Calitatea sistemelor software

10.1. Indicatori de calitate

moduri în care se defineşte în acest calitatea sistemelor software:

- "ceva potrivit pentru utilizare",
- "satisfacerea cerinţelor utilizatorilor"
- "absenţa defectelor" (Grady, 1993).

Unul dintre principalele obiective ale ingineriei software este de a pune la dispoziţia dezvoltatorilor metodologii şi instrumente pentru a realiza software de mai bună calitate.

Tabelul 10.1 elaborat de Mayer, prezintă cu titlu de exemplu zece factori de calitate ai software-ului.

Calitatea va fi adesea rezultatul unui compromis.

Când se lucrează fără un instrument de realizare, acest compromis se face într-o manieră inconsistentă de către programator, ceea ce nu este de dorit.

Când se lucrează cu un instrument software, aceasta se face într-o manieră mai mult sau mai puţin voluntară, de către constructorul instrumentului software respectiv.

Între eficacitate şi fiabilitate, majoritatea instrumentelor au ales deja, dar mai trebuie ca nivelul de compromis care a fost adoptat să fie şi acceptat de cei care cumpără instrumentul software respectiv.

Tabelul 10.1. **Factorii de apreciere ai calităţii - B. Mayer**

Factor	Definiţii
Validitate	Aptitudinea produsului software de a îndeplini exact funcţiile sale, definite prin caietul de sarcini şi prin specificare.
Fiabilitate	Aptitudinea produsului software de a funcţiona în condiţii anormale
Extensibilitate	Uşurinţa cu care un software se pretează la o modificare sau la o extindere a funcţiilor solicitate.
Reutilizabilitate	Aptitudinea unui produs software de a fi reutilizat, în totalitate sau şi parţial, într-o nouă aplicaţie.
Compatibilitate	Uşurinţa cu care un sistem poate fi combinat cu altele
Eficacitate	Utilizarea opţională a resurselor materiale (procesoare, memorie internă şi externă, protocoale de comunicaţie, etc)
Portabilitate	Uşurinţa cu care un produs poate fi transferat pe medii diferite hardware şi software.
Verificabilitate	Uşurinţa cu care se pregătesc procedurile şi testele de validare (în particular "jocurile de încercare") şi procedurile de detectare a erorilor care urmează unui incident în exploatare.
Integritate	Aptitudinea software-ului de a-şi proteja codul şi datele de accesări neautorizate.
Uşurinţa de utilizare	Facilitatea de înţelegere, de utilizare, de pregătire a datelor, de interpretare a erorilor, de revenire în caz de utilizare eronată.

Ergonomia este pe punctul de a deveni un criteriu foarte important pentru funcţionalitatea însăşi a sistemului, pentru că, fără îndoială, se consideră drept “competenţă“ a unui sistem dacă el se achită corect de sarcina sa (criteriul de validitate).

10.2. Productivitatea

Al doilea obiectiv al unui instrument software este de a produce software cât mai repede posibil .

Cum calitatea şi productivitatea acoperă aspecte diferite, ele presupun adesea un compromis.

Dacă se vizează productivitatea în mentenanţă, va trebui fără îndoială să se investească mai mult în concepţie, pentru a găsi structuri de date mai generale şi să se determine ceea ce poate fi parametrizat.

Dar aceasta se va face în detrimentul termenului de livrare al software-ului.

Dacă, din contra, obiectivul este să fie “ceva“ care funcţionează, modul de abordare este altul.

Sunt situaţii în care nu există altă alegere sau sunt cazuri extreme care pot justifica existenţa chiar a două instrumente de dezvoltare, unul care să producă mai repede, dar care furnizează un cod mai puţin eficient sau mai puţin mentenabil, celălalt care permite să se lucreze mai riguros, atunci când este timp pentru aceasta.

Productivitatea are punct de conflict cu calitatea atunci când se atinge un nivel minim al acesteia din urmă, sub care nu se poate coborî.

Productivitatea unui proiect trebuie să se măsoare în mod global, la capătul mai multor ani de utilizare care să includă toate fazele, de la concepţie la mentenanţă.

În practică, se întâmplă relativ rar să se măsoare productivitatea în afara fazei de realizare, ceea ce este cu certitudine interesant, dar nu reprezintă decât o mică parte din ceea ce trebuie măsurat.

Cu cât productivitatea de realizare a unui proiect este mai mare, cu atât timpul de livrare al produsului scade şi implicit şi costul produsului va fi mai mic.

