Testarea software orientat obiect

Prof.dr. Ion IVAN, stud. Paul POCATILU
Catedra de Informatica Economica, A.S.E. Bucuresti

Generatia a patra de software este rezultatul analizei, proiectarii si programarii orientate obiect. Tehnologiile orientate obiect genereaza particularitati care se reflecta în toate etapele ciclului de viata software, inclusiv în testare. Pornind de la testarea empirica, pâna la utilizarea unor tehnici si metode moderne, viziunea orientata obiect genereaza noi abordari. Testarea se face atât pentru un produs software luat ca un întreg, cât si pentru componentele acestuia. Daca se testeaza sistemul de programe procesul se desfasoara da la simplu spre complex. Se testeaza mai întâi modulele apoi programul si în final sistemul de programe ca ansamblu complex. Rezultatele asamblarii, software integrat, este testat în continuare (integration testing). Atingerea tuturor ramurilor din fiecare modul presupune dezvoltatea grafului asociat modulului si corespunzator generarea de date de test care sa activeze nodurile grafului. Testarea software are un caracter statistic prin activarea selectiva a unor ramuri din structura arborescenta asociata programului.

Cuvinte cheie: *testare software, programare orientata obiect, metode de testare.*

1. Introducere

Testarea este un proces cu periodicitate, însotit de efect de ondulanta. Eliminarea tuturor erorilor este însotita de costuri mari. De aceea se impune gasirea unui echilibru între numarul erorilor ramase în software si costul eliminarii.

În conditiile dezvoltarii software dupa toate cerintele definite prin tehnologii, testarea software este nenecesara. Cum însa aceste conditii nu sunt îndeplinite integral, testarea apare ca singura posibilitate de a face software-ul operational. Este componenta a etapelor procesului de dezvoltare software. Fiecare etapa aduce erori specifice, ceea ce implica modalitati, de asemenea, specifice, de testare.

Realizarea de software fiabil este conditionata de efectuarea unei testari cât mai riguroase. Cu cât procesul de testare este mai riguros, cu atât costurile care se propaga în timp la utilizatori sunt mai reduse. Aceleasi efecte le resimte însusi producatorul. Crearea de software orientat spre reutilizare 100 % schimba atât atitudinea producatorului cât si pe cea a echipei de testare, acordând o atentie speciala produselor care se realizeaza. Oricare dintre metodele de testare software orientat obiect este folosita, rezultatele ei vor fi benefice cu conditia deprinderii subtilitatilor care evidentiaza erori, tipuri de erori si mai ales modalitati de propagare.

Metricile testarii evidentiaza caracteristici de calitate software si progresele acestora ca rezultate ca rezultat al procesului de testare-corectie.

2. Testarea software orientat obiect

Programarea orientata obiect pesupune definire de clase si referire de obiecte.

Clasele sunt definite, caz în care reutilizarea genereaza efecte pozitive. Când clasele sunt subdefinite este necesara construirea de clase derivate de completare. Când clasele sunt supradefinite apar restrictii de referire si de reutilizare. Testarea claselor este premisa reutilizarii neconditionate. Testarea claselor evidentiaza raportul operanzi–operatori, gradul de mostenire si de acoperire probabila a tipologiilor de clase de probleme prin constructori/destructori definiti. Programarea orientata obiect este caracterizata printr-un nivel foarte ridicat al reutilizarii software.

Numeroase aplicatii complexe se prezinta sub forma unor programe apelatoare (functii

main) care contin lungi secvente de directive **#include** si ca instructini executabile, structuri liniare de expresii de referire a functiilor membre ale obiectelor folosite.

Testarea software orientat obiect presupune doua planuri:

- testarea constructiilor proprii;
- testarea constructiilor incluse pentru reutilizare.

Testarea software-ului orientat obiect are pe lânga obiectivul general al stabilirii masurii în care produsul software realizeaza sarcinile date în specificatii, obiective specifice legate de:

- testarea functiilor membre ale fiecarei clase;
- testarea gradului de încapsulare si a efectelor acestuia;
- testarea efectelor induse de nivelele de mostenire si de derivare;
- testarea efectelor induse de polimorfismul functiilor membre.
- testarea interactiunilor dintre clase.

