o entitate cu un id unic care e hard-codat şi în predicatul de control pentru a le puteae asocia în viitor.

5.2.2 Modulul de execuţie

Execuţia testelor se face folosind interfeţele oferite de JUnit şi TestNG. Există câte o clasă de interfaţare cu fiecare dintre ele. Aceasta implementează interfaţa Executor. Ambele cadre de testare oferă o interfaţă asemănătoare bazată pe conceptul de Listener. Spre exemplu pentru JUnit rularea testelor se face astfel:

Listing 5.4: Rularea testelor folosind JUnit

```
JUnitCore junit = new JUnitCore();
JunitTestResultListener listener = new JunitTestResultListener();
junit.addListener(listener);
Result res = junit.run(testClass);
results.addAll(listener.getResults());
junit.removeListener(listener);
```

Pentru TestNG clasa care ascultă rezultatele testelor trebuie să implementeze ITestListener, iar pentru JUnit RunListener. Ambele au metode de tipul testFailed(), testStarted() sau testFinished(). În interiorul claselor de acest tip se creează obiecte de tip TestResult care vor reprezenta o parte din rezultatul analizei SherlockJ. Rezultatele testelor sunt pasate mai apoi modulului de analiză alături de spectrul program.

Comunicare cu modulul de instrumentare este realizat folosind conceptul de încărcare de clase oferit de Java. Acest mecanism este definit de specificaţiile JVM [LY99] astfel:

> Încărcarea se referă la procesul de găsire a reprezentării binare a unei clase sau interfeţe cu un nume dat. Aceasta se poate crea dinamic, dar de cele mai multe ori se citeşte dintr-un fişier .class şi se creează un obiect Class. Acest proces este implementat de clasa ClassLoader şi de subclasele acesteia. Diferite subclase pot implementa diferite politici de încărcare a claselor. În particular, o astfel de clasă poate salva reprezentările unei clase, preîncărca altele în funcţie de utilizarea aşteptată sau chiar încărcarea în grupuri de clase.
>
> Există două tipuri de ClassLoadere: cele predefinite de JVM şi cele definite de utilizator. Următorii paşi sunt urmaţi de JVM atunci când se încarcă o anumită clasă folosind un ClassLoader(CL) definit de utilizator: JVM apelează metoda loadClass din CL care va returna obiectul Class corespunzător. Mai apoi JVM înregistrează CL-ul curent ca încărcătorul iniţial pentru clasa respectivă. Clasele utilizate mai apoi de această clasă sunt încărcate folosind acelaşi CL dacă nu se specifică altfel. CL-ul poate pasa cererea de încărcare a clasei altui CL.

Acest mecanism a fost folosit pentru ca oridecâte ori se doreşte încărcarea unei clase din proiectul analizat ea să fie mai întâi instrumentată. Aceasta este o parte deosebit de importantă a aplicaţiei deoarece face posibilă rularea testelor în contextul lor din eclipse. ClassLoader-ul definit de SherlockJ este unul pe trei nivele ca în figura 5.3: primul nivel încearcă să localizeze clasa în interiorul claselor din proiect. Dacă aceasta a fost localizată aici, se verifică absenţa numelui clasei din lista de clase ignorate de procesul instrumentare. În funcţie de aceasta, clasa se instrumentutează sau nu înainte de încarcarea în JVM. De precizat ar fi că localizarea se înccearcă în interiorul dosarului de clase binare ale proiectului. Dacă ea nu se află aici se încearcă localizarea ei în rândul dependenţelor externe ale proiectului. În caz că aceasta se află aici ea va fi încărcată ad literam. În caz că nu s-a putut face încărcarea clasei nici de aici se va încerca încărcare clasei folosind CL-ul implicit al aplicaţiei. Acest lucru este util mai ales atunci când SherlockJ este rulat din linia de comandă şi anumite dependenţe externe sunt specificate direct la rulare ca argumente ale maşinii virtuale

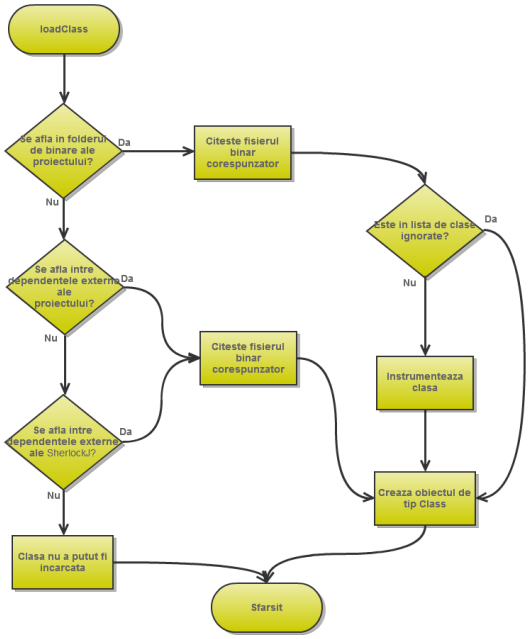


Figura 5.3: Procesul de încărcare a unei clase

în care rulează. Bazându-ne pe faptul că o dată încărcată o clasă cu un CL toate clasele folosite de acea clasă vor fi încărcate cu acelaşi CL, încărcarea cadrului de testare se va face manual folosind un CL personalizat. Acesta va fi utilizat mai departe în procesul de rulare a testelor. Aici am implementat şi un mecanism de memorare a claselor instrumentate, dar care va fi descris în capitolul următor.

Este foarte interesant de văzut cum anume se face legătura dintre rezultatul testelor şi entităţile evaluate în cadrul rulării unui test. Această operaţiune se efectuează folosind un obiect ce implementează interfaţa GlobalLogger. Acesta este apelat atât de predicatele de control cât şi de clasa ce ascultă rularea testelor. În cadrul SherlockJ am implementat două clase de tipul GlobalLogger: unul pentru salvarea în fişier a rezultatelor în format XML, iar unul pentru legarea rezultatelor testelor la entităţile create de modulul de instrumentare. În acest punct putem clarifica ce înseamnă de fapt instrumentarea efectuată de SherlockJ: extinderea unei clase cu capabilitatea de a transmite anumite informaţii unui GlobalLogger în timpul ruării. În lucrarea de faţă, valorile transmise sunt rezultatul evaluării predicatelor din cadrul condiţiilor sau apelurile de metode. Acest mecanism este extrem de uşor de folosit, putând introduce transmiterea oricăror parametrii observaţi în timpul rulării. Pe viitor se doreşte optimizarea acestei tehnici prin transimterea valorilor în grupuri mai mari, nu câte unul cum se face acum.

În figura 5.4 se poate vedea cum un GlobalLogger este apelat atât din interiorul listener-ului cât şi de către obiectul instrumentat. Din interiorul clasei instrumentate se apelează două tipuri de metode ale clasei jurnal (logger): logPredicate care transmite valoarea unui predicat în timpul rulării şi logFunctionCall care notifică apelarea unei funcţii. GlobalLogger-ul este cel care atunci când primeşte o cerere de tip logPredicate caută entitatea respectivă în lista primită de la modulul de instrumentare şi îi asociază acesteia un obiect de tip TestResult corespunzător testului curent.