Mai mult, interesul producătorilor de software pentru achiziţionarea de instrumente performante de lucru care să mărească productivitatea va fi în continuă creştere.

10.3. Asigurarea fiabilităţii produselor software

10.3.1. Niveluri prescrise de fiabilitate

Încă din faza stabilirii cerinţelor şi specificaţiilor pentru o aplicaţie informatică trebuie să se impună nivelul de fiabilitate al sistemului ca un compromis între preţul de cost şi consecinţele defectărilor.

Clasele de fiabilitate ale produselor software sunt următoarele:

a) Foarte scăzută (very low). Acest nivel se prescrie atunci când defectarea are ca singură consecinţă inconvenientul producătorului de a înlătura o neregulă în program. Asemenea situaţie intervine, de exemplu în cazul modelelor demonstrative sau al simulărilor.

b) Scăzută (low). Acest nivel corespunde în cazurile în care defectarea implică o pierdere mică, uşor de recuperat, pentru beneficiar. Sistemele utilizate în predicţie pe termen lung sunt exemple tipice.

c) Nominală (nominal). Acest nivel corespunde cazului când defectarea implică o pierdere moderată pentru beneficiar, dar remedierea situaţiei nu este prea costisitoare.

Sistemul de gestionare a stocurilor sau sistemele informaţionale de conducere sunt exemple tipice pentru această clasă de fiabilitate.

d) Ridicată (high). Efectele defectării pot implica o pierdere financiară mare sau un inconvenient major pentru factorul uman. Exemple tipice sunt sistemele bancare şi cele ale distribuţiei energiei electrice.

e) Foarte ridicată (very high). Efectele defectării pot consta în pierderi de vieţi omeneşti. Cazul tipic îl constituie sistemul de control al reactoarelor nucleare.

În funcţie de nivelul de fiabilitate prescris, se poate stabili ce efort necesar să fi alocat în fiecare etapă a realizării produsului.

În fig. 10.1 se reprezintă acest efort normat la cel necesar obţinerii unei fiabilităţi nominale.

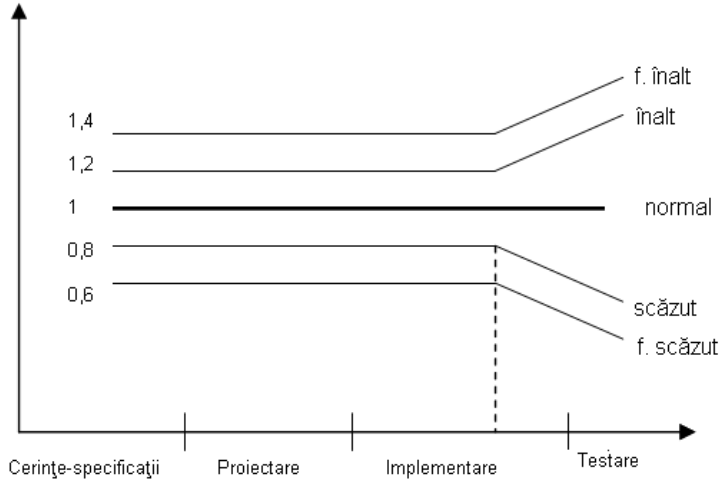

Fig.10.1 Reprezentarea efortului pentru asigurarea nivelului de fiabilitate prescris

10.3.2. Proiectarea structurii pentru asigurarea fiabilităţii

Asigurarea fiabilităţii la nivelul proiectului se bazează pe realizarea unei structuri cât mai simple şi transparente. Pentru a avea un control asupra acestor caracteristici trebuie să se cunoască:

P_1 = numărul total de module din program

P_2 = numărul de module dependente de intrări (I) sau ieşiri (E)

P_3 = numărul de module dependente de o procesare anterioară

P_4 = numărul de elemente din bazele de date

P_5 = numărul elementelor neunice din baza de date (BD)

P_6 = numărul de segmente din baza de date (BD)

P_7 = numărul de module cu mai mult de o intrare şi o ieşire

Cu ajutorul mărimilor P_1 - P_7 se evaluează şase mărimi derivate D_1 - D_6 care exprimă simplitatea structurii programului. Cu cât valorile D_i sunt mai mari, cu atât este mai îndoielnică fiabilitatea programului.