Spre deosebire de software-ul dezvoltat prin alte metode, în cazul programarii orientate obiect, testarea vizeaza si masura în care clasele sunt proiectate în vederea reutilizarii. Adica, se evidentiaza gradul de generalitate si mai ales concordanta dintre specificatiile fiecarei functii si ceea ce efectiv functia realizeaza.

Rezultatul testarii vizeaza doua aspecte si anume:

- secventa referirilor determina pentru exemplele de test rezultate acceptabile sau nu ceea ce se rasfrânge asupra produsului ca atare;
- rezultate privind masura în care clasele definite sau referite acopera cerintele unei reutilizari confortabile, fara nevoia de a construi interfete sau de a realiza derivari în obtinerea de clase noi cu un nivel de saturare redus, create în mod special si destinate unor utilizari cu totul particulare.

Daca produsul poate fi acceptat pentru calitatile lui în raport cu problema de rezolvat, nu este obligatorie si acceptarea claselor definite. În acelasi fel, clasele definite pot îndeplini conditiile de reutilizare, fara ca programul sa fie acceptat în urma testarii.

Metoda de testare ierarhica sta la baza sistemului de testare orientat obiect. Aceasta metoda de testare utilizeaza si construieste pe baza unor tehnici de testare cunoscute, legându-le împreuna într-un sistem de testare corespunzator. Metoda defineste si aplica standarde de testare pentru câteva nivele ale componentelor software: obiecte, clase, componente de baza si sisteme [SIEG96].

Metoda de testare ierarhica desemneaza ca fiind "sigure" acele componente care îndeplinesc standardele de testare pentru tipul respectiv de componenta. Odata ce o componenta a fost desemnata ca "sigura", ea poate fi integrata cu alte componente "sigure" pentru a realiza împreuna componentele de pe nivelul urmator.

"Sigur" este o stare relativa. Depinde în întregime de:

- standardele alese pentru realizarea aplicatiei;
- atitudinea fata de risc;
- riscurile specifice si practicile de management al riscului adoptate în proiect.

Metoda de testare ierarhica se axeaza pe componentele de baza. O componenta de baza poate fi o ierarhie completa de clase sau aumite clustere de clase care realizeaza o functie de baza sau care reprezinta o componenta arhitecturala logica sau fizica.

Dupa ce este testata o componenta de baza la un nivel "sigur", ea poate fi integrata cu alte componente de baza. Testarea de integrare a componentelor de baza "sigure" necesita accesarea doar a interconexiunilor dintre componentele de baza si orice functionalitate complexa noua. Metoda ierarhica elimina obligativitatea testarii tuturor combinatiilor de stari în timpul testarii de integrare, ceea ce duce la cresterea productivitatii. În figura 1 este o reprezentare grafica a metodei ierarhice

[SIEG96].

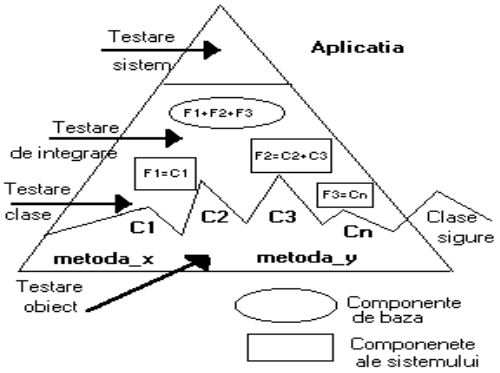

Fig. 1. Metoda ierahica de testare

La baza piramidei sunt testate complet metodele individuale pentru a fi denumite "sigure". Metodele se integreaza în clase si se va testa întreaga ierahie de clase pentru a se atinge un nivel "sigur". Colectia de clase "sigure" (componentele de baza) formeaza baza produsului final. Aria larga de integrare a piramidei este locul în care se integreaza componentele de baza in diferite structuri arhitecturale, functionale si logice. Ele formeaza un sistem. Vârful piramidei este un test de sistem scurt si concentrat. Urmatoarea suita de teste completeaza structura piramidei:

- **suita de teste conditionale** testeaza clasele utilizând modelul de testare conditionala si asertiunile, exceptiile, operatiile de testare concurente aditionale;
- **suita de teste ierarhice incrementale** testeaza componentele de baza utilizând diverse modele de testare si scenarii;
- **suita de teste de integrare** testeaza combinatii de componente de baza utilizând modelul ierarhic incremental cu scenarii care utilizeaza toate componentele, nu doar **stub**-uri;
- **suita de teste de sistem** testeaza sistemele componentelor de baza utilizând modele de testare de sisteme;
- **suita de teste de regresie** testeaza componentele de baza si sistemul.