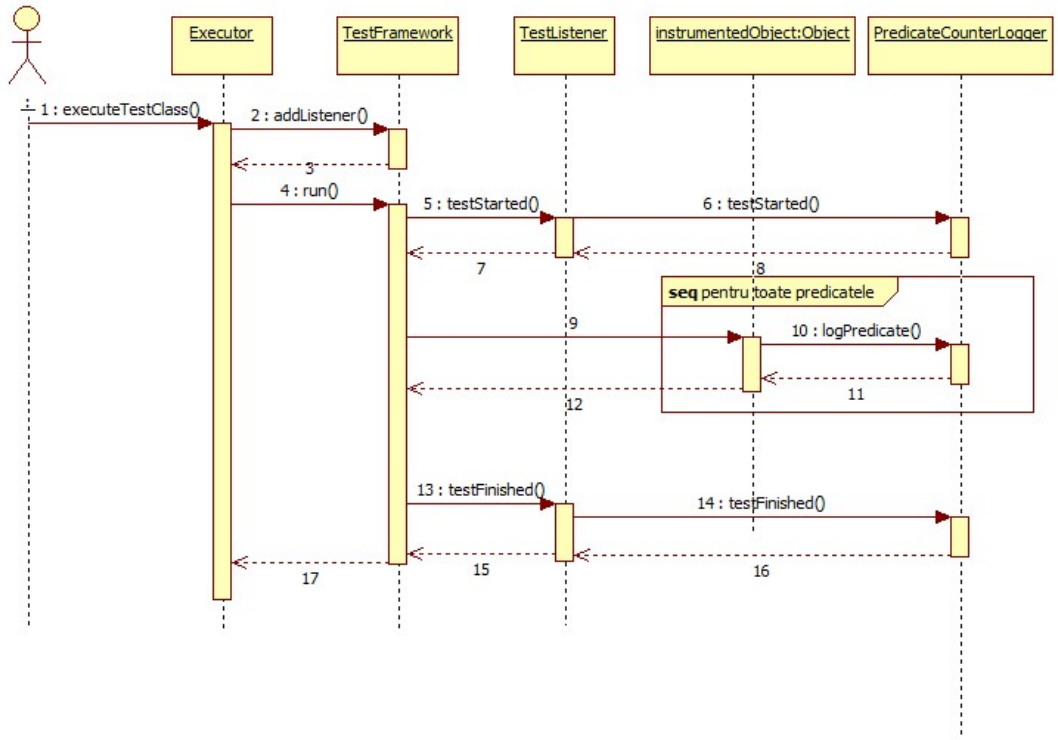


Figura 5.4: Cum se utilizeaă PredicateGlobalLogger

5.2.3 Modulul de analiză

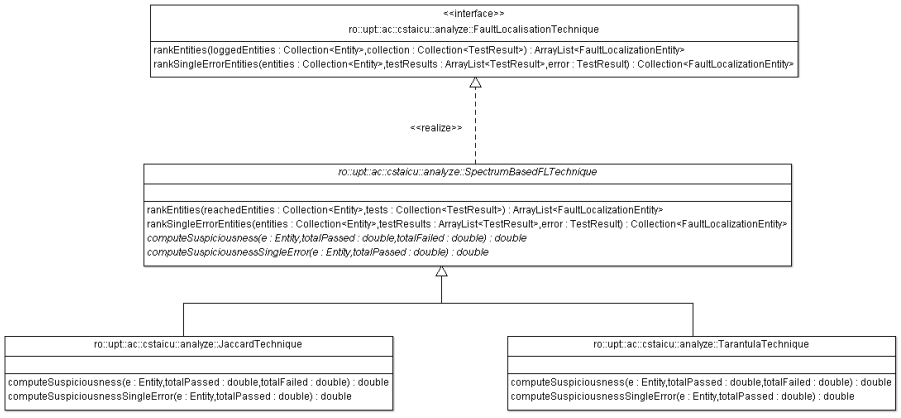


Figura 5.5: Ierarhia de tehnici de localizare

Implementarea modulului de analiză este una destul de simplă. Aici se asociază entităţilor create de modulul de instrumentare, coeficienţi de suspiciune, în funcţie de informaţiile asociate respectivei entităţi în timpul execuţiei. Tehnicile de calcul ale acestor coeficienţi au fost descrise în capitolul 2. Ierarhia de clase pentru tehnicile de analiză existente este cea din figura 5.5. Am folosit tiparul "metoda şablon" (template method) pentru a adăuga cele două clase concrete. Astfel, în clasa Spectrum SpectrumBasedFLTechnique am calculat coeficienţi necesari: numărul total de teste eşuate, numărul total de teste corecte. Folosind aceşti parametri, pentru

fiecare entitate am apelat metoda abstractă computeSuspiciousness(). Fiecare din clasele concrete implementează o metodă diferită de calculare a coeficienţilor. Mai jos puteţi vedea cum este implementată această metodă în cadrul tehnicii Tarantula. Tot în cadrul SpectrumBasedFLTechnique se face o sortare a entităţilor în funcţie de coeficienţii de suspiciune şi se creează câte un obiect FaultLocalizationEntity pentru fiecare entitate.

Listing 5.5: "Calcularea suspiciunii în cadrul tehnicii Tarantula"

```
public double computeSuspiciousness(Entity e, double totalPassed, double
    totalFailed) {
  return      (e.getNoFailed() / totalFailed) /
     (e.getNoPassed() / totalPassed + e.getNoFailed() / totalFailed);
}
```

A doua metodă din interfaţa FaultLocalisationTechnique şi anume rankSingleErrorEntities presupune că numărul total de teste picate este unu şi anume cel primit ca şi parametru. În felul acesta putem identifica entităţile care cel mai probabil au dus la eşuarea respectivului test, analizând testul picat împreună cu toate cele corecte. În continuare va fi prezentat un exemplu care să ilustreze utilitatea acestei abordării şi să arate modalitatea de calcul a coeficienţilor de suspiciune folosind cele două tehnici implementate. Vom începe mai întâi cu tehnica Tarantula. Vom considera o metodă care primind 3 valori reprezentând laturile unui triunghi testează dacă acestea formează un triunghi echilateral, dreptunghic, unul oarecare sau nu pot forma un triunghi. Codul original este prezentat mai jos, iar în dreptul fiecărui predicat se poate vedea identificatorul său.

Listing 5.6: "Exemplu: metodă ce returnează tipul unui triunghi"

```
public static int tipTriunghi(int a, int b, int c) {
    if (a <= 0 ||                                    //P0
      b <= 0 ||                                      //P1
      c <= 0)                                        //P2
      return Triunghi.IMPOSIBIL;
    if (a != b) {                                    //P3
      if (a + b <= c)                                //P4
        return Triunghi.IMPOSIBIL;
      if (a + c <= b)                                //P5
        return Triunghi.IMPOSIBIL;
      if (b + c <= a)                                //P6
        return Triunghi.IMPOSIBIL;
    }

    if ((a*a) + (b*b) - (c*c) == 0 ||                //P7
      ((a*a) + (c*c) - (b*b) == 0) ||                //P8
      ((b*b) + (c*c) + (a*a) == 0))                  //P9
      return Triunghi.DREPTUNGHIC;

    if (a == b &&                                    //P10
      b != c)                                        //P11
      return Triunghi.ECHILATERAL;

    return Triunghi.OARECARE;
```

Aşa cum se poate observa predicatele P9 şi P11 conţin o eroare care trebuie identificată cât mai bine de SherlockJ.