D_1 este o variabilă binară egală cu zero dacă proiectarea este descendentă (top-down) şi cu 1 în caz contrar;

D_2 exprimă dependenţa la nivelul modulelor;

D_3 exprimă dependenţa de procesarea anterioară;

D_4 arată mărimea bazei de date;

D_5 arată compartimentarea bazei de date;

D_6 arată mărimea interacţiunii programului cu exteriorul;

O sinteză a mărimilor D_1 - D_6 este dată de indicele de structurare a proiectului DSM, care are expresia:

$$\boxed{DSM = \sum_{i=1}^{6} w_i D_i \ , \sum_{i=1}^{6} w_i = 1}$$

unde w_i sunt ponderi alese în funcţie de importanţa fiecărei caracteristici D_i. Valori apropiate de 1 pentru D_1 - D_6, respectiv pentru DSM, indică o deficienţă a proiectului care trebuie corectată prin reproiectare, astfel încât să se micşoreze mărimile primare P_1 - P_7. Orice modificare în proiect trebuie analizată din unghiul de vedere al indicelui de structurare, pentru a decide în ce măsură modificarea propusă îmbunătăţeşte proiectul.

- Indicele de structurare permite de asemenea evaluarea comparativă a unor proiecte diferite.

10.3.3. Complexitatea fluxului informaţional

Până acum s-a avut în vedere structura proiectului privit din punct de vedere static. Abordând un punct de vedere dinamic, este necesar să se considere complexitatea fluxului de informaţie, care parcurge structura. În acest scop, se determină fluxul de informaţie între module, considerându-se că un flux de informaţie de la A la B are loc dacă:

(1) A apelează B sau,

(2) B apelează A şi A returnează o valoare utilizată ulterior de B sau,

(3) atât A cât şi B sunt apelaţi de un alt modul care face să treacă o valoare de la A la B.

Fie:

- $\Rightarrow$ l_{fi} nr. de fluxuri care intră într-o procedură (local flows into);
- $\Rightarrow$ l_{fo} nr. de fluxuri care iese dintr-o procedură (local flows out);
- $\Rightarrow$ datain = numărul structurilor de date din care o procedură îşi ia datele
- $\Rightarrow$ dataout = numărul structurilor de date care sunt actualizate de procedură
- $\Rightarrow$ lenght = numărul de instrucţiuni scrise într-o procedură (inclusiv comentariile).

Pentru a evalua complexitatea fluxului informaţional global se calculează numărul de căi informaţionale care intră în, respectiv ies din, module:

fanin = l_{fi} + datain;

fanout = l_{fo} + dataout

În ansamblu, complexitatea informaţională IFC este dată de expresia:

IFC = length (fanin * fanout)^2.

Cu ajutorul acestei caracterizări se pot identifica acele module care, având o complexitate informaţională mai mare, sunt probabil afectate de erori, deci au o fiabilitate mai redusă.

Se identifică, de asemenea, procedurile care au mai mult decât o funcţie, punctele de trafic informaţional intensiv şi modulele cu o complexitate funcţională excesivă.

Examinarea complexităţii informaţionale trebuie întreprinsă în faza proiectării detaliate şi în faza integrării produsului.

10.4. Metrici software pentru paradigma orientată pe obiect

Acestea nu sunt atât de numeroase ca în cazul paradigmei procedurale.

S-au propus (Chidamber şi Kemerer, 1991) şase metrici de proiectare orientate obiect şi anume:

- DIT (Depth of the Inheritance Tree) - adâncimea arborelui de moştenire;
- NOC (Number of Children) - numărul de moştenitori;
- CBO (Corepling Between Object) - cuplarea dintre obiecte;
- RFC (Response For a Class) - răspunsul pentru o clasă;
- LCOM (Lack of Cohesion of Methods) - lipsa coeziunii metodei
- WMC (Weighted Method per Class) - ponderea metodei în clasă.

a). Metrica **DIT** măsoară poziţia clasei în ierarhia de clase. Se adresează conceptului de moştenire din OOP. Se poate face ipoteza că cu cât este mai mare metrica **DIT** cu atât este mai greu de menţinut clasa. Valoarea metricii **DIT** este dată de numărul nivelului clasei în ierarhia de clase. **DIT-ul** pentru rădăcină este zero.

DIT = numărul de nivel de moştenire

$$DIT \in [\,0, N\,], \quad N \geq 0$$

b). Metrica **NOC** măsoară numărul de moştenitori direcţi ai unei clase. Se are în vedere acelaşi concept de moştenire din OOP dar se consideră că cu cât numărul de moştenitori direcţi ai unei clase este mai mare cu atât este mai afectat potenţialul de moştenire.