Suita de teste ierarhice incrementale este cea mai complexa dintre suitele de testare. Se poate construi suita utilizând oricare din urmatoarele modele de testare:

- modelul de testare bazat pe stari de tranzitie;
- modelul de testare bazat pe fluxul de tranzactii;
- modelul de testare bazat pe exceptii;
- modelul de testare bazat pe fluxul de control;
- model de testare bazat pe fluxul datelor.

Suita de teste ierahice incrementale este un set de scenarii de test care trateaza obiectele ca instante ale claselor si ca parte a unui sistem corelat cu alte sisteme. Termenul ierahice si incrementale exprima scopul de a testa obiectul în contextul ierahiei de mosteniri si a piramidei sistemului. Astfel orice obiect nu consta numai în proprietatile care i se atribuie ci, de asemenea, si în acelea pe care le mosteneste de la ascendentii sai mai generali si mai abstracti. Trebuie deci ca operatiunea de testare a obiectului sa aiba în vedere acest lucru.

La o extrema poate exista o singura serie de test ierarhice si incrementale pentru întregul sistem; la cealalta extrema o suita pentru fiecare clasa. Solutia optima este undeva la mijloc.

O modalitate potrivita de a începe este aceea de concentrare asupra sistemului, acesta continând de obicei toate clasele ce pot fi subiectul unui singur grup de teste. Este mai logic în unele cazuri sa se lucreze cu subseturi de clase într-un sistem, acest lucru fiind similar cu conceptul de testare a clusterelor în timpul integrarii.

Este important sa se faca diferenta dintre clasele concrete si cele abstracte. Putem testa clasele concrete direct, lucru pe care nu îl putem face în cazul claselor abstracte datorita faptului ca acestea nu pot fi instantate. Structural, scenariile se construiesc separat pentru clasele abstracte, dar se ruleaza împreuna cu testele pentru subclase concrete. Pastrarea scenariilor de testare separat permite reutilizarea testelor pentru subclase diferite profitând din plin de mostenire pentru reutilizarea lor. Când se

instanteaza un obiect si se executa teste asupra sa, acestea testeaza si clasa a carei instanta este obiectul si superclasa sa urcând în piramida.
Esenta metodei ierarhice si incrementale de testare este ansamblul de relatii dinte clase. În lumea orientat obiect, când se testeaza o clasa se testeaza în acelasi timp si clasele parinte si, construind teste, acestea pot fi reutilizate ulterior si pentru testarea subclaselor.
Fiecare subclasa este rezultatul combinarii structurii si comportamentului claselor parinti cu atribute si metode noi. În figura 2 este o reprezentare grafica din care se observa cum prin mostenire, clasa derivata D se obtine prin combinarea clasei de baza B cu un modifcator, M, ce cuprinde metodele si proprietatile specifice clasei derivate. Astfel se poate scrie ca $D = B \oplus M$ [HARR92].

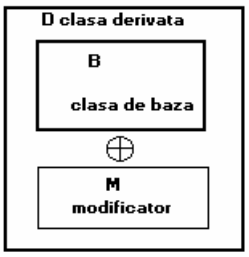

Fig. 3. Mostenirea, o tehnica de modificare incrementala

În cadrul unei clase derivate au fost distinse trei metode:

- **metode noi,** în subclase, incluzându-le pe acelea cu acelasi nume ca metoda din superclasa dar cu parametri diferiti;
- **metode recursive,** definite într-o superclasa si care nu sunt supradefinite sau supraîncarcate în subclasa;
- **metode redefinite,** definite într-o superclasa si supradefinita sau supraîncarcata într-o subclasa.