În tabelul 5.1 pe primele coloane sunt ilustrate spectrele program pentru fiecare test (T1 - T7). Pe ultimul rând cu 1 este marcat un test reuşit, iar cu 0 unul eşuat. Ultimele 3 coloane prezintă pentru fiecare predicat coeficientul de suspiciune asociat.

Predicatul	T1	T2	T3	T4	T5	T6	T7	TNT(E)	TNT(E, T1)	TNT(E, T4)
P0	F	F	F	F	F	F	F	0.5	0.5	0.5
P1	F	F	F	F	F	F	F	0.5	0.5	0.5
P2	F	F	T	F	F	F	F	0.5	0.5	0.5
P3	T	T	-	T	F	T	T	0.55	0.55	0.55
P4	F	F	-	F	-	F	F	0.38	0.55	0
P5	F	F	-	F	-	F	F	0.38	0.55	0
P6	F	F	-	T	-	F	F	0.38	0.55	0
P7	F	F	-	-	F	T	F	0.62	0.625	0.625
P8	F	T	-	-	F	-	F	0.71	0.71	0.71
P9	F	-	-	-	F	-	F	0.83	0.83	0.83
P10	F	-	-	-	T	-	F	0.83	0.83	0.83
P11	-	-	-	-	F	-	T	1	-	1
-	0	1	1	1	0	1	1	-	-	-

Tabelul 5.1: Exemplu de analiză SherlockJ folosind metoda Tarantula

Pe coloana TNT(E) este coeficientul agregat format din analiza tuturor test, iar pe celelalte două este prezentată câte o analiză pentru fiecăre test eşuat în parte. După cum se vede, coeficienţii variază în funcţie de spectrul program al respectivului test. Mai precis, predicatul P11 apare cu coeficient de suspiciune 0 atunci când analizăm testul 1 şi totuşi el este suspectul principal atunci când analizăm testul 5 al cărui cauză de eroare şi este. În continuare vom prezenta aceiaşi analiză făcută cu tehnica Jaccard.

Predicatul	T1	T2	T3	T4	T5	T6	T7	JCD(E)	JCD(E, T1)	JCD(E, T4)
P0	F	F	F	F	F	F	F	0.28	0.17	0.16
P1	F	F	F	F	F	F	F	0.28	0.17	0.16
P2	F	F	T	F	F	F	F	0.28	0.17	0.16
P3	T	T	-	T	F	T	T	0.33	0.2	0.2
P4	F	F	-	F	-	F	F	0.16	0.2	0
P5	F	F	-	F	-	F	F	0.16	0.2	0
P6	F	F	-	T	-	F	F	0.16	0.2	0
P7	F	F	-	-	F	T	F	0.4	0.25	0.25
P8	F	T	-	-	F	-	F	0.5	0.33	0.33
P9	F	-	-	-	F	-	F	0.66	0.5	0.5
P10	F	-	-	-	T	-	F	0.66	0.5	0.5
P11	-	-	-	-	F	-	T	0.5	-	1
-	0	1	1	1	0	1	1	-	-	-

Tabelul 5.2: Exemplu de analiză SherlockJ folosind metoda Jaccard

După cum se poate observa, valorile coeficienţilor de suspiciune asociaţi de această a doua metodă sunt mult mai naturale. Un avantaj al acestei tehnici constatat în urma testelor este faptul că scoate mult mai bine în evidenţă entităţile suspicioase. La Tarantula toate valorile sunt apropiate. În plus, pentru exemplul considerat toate valorile sunt peste 0.4, fapt ce l-ar putea face pe un observator neexperimentat să fie sceptic în ceea ce priveşte validitatea tehnicii.

5.2.4 Plugin-ul de eclipse

În cadrul eclipse, plugin-urile se definesc folosind un fişier xml special (plugin.xml). Pentru o editare mai uşoară există un editor special pentru accest tip de fişiere care ajută la configurarea mai uşoară a poprietăţilor proiectului. Informaţiile generale ale proiectului pot fi văzute în figura 5.6.

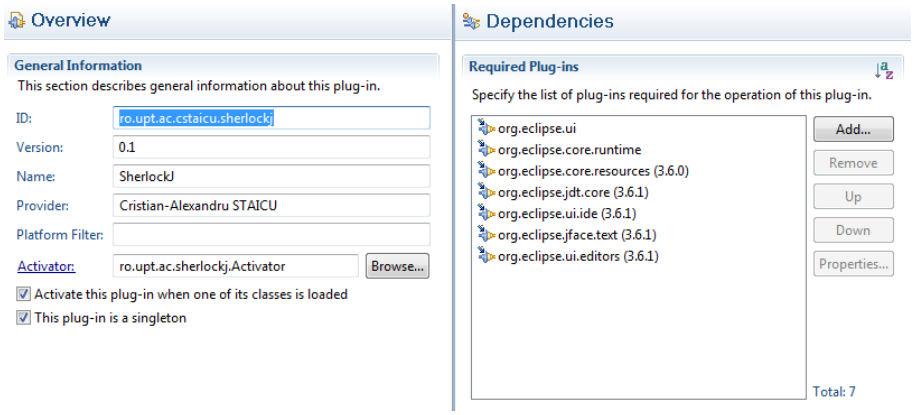


Figura 5.6: Proprietăţile generale SherlockJ

Tot aici se definesc punctele de extensie pentru SherlockJ. Prima dintre acestea este intrarea în meniul contextual: atunci când se dă click dreapta pe un proiect Java. Vizibilitatea acestei intrări e condiţionată de selectarea unui obiect de tip org.eclipse.core.resources.IProject în eclipse. La apăsarea acestei acţiuni se declanşează ro.upt.ac.sherlockj.ContextMenuListener, definită în interiorul proiectului. Aceasta reprezintă punctul de intrare în aplicaţia SherlockJ. O a doua extindere oferită de aplicaţie este o vedere ce conţine rezultatele analizei. Declaraţia acesteia poate fi văzută în figura 5.7.