De exemplu, dacă sunt mai multe subclase a unei clase care au dependenţe de metode sau instanţe de variabile definite în superclase atunci orice schimbări în aceste metode sau variabile afectează subclasele.

Se poate spune că cu cât este mai mare metrica NOC cu atât este mai greu de menţinut clasa. Deci:

NOC = numărul subclaselor directe

$NOC \in [0, N]$, $N > 0$

c). Metrica **RFC** măsoară cardinalitatea clasei, răspunsuri ale unei clase. Mulţimea de răspunsuri ale unei clase constă din toate metodele locale şi toate celelalte metode apelate de metodele locale. Pare intuitiv că, cu cât este mai mare mulţimea de răspunsuri, cu atât mai complexă este clasa.

Deci cu cât este mai mare metrica RFC cu atât mai dificil este de menţinut clasa din cauza apelurilor unui număr mare de metode în răspunsul cărora se pot depista cu dificultate erorile.

RFC = numărul de metode locale + numărul de metode apelate de metodele locale.

RFC $\in$ [0, N] ; N > 0

d). Metrica **LCOM** măsoară lipsa coeziunii clasei. Coeziunea unei clase este caracterizată prin cât de strâns sunt legate metodele locale de variabile locale instanţiate în clasă. Această metodă se adresează conceptului de metodă din OOP.

Pare logic că o clasă care are o mai mare coeziune este mai uşor de întreţinut.

Deci cu cât metrica LCOM este mai mare cu atât este mai dificil de întreţinut clasa, deoarece dacă toate metodele definite din clasă accesează mai multe seturi independente de structuri de date încapsulate în clasă, clasa nu poate fi bine proiectată şi partiţionată.

LCOM = numărul de seturi disjuncte din metodele locale.

Nu există intersecţie a două mulţimi şi nici două metode care să aibă în comun o variabilă locală în domeniul [0, N] , N > 0.

e). Metrica **WMC** măsoară complexitatea statică a tuturor metodelor. Intuitiv, cu cât există mai multe metode, cu atât mai complexă este clasa. De asemenea, cu cât este mai mare fluxul de control al metodelor unei clase cu atât mai dificil este de înţeles şi de întreţinut.

WMC = suma complexităţilor ciclice McCabe din metodele locale.

$WMC \in [\ 0, N\ ]\ ,\ N > 0.$

McCabe a definit complexitatea ciclică ca o măsură bazată pe controlul fluxului în proceduri/funcţii. Complexitatea ciclică se bazează pe complexitatea din graful direct.

Pentru programele structurate o echivalenţă mai simplă pentru complexitatea ciclică este contorizată de condiţiile booleene simple din structurile de control (adică while, if, case, do-while, for, etc).

McCabe (1989) a extins apoi complexitatea ciclică pentru a măsura diagrama de structură de proiectare.

10.4.1. Metrici de definire şi adăugare orientate pe obiect

Două obiecte se spune că sunt cuplate dacă ele acţionează unul cu celălalt.

Formele de cuplare ale obiectelor sunt: cuplare prin:

- moştenire
- transmitere de mesaje
- date abstracte.

Cuplarea prin moştenire

Moştenirea promovează reutilizarea software-ului în metodele orientate pe obiect, dar creează de asemenea posibilitatea violării încapsulării şi ascunderii informaţiei.

Aceasta apare deoarece proprietăţile din superclase sunt expuse subclaselor fără nici o restricţie de acces.

Utilizarea moştenirii, dacă nu este bine proiectată, poate introduce complexitate suplimentară în sistem, datorată atributelor care sunt încapsulate în superclasă care sunt expuse fără restricţii accesului din subclase.

Mai mult, o clasă moşteneşte mai multe atribute neprivate decât accesează.

DIT (adâncimea arborelui de moştenire) şi NOC (numărul de moştenitori) sunt folosite pentru măsurarea caracteristicilor de moştenire.

DIT indică câte subclase are o clasă, arătând astfel câte clase depind de o anumită clasă. *NOC* indică câte clase pot fi direct afectate de o clasă.

Cuplarea prin transmiterea de mesaje

Un canal de comunicare în tehnologia OOP este transmiterea de mesaje.

Când un obiect are nevoie de servicii de la alt obiect, el transmite un mesaj.