Metodele abstracte sunt metode de orice tip asociate cu clase abstracte. Metodele abstracte pot sau nu sa aiba o implementare. Daca nu au, subclasle trebuie sa redefineasca metoda, supraîncarcând-o. Fiecare din aceste metode are o cerinta diferita de testare în contextul clasei, de aceea este importanta aceasta tipologie. Astfel:

- metodele noi – necesita o testare completa;
- metodele recursive – necesita o retestare limitata daca metoda interactioneaza cu functii membre noi sau redefinite. Este nevoie doar sa fie retestat obiectul care interactioneaza, nu toate obiectele;
- metodele redefinite – se retesteaza prin reutilizarea modelelor de teste si obiecte dezvoltate din specificatii mai degraba decât din controlul logic intern.

Acest lucru înseamna ca nu este nevoie sa se testeze fiecare tip de obiect în parte daca acesta ar fi unul complet nou; va fi nevoie sa se testeze doar partile într-adevar noi. Acest lucru creste simtitor productivitatea testarii, precum si productivitatea programarii si a proiectarii.
Reutilizarea structurala a acestui tip este partea cheie a paradigmei orientarii obiect.
Planul de testare ar trebui sa corespunda listei detaliate de clase si metodele de testare.
Mostenirea multipla face clasificarea mai dificila, dar nu afecteaza categoriile în sine. Anumite modele de mostenire multipla, cum ar fi cele din C++, fac sa existe aceeasi superclasa în doua sau mai multe instante, în obiect, creând astfel mai mult decât o metoda recursiva. Astfel se poate numi metoda "diferita" o metoda care este de fapt aceeasi doar atribuind-i calea superclasei. Atâta timp cât metoda este recursiva nu este nevoie ca acest lucru sa fie testat. Daca metoda este redefinita, nu va mai fi nevoie sa fie testate cele ale superclaselor, ci doar redefinirea. Singura situatie în care ar putea fi nevoie de o testare a mai multor cai diferite cu aceesi metoda este atunci când metoda interactioneaza cu metode virtuale redefinite într-o subclasa.
Este de remarcat faptul ca legarea dinamica joaca un rol în determinarea cerintelor testarii, dar numai în contextul testarii de

integrare. Legarea dinamica creeaza posibilitatea unui set de mesaje (combinatii de obiecte emitatoare si receptoare de mesaje), ceea ce înseamna ca va fi nevoie de mai multe teste în locul unuia singur pentru a testa un mesaj specific. Când doar se testeaza metode, ca fiind opuse mesajelor, nu prezinta interes altceva decât varietatea de valori posibile ale datelor pe care mesajele le pot introduce în metode.

În functie de tipul modelului de test utilizat si de dezvoltarea suitei de test, ar putea fi necesara adaugarea unor obiecte de test conditilor testelor care apar în metodele recursive. De exemplu daca o metoda recursiva utilizeaza unele atribute si subclasele adauga o noua serie de valori acelor atribute, ar trebui adaugate obiecte de testare si scenarii pentru a testa noua serie ca parte a unui model de flux de control sau model de flux de date. Aceasta situatie nu este tocmai obisnuita, dar întotdeauna trebuie reexaminate modelele de testare pentru superclase pentru a fi sigur faptul ca acoperirea este adecvata.

Concluzii

Testarea are un loc bine precizat în structura ciclului de dezvoltare software. Functie de modul de organizare, de obiective, rezultatele testarii conduc la clase de software diferentiat în raport cu riscul prelucrarilor. Astfel, software de clasa A este obtinut prin testare completa si functioneaza independent de utilizator fara riscul de a include prelucrari nefinalizate. Calitatea rezultatelor depinde exclusiv de utilizator. Programele de clasa B sunt rezultatul unei testari aproape complete dar exista riscul de a se produce întreruperi. Remedierile se efectueaza în timp cu posibilitatea trecerii la clasa A.