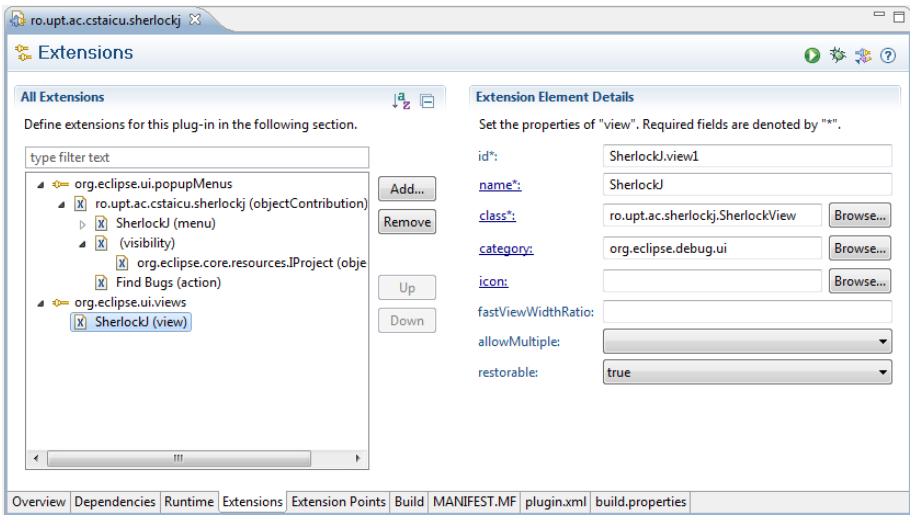


Figura 5.7: Declaraţia vederii eclipse

În cadrul plugin-ului, nucleul SherlockJ este definit ca bibliotecă externă. La declanşarea acţiunii de analiză se identifică şi filtrează testele unitare după care se face o cerere nucleului. Rezultatele primite se afişează în vederea definităm, într-o manieră specifică platformei eclipse prezentată în fragmentul de cod de mai jos.

Localizarea vederii se face folosind serviciile din clasa Platform şi id-ul vederii ca în porţiunea de cod de mai jos:

Listing 5.7: "Afisarea rezultatelor analizei în SherlockView"

```
IViewPart view = PlatformUI.getWorkbench().getActiveWorkbenchWindow().
    getActivePage().findView("SherlockJ.view1");
  if (view instanceof SherlockView) {
    SherlockView sherlockView = (SherlockView) view;
    sherlockView.setTests(result.testResults);
    sherlockView.drawNewRankingTable(
      result.rankedEntities,
      TestFilter.this.selectedProject);
  }
```

5.3 Probleme speciale şi rezolvarea lor

5.3.1 Încărcarea claselor din biblioteci de tip jar

În cadrul mecanismului personalizat de încărcare a claselor descris în secţiunea 5.2.2 am avut nevoie de un mecanism rapid de încărcare a unei clase dintr-o arhivă de tip jar. Astfel de arhive sunt uzual folosite pentru distribuirea claselor binare ale uneie biblioteci Java. Problema era faptul că deschiderea şi parcurgerea arhivelor de fiecare dată când se dorea încărcarea unei clase este mare consumatoare de timp. Pentru rezolvarea acestei probleme am construit un obiect care să gestioneze încărcarea claselor dintr-o arhivă dată. La iniţializare se citesc numele tuturor claselor din arhivă şi se salvează într-o asociere alături de dimensiunea lor. Astfel că de fiecare dată când trebuie verificată prezenţa unei clase în arhiva curentă, operaţie destul de frecventă, nu este necesară deschiderea arhivei. O altă îmbunătăţire a acestei metode a fost salvarea claselor deja încărcate pentru a fi folosite la o încărcare viitoare. Mecanismul de iniţializare descris este prezentat în fragmentul următor de cod:

Listing 5.8: "Clasa ImprovedJarReader"

```
private Hashtable<String, Integer> classSizes = new Hashtable<String, Integer>();
private Hashtable<String, byte[]> classesContent = new Hashtable<String, byte
    []>();

private void init() throws IOException {
  ZipFile zf;
  zf = new ZipFile(jarFileName);
  Enumeration<? extends ZipEntry> e = zf.entries();
  while (e.hasMoreElements()) {
    ZipEntry ze = (ZipEntry) e.nextElement();
    classSizes.put(ze.getName(), new Integer((int) ze.getSize()));
  }
  zf.close();
}
public byte[] getResource(String name) {
  if (!classSizes.keySet().contains(name))
    return null;
  if (classesContent.keySet().contains(name))
    return classesContent.get(name);
  return readClass(name);
}
```

5.3.2 Problema instrumentării multiple a claselor

Această problemă a apărut în cadrul modulului de instrumentare şi producea inconsistenţe la nivelul entităţilor. Acestea se datorau faptului că pentru o entitate oarecare din program se creau mai multe obiecte de tip Entity: câte unul la fiecare intrumentare. O altă problemă era şi penalizarea de timp inutilă adusă de repetarea acestui proces. Rezolvarea propusă implică salvarea claselor instrumentate în fişiere temporare ce vor fi şterse la terminarea execuţiei programului. ClassLoader-ul verifică înainte de a cere instrumentarea unei clase dacă nu există deja un astfel de fişier temporar. Dacă există se încarcă de acolo, iar dacă nu, se instrumentează şi mai apoi se creează un fişier temporar care va memora binarele clasei instrumentate. Mai jos este prezentată clasa care se ocupă cu gestiunea fişierelor temporare ce conţin clase instrumentate:

Listing 5.9: "Mecanismul de salvare a claselor deja instrumentate"

```
public class CachedClassFilesManager {

  private static HashMap<String, File> history = new HashMap<String, File>();

  public synchronized static CachedClassFileResult getClass(String name) throws
      IOException {
    if (history.containsKey(name)) {
      return new CachedClassFileResult(history.get(name), true);
    } else {
      File tempFile = File.createTempFile(name.replace(".", ""), "class");
      tempFile.deleteOnExit();
      history.put(name, tempFile);
      return new CachedClassFileResult(tempFile, false);
    }
  }

  public static void clearHistory() {
    history = new HashMap<String, File>();
  }
}
```

Este interesant de văzut faptul că se întoarce oricum un obiect şi pentru o clasă care nu a fost încă instrumentată. Acesta este însă marcat ca fiind incomplet şi va fi completat după instrumentare. Salvarea codului binar în fişierul temporar se face de către ClassLoader astfel:

Listing 5.10: "Scrierea clasei într-un ficşier temporar"

```
if (!result.isWrote) {
  byte classBytecode[] = instrumentationModule.tracePredicatesFromClass(
      projectClassPath, name);
  FileOutputStream cachedClassFile = new FileOutputStream(result.classFile);
  cachedClassFile.write(classBytecode, 0, classBytecode.length);
  cachedClassFile.close();
}
```

5.3.3 Problema dezalocării obiectelor Class nefolosite

În cadrul maşinii virtuale Java, obiectele Class sunt dezalocate atunci când nu mai există nici o instanţă de-a lor încărcată şi când *nu mai există ClassLoader-ul care le-a încărcat*. Acest fapt a dus la depăşirea dimensiunii permise de către JVM pentru numărul de clase. În unul din proiectele analizate se crea un proces daemon care avea o referinţă spre ClassLoader-ul care l-a încărcat şi astfel ClassLoader-ul

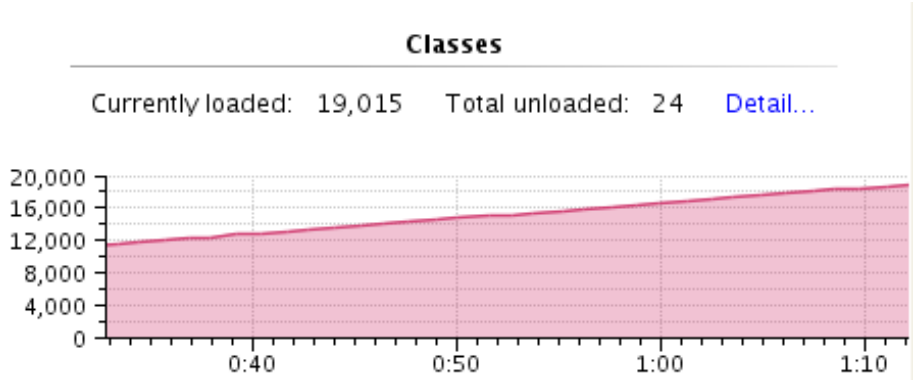


Figura 5.8: Ilustrarea problemei dezalocării claselor

respectiv nu putea fi colectat de GC, chiar dacă analiza SherlockJ s-a terminat. Odată cu el rămâneau încărcate toate clasele pe care le încărcase. În figura 5.8 este ilustrată această problemă folosind YourKit Java Profiler [YKJ11].