Mesajul se compune de obicei din identificatorul obiectului, serviciul (metoda) solicitată şi lista de parametrii pentru metodă.

Cuplarea prin mesaje (MPC – Message Passing Coupling) este utilizată pentru măsurarea complexităţii mesajului care trece prin clase.

Deoarece tipul unui mesaj este definit de o clasă şi utilizat de obiectele clasei, metrica MPC dă de asemenea un indiciu despre câte mesaje trec printre obiectele clasei.

MPC = numărul de "instrucţiuni" definite şi transmise într-o clasă.

Numărul de mesaje transmise în afară de o clasă poate indica cât de dependentă este implementarea unei metode locale de alte metode din alte clase.

Aceasta nu poate fi sugestivă pentru numărul mesajelor recepţionate de o clasă.

Cuplarea prin tipuri abstracte de date

Conceptul de tip de dată abstractă (ADT) a fost introdus de McGregor (McGregor, 1990), iar clasa poate fi privită ca o implementare a ADT-ului.

O variabilă declarată în interiorul unei clase X poate avea tipul ADT care, este o altă definiţie a clasei, determinând astfel un tip special de cuplare dintre X şi cealaltă clasă, pentru că X are acces la proprietăţile clasei ADT.

Acest tip de cuplaj poate produce violarea încapsulării dacă limbajul de programare permite accesul direct la proprietăţile private din ADT.

Metrica care măsoară complexitatea de cuplaj determinată de ADT este cuplajul prin date abstracte (DAC):

DAC = numărul de ADT-uri definite într-o clasă.

Numărul de variabile care au ca tip ADT poate indica numărul de structuri de date dependente de definiţiile altor clase.

O altă metrică utilizată este numărul de metode dintr-o clasă (NOM).

Deoarece metodele locale dintr-o clasă constituie interfaţă clasei, NOM serveşte drept cea mai bună metrică de interfaţă.

NOM = numărul de metode locale.

Numărul de metode locale definite într-o clasă poate să indice proprietatea de operare a unei clase.

Mai multe metode într-o clasă constituie o interfaţă mai complexă a clasei.

10.4.2. Metrici de dimensiune

Acest tip de metrici software au fost mult timp utilizate în aprecierea software-ului.

Metrica "linie de cod" - LOC este folosită pentru a măsura o procedură sau o funcţie, iar cumularea metricilor LOC din toate procedurile şi funcţiile este folosită pentru măsurarea programului.

Totuşi dimensiunea în OOP nu este întotdeauna bine stabilită.

În OOP în locul metricii LOC, care este calculată numărând simbolurile ";" dintr-o clasă, se va folosi metrica numită SIZE1 care face acelaşi lucru.

SIZE1 = numărul de ";" dintr-o clasă.

A doua metrică de dimensiune folosită este numărul de proprietăţi (incluzând atribute şi metode) definite într-o clasă.

Această metrică este SIZE2 şi este egală cu numărul de atribute plus numărul de metode locale.

Concluzii:

Pentru a fi utile aceste metrici trebuie incluse într-o analiză care să aibă la bază un model matematic, date şi instrumente.

Modelul poate fi unul statistic, dacă datele colectate sunt suficiente, (numeroase), iar instrumentele trebuie să aibă în vedere un anumit limbaj de programe orientat pe obiecte şi eventuale instrumente de "înregistrare" ale acestor metrici.

10.5. Modele de studiere “a posteriori” a fiabilităţii produselor software

În literatura de specialitate există o serie de metode şi modele care permit studierea fiabilităţii sistemelor software după ce ele au fost proiectate (a posteriori).

Aşa cum s-a arătat anterior, aceasta conduce la mărirea timpului de testare şi nu garantează în totalitate că la sfârşit s-au eliminat toate erorile.

Fiecare model prezentat are la bază câteva ipoteze de lucru.

MODELUL I

Se bazează pe următoarele ipoteze:

1. Există un număr dat de erori la momentul iniţial;
2. Rata de detecţie a erorilor este proporţională cu numărul erorilor reziduale[1];
3. Fiecare eroare descoperită este imediat corectată, astfel încât numărul erorilor conţinute în program scade cu o eroare la fiecare moment de detecţie;
4. Rata erorilor între două momente de detecţie succesive este constantă.