Programele de clasa C sunt testate dar nu este asigurata integral posibilitatea de a functiona independent de utilizator. Prin dezvoltarea interfetei, prin cresterea fiabilitatii se poate trece spre clasa B. Testarea este necesara pentru a stabili nivelurile caracteristicilor de calitate pentru programul care se livreaza si pentru a le compara cu nivele planificate, respectiv nivele anuntate de elaborator. Programele efectueaza pe lânga prelucrarile din specificatii si alte prelucrari complementare. Rolul testarii este de a evidentia combinatii de optiuni care genereaza aceste prelucrari. Testarea este absolut necesara în procesul industrializarii productiei de software. Lucrul în echipe de programare presupune predare si preluare de module. Testarea este singurul mod de a verifica un modul înainte de predare repcum si la predarea de catre producator a produsului software. Eficienta testarii influenteaza toate fazele ciclului de dezvoltare software. Revenirile la fazele precedente sunt rezultatul concluziilor ce se desprind din testare. Rigurozitatea si gradul de acoperire scurteaza distanta dintre punctul de unde se efectueaza revenirea si faza care necesita modificari. Costurile testarii se reflecta direct în structura cheltuielilor atât la producatorii de software cât mai ales la utilizatori. Daca s-a acordat atentie testarii, costurile la producator sunt cosiderabile, însa ele se recupereaza prin efectele pozitive de la nivelul utilizatorilor, efecte care sunt bazate pe calitatea produselor, iar aceasta cu cât este mai ridicata cu atât presupune un cost stimulativ mai ales pentru utilizator. Cu atât mai mult testarea software orientat obiect este mai necesara, întrucât însasi conceptia de orientat obiect este bazata pe ideea de reutilizare software. Ori, poate fi reutilizat numai software în care exista încredere si aceasta se construieste numai si numai prin procesul de testare.

Bibliografie

[BOOC91] Booch, Grady, *Object Oriented Design with Applications*, The Benjamin/ Cummings Publishing Company, Inc., 1991.

[CHOW85] Chow, Tsun S., *Technical issue: Testing and Validation* in *Tutorial software quality assurance, a practical approach*, pp. 269-274, edition IEEE Computer Press, 1985.

[HARR92] Harrold, Mary Jean, McGregor, John D., Incremental Testing of Object Oriented Class Structures, 1992.

[IEEE94] IEEE, *IEEE Standards Collection: Software Engineering*, IEEE, 1994

[INFO79] *Software Testing: State of the Art Report, vol 1: Analysis and Bibliography*, Infotech International, Edition : Anne E Westley, 1979.

[INFO79a] *Software Testing: State of the Art Report, vol 2: Invited Papers*, Infotech International, Edition : Anne E Westley, 1979.

[McGR97] McGregor, John D., *Quality Assurance: An overview of testing*, Journal of Object-Oriented Programming, vol.9, No. 8,1997.

[McGR97a] McGregor, John D., *Quality Assurance: Planning for testing*, Journal of Object-Oriented Programming, vol. 9, No. 9, 1997, pp 8-12.

[McGR97b] McGregor, John D., *Quality Assurance: Component testing*, Journal of Object-Oriented Programming, vol. 10, No. 1, 1997, pp 6-9.

[McGR97c] McGregor, John D., *Quality Assurance: Parallel architecture for component testing*, Journal of Object-Oriented Programming, vol. 10, No. 2, 1997, pp 10-14.

[McGR97d] McGregor, John D., *Quality Assurance: A component testing method*, Journal of Object-Oriented Programming, June 1997.

[McGR97e] McGregor, John D., Quality Assurance: *Making component testing more effective*, Journal of Object-Oriented Programming, July 1997.

[McGR97f] McGregor, John D., *Quality Assurance: Testing from specifications*, Journal of Object-Oriented Programming, August 1997.

[McGR97h] McGregor, John D., *Quality Assurance: Building tests from specifications*, Journal of Object-Oriented Programming.

[RUMB91] Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick, Lorensen, William, *Object Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[SIEG96] Siegel, Shel, *Object Oriented Software Testing: A Hierarchical Approach*, Wiley Computer Publishing, 1996.

[SPIR95] Spircu, Claudia, Lopatan, Ionut, *POO Analiza, proiectarea si programarea orientate spre obiecte*, Teora, 1995.