Listing 5.11: "Terminarea thread-urilor rămase după o analiză"

```
Thread[] threads =
  new Thread[Thread.currentThread().getThreadGroup().activeCount()];
Thread.currentThread().getThreadGroup().enumerate(threads);
Thread currentThread = Thread.currentThread();
for (Thread t : threads) {
  if (!t.equals(currentThread))
    t.stop();
}
```

Problema a fost rezolvată controlând numărul de fire de execuţie după fiecare analiză. În cazul în care se descoperă un fir de execuţie neterminat acesta este oprit brutal, dezalocând astfel obiectele Class nefolosite. În figura 5.9 se pot observa rezultatele după rezolvarea acestei probleme.

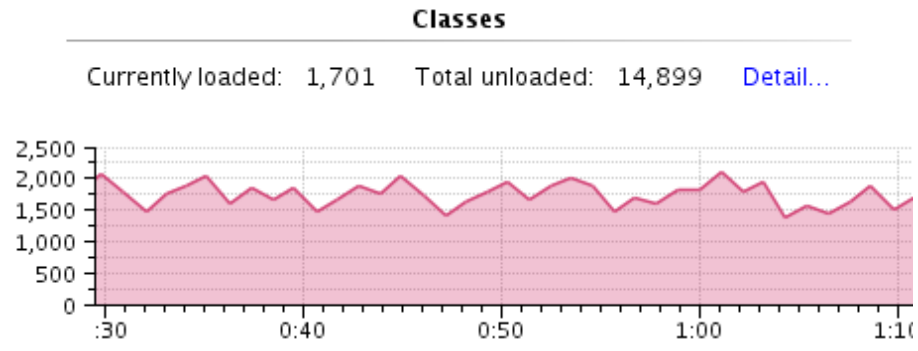


Figura 5.9: Variaţia numărului de clase după rezolvarea problemei

5.3.4 Identificarea testelor dintr-un proiect Java

Identificarea testelor din proiectul analizat a fost o prioritate încă de la începutul dezvoltării plugin-ului. Se dorea scutirea utilizatorului de fişiere de configurare adiţionale, dar în acelaşi timp oferirea posibilităţi de customizare a testelor ce fac obiectul analizei.

Atât JUnit cât şi TestNG, în ultimele versiuni, acceptă adnotările ca mecanism de marcare a testelor unitare. Pentru compatibilitate cu proiectele mai vechi SherlockJ detectează însă şi acele teste unitare care au fost scrise prin extinderea clasei TestCase (mecanism folosit de versiunile JUnit mai vechi). Pentru implementarea strategiei de detecţie am folosit modelul AST oferit de interfaţa de programare eclipse. Acesta reprezintă, în memorie, codul sursă Java ca un arbore.

Listing 5.12: "Detecţia automată a testelor unitare"

```
public static boolean isTest(IJavaElement element) {
  if (IJavaElement.COMPILATION_UNIT == element.getElementType())
    try {
      IType[] types = ((ICompilationUnit)element).getAllTypes();
      for (int i = 0; i < types.length; i++)
        if (isValidType(types[i]))
          return true;
    } catch (JavaModelException e) {
      return false;
    }
  else if (IJavaElement.TYPE == element.getElementType()) {
    if (isValidType((IType)element))
      return true;
  }
  return false;
}

private static boolean isValidType(IType iType) {
  ITypeHierarchy th = iType.newSupertypeHierarchy(null);
  for (IType ancestor : th.getAllClasses())
    if (ancestor.getFullyQualifiedName().equals("junit.framework.TestCase"))
      return true;
  String source = iType.getSource();
  ASTParser parser = ASTParser.newParser(AST.JLS3);
  parser.setProject(iType.getJavaProject());
  parser.setKind(ASTParser.K_COMPILATION_UNIT);
  parser.setUnitName(iType.getCompilationUnit().getPath().toString());
  parser.setResolveBindings(true);
  parser.setSource(source.toCharArray());
  CompilationUnit compUnit = (CompilationUnit)parser.createAST(null);
  TestVisitor visitor = new TestVisitor();
  compUnit.accept(visitor);
  return visitor.isTest();
}
```

Aşa cum se poate observa şi această strategie se bazează pe tiparul visitor. Clasa TestVisitor are ca principal scop detecţia adnotărilor din codul vizitat. Marele dezavantaj al acestei tehnici este timpul de detecţie relativ mare. Pe viitor trebuie implementat un mecanism de stocare al acestor informaţii pentru o reutilizare viitoare.

Capitolul 6

Utilizarea sistemului

6.1 Cerinţe minime

Pentru a rula SherlockJ aveţi nevoie de:

- Eclipse 3.5+
- un proiect cu o suită de teste TestNG sau JUnit

6.2 Instalarea

Pentru instalare se descarcă fişierul ro.upt.ac.cstaicu.sherlockj_0.1.0.jar de la adresa `http://code.google.com/p/junit-debugger/downloads/list` şi se copiază în directorul plugin al Eclipse. Pentru a verifica dacă plugin-ul a fost instalat verificaţi în secţiunea Help -> Install New Software -> allready installed -> Plug-ins că există SherlockJ.

6.3 Scenariu tipic de utilizare

6.3.1 Pluginul de eclipse

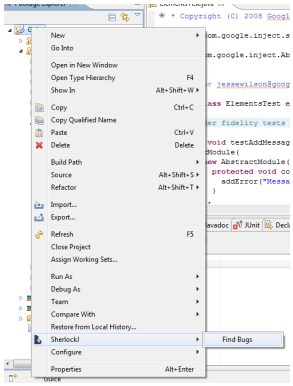


Figura 6.1: SherlockJ în meniu contextual

Pentru o utilizare corectă trebuie deschisă mai întâi vederea SherlockJ. Selectaţi Window -> Show view -> Other şi aici alegeţi din categoria Debug vederea SherlockJ. În acest moment plug-inul este gata de utilizare. Selectaţi un proiect şi daţi click dreapta. În meniul contextual selectaţi SherlockJ -> Find bugs ca în Figura 6.1.

În acest moment au fost detectate testele unitare din proiectul selectat. Pe ecran va apărea o fereastră (Figura 6.2) în care trebuie selectată strategia de localizare (Tarantula / Jaccard) şi cadrul de testare folosit (JUnit / TestNG). Tot aici se pot selecta testele care vor fi rulate în cadrul analizei. Spre exemplu dacă se lucrează la un singur modul din întreagul proiect, aici trebuie selectate doar testele corespunzătoare acelui modul. În felul acesta se reduce semnificativ timpul de analiză. După setarea parametrilor amintiţi se apasă "Find bugs". Acum SherlockJ rulează testele, instrumentând clasele necesare după care efectuează analiza selectată. Acaste operaţiuni pot dura destul de mult, în funcţie de numărul de teste selectate.