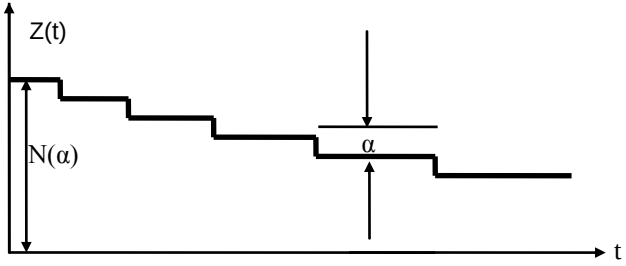

Fig.10.2. Curba lui z(t)

Pornind de la aceste ipoteze, rata de detecţie a erorilor z(t) se modelează cu relaţia:

[1] Erori reziduale = sunt acele erori care rezultă din diferenţa între valorile obţinute efectiv şi cele aşteptate a fi obţinute

$$z(t) = \alpha\ (N - d),$$

unde:

N - numărul iniţial al erorilor;

α - un coeficient de proporţionalitate, reprezentând decrementul lui z(t) corespunzător detecţiei şi corecţiei unei erori;

d - numărul de erori detectate şi corectate până la momentul t.

Fig. 7.6 reprezintă dependenţa lui z în funcţie de t.

MODELUL II

Acest model se bazează pe următoarele ipoteze:

1. Numărul de erori existente într-un program la începutul fazei de testare descreşte pe măsură ce erorile sunt corectate;

2. Numărul de erori reziduale Nr este egal cu diferenţa între numărul de erori iniţial prezente N şi numărul cumulativ de erori corectate Nc;

3. Numărul de instrucţiuni în limbaj maşină rămâne constant.

Modul de variaţie în timp a numărului de erori reziduale şi a celor corectate în conformitate cu acest model este reprezentat în Fig.10.3.

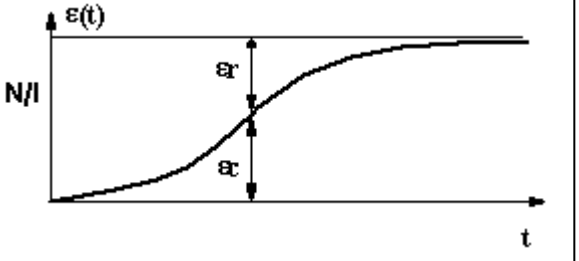

Fig.10.3. Curba erorilor reziduale

$$\varepsilon_r+\varepsilon_c=N/I$$

ε_c - număr de erori corectate pe instrucţiune

ε_r - număr de erori reziduale pe instrucţiune

În intervalul de timp (t, t+Δt) scurs după perioada de testare probabilitatea de a avea o eroare, P_e, este:

$$P_c = \gamma \cdot \varepsilon_r \cdot r_u \cdot \Delta t$$

unde γ este o constantă determinată de structura programului, iar r_u este rata de utilizare a unei instrucţiuni.

Dar :

$$P_e = z(t) \cdot \Delta t$$

$$z(t) = \gamma \cdot r_u \cdot (N/I - \varepsilon_c(\delta))$$

MODELUL III

Acest model, păstrând o parte din ipotezele şi formulările modelului I, este valoros prin aceea că permite o delimitare mai precisă a timpului (timp real de execuţie - adică timpul cât procesorul execută programul, în raport cu timpul calendaristic de dezvoltare a programului).

Modelul se bazează pe următoarele ipoteze principale:

1. Rata de corecţie a erorilor este proporţională cu rata de detecţie a erorilor;
2. Rata erorilor instantanee este proporţională cu numărul erorilor reziduale;
3. Manifestarea tuturor erorilor care intervin în cursul execuţiilor programului este efectiv observată.

Rata de detecţie a erorilor z(t) este modelată de relaţia:

$$z(t) = \varphi \cdot \alpha \cdot (N - d),$$

unde:

N - numărul iniţial al erorilor;

t - timpul cumulat de funcţionare al unităţii centrale;

α - coeficient de proporţionalitate, estimat;

φ - coeficientul frecvenţă de execuţie (definit ca raportul dintre numărul mediu de execuţii a unei instrucţiuni şi numărul total de instrucţiuni);

d - numărul de erori detectate şi corectate până la momentul t.

Numărul erorilor reziduale va fi:

$$N - n = N_0 \cdot \exp(-\varphi \cdot \alpha \cdot t),$$

iar timpul mediu de erori:

$$TMIE = \frac{e^{\varphi \alpha t}}{\varphi \cdot \alpha \cdot N_0}$$

În relaţiile anterioare N_0 reprezintă numărul de erori inerente (necorectate prin punerea la punct a programului).