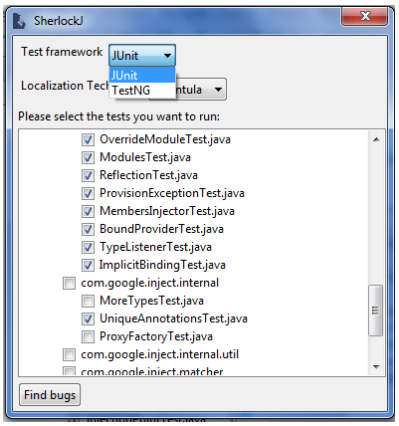
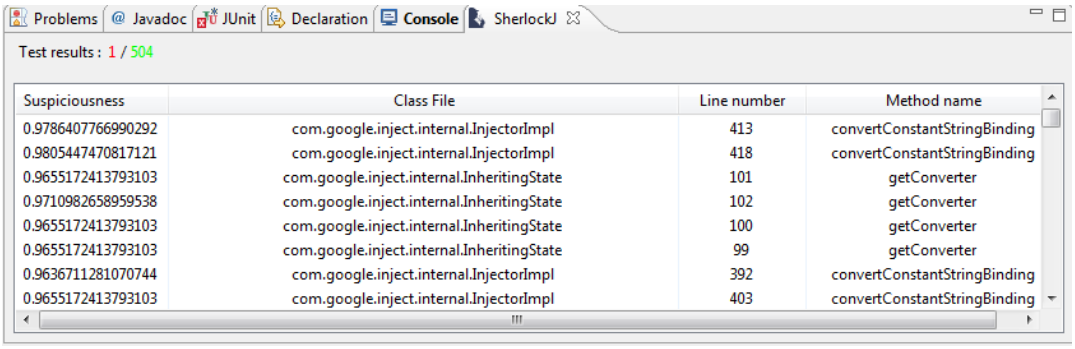


Figura 6.2: Filtrarea testelor şi alegerea strategiei

După terminarea analizei SherlockJ populează Tabelul de entităţi suspicioase, ordonând entităţile în ordine descrescătoare a probabilităţii ca ele să conţină erori (Figura 6.3). După cum se poate observa, tabelul conţine pentru fiecare entitate coeficientul de suspiciune, numele complet al clasei, linia şi numele metodei. Cu roşu sunt afişate numărul de teste eşuate, iar cu verde numărul celor reuşite.



Suspiciousness	Class File	Line number	Method name
0.9786407766990292	com.google.inject.internal.InjectorImpl	413	convertConstantStringBinding
0.980544747081721	com.google.inject.internal.InjectorImpl	418	convertConstantStringBinding
0.9655172413793103	com.google.inject.internal.InheritingState	101	getConverter
0.9710982658959538	com.google.inject.internal.InheritingState	102	getConverter
0.9655172413793103	com.google.inject.internal.InheritingState	100	getConverter
0.9655172413793103	com.google.inject.internal.InheritingState	99	getConverter
0.9636711281070744	com.google.inject.internal.InjectorImpl	392	convertConstantStringBinding
0.9655172413793103	com.google.inject.internal.InjectorImpl	403	convertConstantStringBinding

Figura 6.3: Tabelul de entităţi suspicioase

În acest punct utilizatorul poate începe analiza secvenţială a tabelului generat. Prin apăsarea unui rând din tabel se va deschide un editor în care va fi evidenţiată respectiva entitate. Dacă aceasta este cea care conţine erori analiza se încheie, dacă nu se trece la analizarea următoarei intrări în tabel. Rezultatele ce vor fi prezentate în continuare arată că în majoritatea cazurilor eroare se găseşte într-o entitate conţinută între primele 10% din tabel.

Dacă se doreşte inspectarea fiecărui test eşuat în parte se va da click pe numărul de teste eşuate. În acest moment SherlockJ recalculează coeficienţii de suspiciune ca

şi cum un singur test ar fi fost eronat şi anume testul curent. În figura 6.4 se pot observa trei butoane de navigare care pot fi folosite pentru a trece de la un test la altul sau pentru a reveni la vederea iniţială cu coeficienţii globali. Tot deasupra tabelului se găsesc informaţii despre testul curent şi motivul pentru care a eşuat testul.

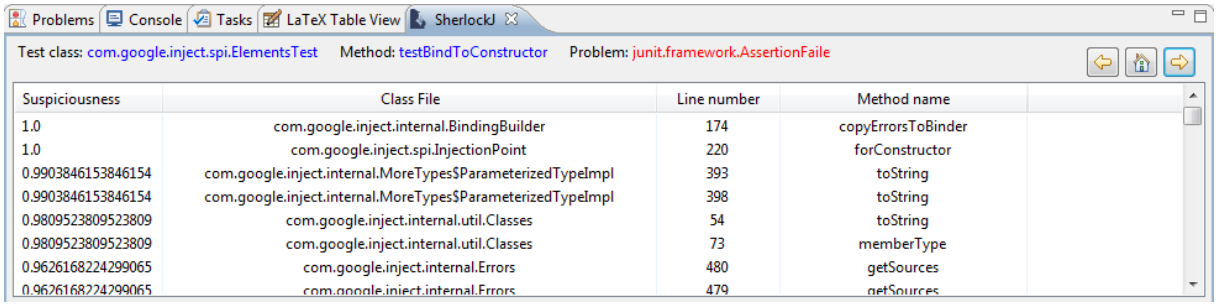


Figura 6.4: Inspectarea testelor eşuate

6.3.2 Folosirea nucleului SherlockJ din linia de comandă

Mai jos este prezentat un script care rulează o analiză folosind nucleul SherlockJ din linia de comandă. Rezultatul este un fişier csv ce conţine entităţile ordonate în funcţie de coeficientul de suspiciune. Pentru o descriere a argumentelor acceptate puteţi consulta secţiunea 3.3.1. De amintit ar fi faptul că trebuie specificată calea spre bibliotecile ASM şi JUnit / TestNG alături de calea spre nucleul SherlockJ.

Listing 6.1: Exemplu de utilizare SherlockJ din linia de comandă

```
java -cp SherlockJ_alpha_0.01.jar:\
  /home/cristi/Licenta/workspace/EvaluateSherlockJ/asm-3.3.jar:\
  /home/cristi/Licenta/workspace/junit-debugger/lib/junit-4.8.2.jar
  ro.upt.ac.cstaicu.ui.Main \
  -cp /home/cristi/Licenta/proiecte-de-testat/Guice/src:\
  /home/cristi/Licenta/proiecte-de-testat/Guice/test:\
  /opt/java/jre/lib/resources.jar:/opt/java/jre/lib/rt.jar:\
  /opt/java/jre/lib/jsse.jar:\ /opt/java/jre/lib/jce.jar:\
  /opt/java/jre/lib/charsets.jar:\
  /opt/java/jre/lib/ext/dnsns.jar:\
  /opt/java/jre/lib/ext/sunpkcs11.jar:\
  /home/cristi/Licenta/proiecte-de-testat/Guice/lib/javax.inject.jar \
  -pp /home/cristi/Licenta/proiecte-de-testat/Guice \
  -bp /home/cristi/Licenta/proiecte-de-testat/Guice/bin \
  -tf JUnit -flt Tarantula -of out.csv
  -tcf /home/cristi/Licenta/workspace/EvaluateSherlockJ/tests
```

Capitolul 7

Rezultate experimentale

7.1 Injectarea de erori

Folosind biblioteca ASM am conceput un program de injecţie de erori direct în codul binar Java, urmând ca acestea să fie detectate ulterior folosind SherlockJ. Erorile injectate constau în inversarea fiecărui predicat din program generând pentru fiecare modificare câte o versiune a clasei numită mutant. Astfel că pentru un program ce conţine n predicate vom avea n mutanţi, generaţi prin modificarea instrucţilor de control după cum se poate vedea în Tabelul 7.1.

II	IM	II	IM	II	IM
IF_ACMPNE	IF_ACMPEQ	IFEQ	IFNE	IFNE	IFEQ
IF_ICMPEQ	IF_ICMPNE	IF_ICMPGE	IF_ICMPLT	IFNULL	IFNONNULL
IF_ICMPNE	IF_ICMPEQ	IF_ICMPLE	IF_ICMPGT	IFNONNULL	IFNULL
IF_ICMPLT	IF_ICMPGE	IF_ICMPGT	IF_ICMPLE	IF_ACMPEQ	IF_ACMPNE

Tabelul 7.1: Modul de alterare al predicatelor

Astfel, fiecare instrucţiune din coloana II (instrucţiunea iniţiala) se transformă în instrucţiunea din coloana IM (instrucţiunea modificată).

7.2 Modul de evaluarea al sistemului

Pentru evaluarea automată a lui SherlockJ am creat un alt utilitar care preia mutanţii generaţi anterior şi îi copiază unul câte unul peste clasa originală. Fiecare mutant este identificat prin numărul predicatului pe care l-a alterat din clasa iniţială. După fiecare copiere se rulează o analiză cu SherlockJ şi se caută poziţia predicatului modificat în Tabelul de predicate suspicioase. Ideal ar fi ca acesta să fie primul, dar în [AZvG07] se precizează că o plasare intre primele 20% este satisfăcătoare.

De precizat ar fi faptul că numărul de predicate poate varia de la o rulare la alta deoarece anumite căi din program pot fi impracticabile prin alterarea unui predicat. Dacă prin înlocuirea clasei originale cu o clasă mutant nu avem nici un test eşuat înseamnă că acel predicat nu este acoperit de suita de teste sau că eroarea a fost absorbită undeva în interiorul modulului. Într-un astfel de caz spunem că respectivul mutant nu este detectat de suita de teste.

7.3 Rezultate experimentale

Pentru evaluarea produsului am ales ca proiect de test Google Guice [GGC11], un cadru de injectare de dependenţe, şi am injectat condiţii greşite în clasa InjectionPoint, rezultând 48 de mutanţi. Aceştia au înlocuit rând pe rând clasa originală, rulând pentru fiecare în parte o analiză SherlockJ folosind testele din pachetul com.google.inject.spi. Rezultatele obţinute sunt prezentate în Tabelele 7.2 şi 7.3. Pe prima coloană este numărul liniei la care s-a introdus eroarea, pe a doua numărul de teste care au fost eronate din această cauză. Iar pe ultima coloană se află poziţia predicatului în Tabelul de entităţi suspicioase.

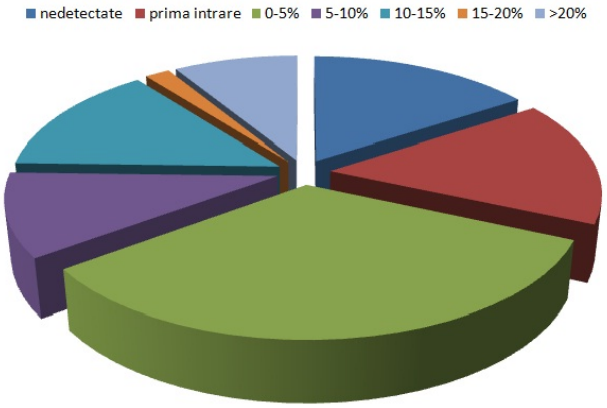


Figura 7.1: Acurateţea diagnozei cu coeficientul Tarantula

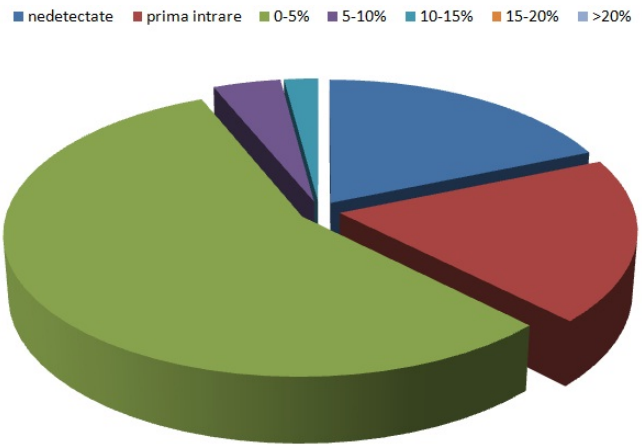


Figura 7.2: Acurateţea diagnozei cu coeficientul Jaccard

În figura 7.1 este prezentată acurateţea diagnozei pentru cazul când s-a folosit coeficientul Tarantula, iar în figura 7.2 pentru Jaccard. După cum se poate observa, coeficientul Jaccard a produs rezultate sensibil mai bune. Aşa cum se poate observa, mai bine de jumătate din predicatele eronate injectate s-au aflat în primele 5% în tabelul de entităţi suspicioase. Cadrul de testare folosit a fost JUnit.

Penalizarea de timp adusă de SherlockJ la rularea testelor a fost de 68,48%, iar cea de spaţiu (cu cât a crescut dimensiunea claselor) de 19.01%. Testele au fost rulate pe un laptop Dell 6400 cu procesor Intel Dual Core 2GHz şi 1GB memorie, pe un sistem de operare Arch Linux.

Linia	Nr. teste picate	Poziţia în tabel
112	5	5/1101
184	7	16/1101
185	5	0/1101
186	5	3/1101
220	0	-
247	13	13/1082
249	5	40/1101
257	3	0/1101
261	3	1/1101
243	3	26/1101
271	13	18/1082
280	4	67/1101
281	6	8/1098
322	1	3/1101
357	5	98/1101
384	14	35/1065
390	0	-
392	0	-
467	15	154/1079
638	33	332/1088
638	1	4/1101
640	1	5/1101
650	9	149/1101
652	60	85/1088
654	0	-
654	0	-
649	21	294/1098
663	7	149/1101
665	56	70/1070
669	8	10/1083
670	0	-
672	0	-
682	1	19/1101
685	10	8/1094
701	53	131/1088
703	0	-
662	18	175/1065
637	15	154/1079
715	15	173/1097
725	2	1/1101
720	15	32/1097
736	0	-
738	1	2/1101
742	1	3/1101
753	15	171/1079
767	1	8/1101
767	1	10/1101
770	1	11/1101

Tabelul 7.2: Coeficientul Tarantula

Linia	Nr. teste picate	Poziţia în tabel
112	5	0/1101
184	7	1/1101
185	5	0/1101
186	5	0/1101
220	0	-
247	13	0/1082
249	5	30/1101
257	3	0/1101
261	3	1/1101
243	3	23/1101
271	13	2/1082
280	4	24/1101
281	6	0/1098
322	1	3/1101
357	5	99/1101
384	14	6/1065
390	0	-
392	0	-
467	15	29/1079
638	33	140/1088
638	1	4/1101
640	1	21/1101
650	9	71/1101
652	60	1/1088
654	0	-
654	0	-
649	21	54/1098
663	7	46/1101
665	56	1/1070
669	8	1/1083
670	0	-
672	0	-
682	1	23/1101
685	10	0/1094
701	53	0/1088
703	0	-
662	18	45/1065
637	15	30/1079
715	15	47/1097
725	2	1/1101
720	15	0/1097
736	0	-
738	1	2/1101
742	1	3/1101
753	15	25/1079
767	1	4/1101
767	1	5/1101
770	1	6/1101

Tabelul 7.3: Coeficientul Jaccard

Capitolul 8

Concluzii

8.1 Ce s-a realizat

În lucrarea de faţă a fost prezentat SherlockJ, o platformă pentru depanare statistică a aplicaţiilor scrise în limbajul Java folosind mediul de dezvoltare Eclipse. Această aplicaţie poate fi folosită şi independent de Eclipse, din linia de comandă. Cele trei module constituente sunt foarte bine decuplate şi pot fi uşor reutilizate pe viitor. Noi tehnici de localizare de erori, dar şi alte instrumente de instrumentare sau cadre de testare pot fi uşor integrate pe viitor, datorită design-ului flexibil.

Aplicaţia permite analizarea oricărui proiect Java ce dispune de o suită de teste JUnit sau TestNG. Înainte de rularea testelor se instrumentează codul binar al anumitor clase fără a altera în vreun fel proiectul original. Instrumentarea a fost făcută folosind biblioteca ASM, aceasta fiind una dintre cele mai eficiente alternative pentru alterarea unui fişier binar Java. Analiza dinamică foloseşte coeficienţii Jaccard şi Tarantula şi permite ierarhizarea globală a predicatelor, cât şi o ierarhizare individuală pentru fiecare test picat în parte.

Am injectat erori într-o aplicaţie reală (Google Guice) după care am observat rezultatele analizei SherlockJ pe acest proiect. Rezultatele promiţătoare mă îndreptăţesc să cred că prin rezolvarea problemelor de scalabilitate şi interpretare a rezultatelor ar putea deveni un sistem util oricărui dezvoltator.

8.2 Direcţii de dezvoltare

Probabil cea mai mare limitare a lui SherlockJ este faptul că detectează erori doar la nivelul condiţiilor din program. Posibilitatea schimbării nivelului de granularitate la care se face analiza este o capabilitate ce trebuie adăugată pe viitor, dar şi adăugarea de noi tehnici de localizare.

Aşa cum se specifică în [Liu06] cea mai mare problemă a depanării statistice o reprezintă interpretarea rezultatelor. Pentru o mai bună înţelegere a rezultatelor trebuie definite metode de vizualizare în locul tabelelor. Astfel dintr-o privire utilizatorul poate să vadă anomaliile sistemului. Posibile astfel de vizualizări ar fi: vizualizarea definită de către Tarantula [JHS02] sau vizualizările polimetrice [LD03] folosite în analiza structurală.

O altă funcţionalitate necesară aplicaţiei ar fi aceea de a oferi utilizatorului feedback referitor la suita de teste. Un astfel de feedback este criteriul TfD [BFLT06] care se bazează pe noţiunea de bloc dinamic de bază (entităţile ce nu pot fi diferenţiate după rularea unei suite de test): mai exact minimizarea numărului de astfel de blocuri.

Bibliografie

[ANL10] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 255–264, New York, NY, USA, 2010. ACM.

[ASM11] Asm. http://asm.ow2.org/, June 2011.

[AZvG07] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 89–98, Washington, DC, USA, 2007. IEEE Computer Society.

[Bal08] Melinda-Carol Ballou. Improving software quality to drive business agility. Technical report, IDC, 2008.

[BCE11] Bcel - the byte code engineering library. http://mercurial.selenic.com/, June 2011.

[BFLT06] Benoit Baudry, Franck Fleurey, and Yves Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 82–91, New York, NY, USA, 2006. ACM.

[Bru07] Eric Bruneton. Asm 3.0 a java bytecode engineering library. Technical report, OW2, 2007.

[CKF+02] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[CLM+09] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 34–44, Washington, DC, USA, 2009. IEEE Computer Society.

[EB03] Thierry Coupay Eric Bruneton, Romain Lenglet. Asm: a code manipulation tool to implement adaptable systems. Technical report, France Telecom R&D, 2003.

[ECH11] Eclipse documentation - current release. http://help.eclipse.org/helios/, June 2011.

[FBG11] Findbugs - find bugs in java programs. http://findbugs.sourceforge.net/, June 2011.

[GGC11] Guice - a lightweight dependency injection framework for java 5 and above, brought to you by google. http://code.google.com/p/google-guice/, June 2011.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[JH05] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[JHS02] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 467–477, New York, NY, USA, 2002. ACM.

[JNT11] Junit.org. http://www.junit.org/, June 2011.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29:782–795, September 2003.

[LFY+06] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and Samuel P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Trans. Softw. Eng.*, 32:831–848, October 2006.

[Lib11] Ben Liblit. What is statistical debugging? http://stackoverflow.com/questions/505907/what-is-statistical-debugging, June 2011.

[Liu06] Chao Liu. Fault-aware fingerprinting: Towards mutualism between failure investigation and statistical debugging. In *Urbana*, volume 51, 2006.

[LNZ+05] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. *SIGPLAN Not.*, 40:15–26, June 2005.

[LY99] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[MER11] Mercurial - work easier, work faster. http://mercurial.selenic.com/, June 2011.

[MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[Pas10] Fabrizio Pastore. *Automatic Diagnosis of Software Functional Faults by Means of Inferred Behavioral Models*. PhD thesis, Universit degli Studi di Milano Bicocca, 2010.

[SB09] Friedrich Steimann and Mario Bertschler. A simple coverage-based locator for multiple faults. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 366–375, Washington, DC, USA, 2009. IEEE Computer Society.

[SER11] Serp. http://serp.sourceforge.net/, June 2011.

[SHK11] Sherlockj. http://code.google.com/p/junit-debugger/, June 2011.

[TNG11] Testng. http://www.testng.org, June 2011.

[TST11] Testability explorer - explore the testability of open-source and commercial java projects. http://www.testabilityexplorer.org/report, June 2011.

[YKJ11] Yourkit - the industry leader in .net & java profiling. http://www.yourkit.com/java/profiler/, June 2011.

[Zel01] Andreas Zeller. Automated debugging: Are we close. *Computer*, 34:26–31, November 2001.