MODELUL IV

În acest model erorile se clasifică în erori critice şi erori necritice; erorile critice sunt cele care conduc la oprirea misiunii în curs de execuţie, în care caz sistemul este indisponibil.

Compararea unui produs software în conformitate cu acest model este dată de graful din fig. 10.4. Se fac următoarele notaţii:

a - raportul dintre numărul de erori critice şi numărul total al erorilor;

λ - rata de detecţie a erorilor;

μ_c - rata de corecţie a unei erori critice;

μ_n - rata de corecţie a unei erori necritice.

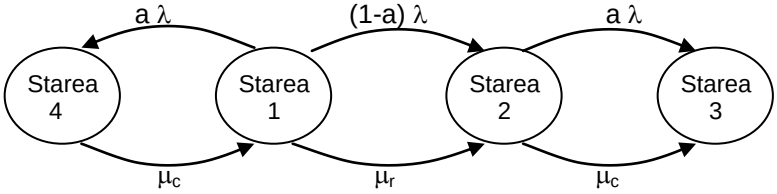

Fig.10.4. Graful de stare

Din graf se poate distinge existenţa a patru stări, care au semnificaţiile:

starea 1 - nu sunt erori, programul funcţionează conform specificaţiilor sale;

starea 2 -au fost detectate una sau mai multe erori necritice pentru program; se aşteaptă să se găsească un anumit număr de erori înainte de a le corecta pe toate;

starea 3 -a fost detectată o eroare critică, după detecţia unui anumit număr de erori necritice; aceasta este corectată imediat şi apoi programul revine la starea 2;

starea 4 -au fost detectate una sau mai multe erori critice; este acordată prioritate corecţiei acestora.

Aceste patru stări permit să se definească un model markovian de evoluţie a sistemului, pe baza căruia este posibilă determinarea indicatorilor de fiabilitate.

De exemplu, disponibilitatea unui modul de program poate fi exprimată prin ecuaţia:

$$\mathbf{A(t) = P_1(t) + P_2(t)}$$

unde $P_1(t)$ şi $P_2(t)$ sunt probabilităţile ca modulul să fie în stările 1 respectiv 2.

Pentru un număr de module M ale unui sistem, fiecare modul având disponibilitatea $A_i(t)$, disponibilitatea întregului sistem este:

$$A_S(t) = \prod_{i=1}^{M} A_i(t)$$

MODELUL V

Acest model presupune ipotezele:

1. Produsul software conţine un număr *n* de erori;

2. Fiecare eroare detectată este corectată, rata de detecţie fiind ca şi în cazul modelelor precedente proporţională cu numărul de erori rezultate;

3. Variabilele aleatoare, timpul de funcţionare fără eroare şi timpul de corecţie a erorii sunt distribuite exponenţial.

Evoluţia sistemului este descrisă de un model markovian, căruia îi corespunde graful de tranziţie din Fig. 10.5.

Se pot defini:

- Starea (n-k) - starea pentru care a fost corectată a (k-1) eroare şi nu a apărut încă eroarea numărului k;
- Starea (m-k) - starea pentru care a fost detectată, dar nu şi corectată, eroarea numărului k;
- $\lambda_{n-k}\Delta t$ - probabilitatea de trecere din starea (m-k) la starea (m-k-1), fiind rata de detecţie a erorii;
- $\mu_{m-k}\Delta t$ - probabilitatea de trecere din starea (n-k) în starea (n-k-1), fiind rata de corecţie a erorii

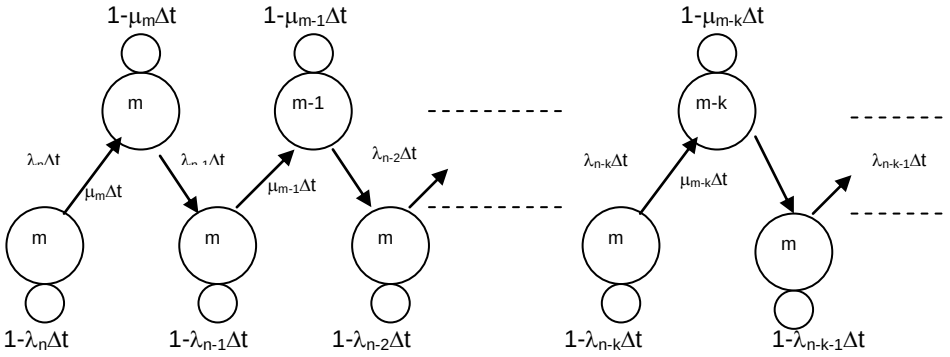

Fig.10.5. Graful de tranziţie în cazul modelului V

MODELUL VI

Se fac urmatoarele ipoteze:

1. Erorile software sunt independente de timpul de operare al programului în cauză: dacă programul este bine testat în perioada de rodaj, nu vor apare erori de exploatare;

2. Atunci când mulţimea datelor de intrare în exploatarea programului nu coincid cu cele din perioada de rodaj, există o anumită probabilitate F să apară erori în cursul rulării programului.

Pornind de la aceste ipoteze, se poate determina probabilitatea de eroare a unui program în funcţie de formele posibile ale acestuia:

$$F = \sum_{i=1}^{N_1} P_1 b_1$$

N_1 - reprezintă numărul de forme posibile ale aceluiaşi program în funcţie de datele de intrare care, prin analogie cu terminologia utilizată în cazul sistemelor materiale, se vor numi număr de intrări ale unui program;

P_i - probabilitatea de apariţie a intrării i;

b_i - o funcţie binară care ia valoarea “1” daca programul i este eronat şi “0” dacă programul este corect.

Dacă formele posibile ale programului au loc cu aceeaşi probabilitate $P_i=1/N$, atunci probabilitatea de eroare a programului devine:

$$F = \frac{\sum_{i=1}^{N_1} b_i}{N_i}$$

Dacă programul este testat pentru n intrări diferite şi apar erori, probabilitatea ca programul să fie eronat se va putea exprima ca:

$$F_{k+1} = F_k - \frac{1}{N_1}$$

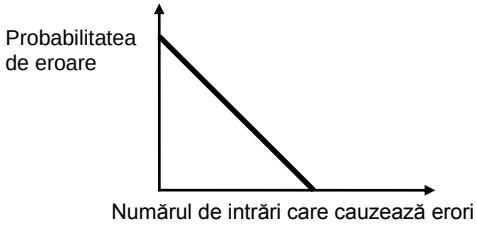

Fig.10.6. Probabilitatea de eroare

În ipoteza ca o eroare odată detectată nu mai apare în aceleaşi condiţii, se poate exprima probabilitatea de eroare a unui program după rodaj ca fiind:

$$F_{k+1} = F_k - \frac{1}{N_1}$$

unde s-au făcut notaţiile:

F_k - probabilitatea de eroare a programului înainte de rodaj;

F_{k+1} - probabilitatea de eroare a programului după rodaj;

N_1 - intrările care conduc la un program eronat;

I - numărul de intrări ale sistemului logic (program) care cauzează erori în timpul testarilor din perioada de rodaj F. F_{k+1} şi F_k pot fi calculate pe baza relaţiei F=e/N, iar N_1 se estimează ca fiind panta dreptei din Fig.10.6, dreaptă care poate fi ridicată experimental în perioada de rodaj.

10.6. Metode software pentru reducerea erorilor

Ceea ce este unanim acceptat este faptul că trebuie focalizate eforturile către depistarea şi îndepărtarea defectelor care, în software se numesc de regulă erori.

Un defect (o eroare) este orice lipsă din specificaţiile de proiectare sau implementare a procesului (Grady, 1992).

Ingineria se străduieşte nu numai să creeze şi să îmbunătăţească noi produse, ci să le producă cu costuri eficiente.

Un cost major în software este stabilirea, depistarea defectului. Multe studii arată că preţul pentru găsirea şi stabilirea "defectelor" software pot varia dramatic în funcţie de timpul scurs până la găsirea lor (Grady, 1992).

Ţinând cont de faptul că există costuri relative mai mari decât 100 la 1, un beneficiu major al practicilor inginereşti este de a reduce costul şi varietatea reluărilor.

Cele mai multe evaluări ale calităţii software se fac de-a lungul activităţilor de testare.

Îngrijorarea managerilor se manifestă în general în acest stadiu şi în consecinţă, unele dintre produsele finale care suportă îmbunătăţiri în urma testării de-a lungul dezvoltării au suportat de fapt primele îmbunătăţiri ale calităţii.

Criteriile utilizate pentru a judeca când s-a încheiat testarea sistemelor software sunt foarte largi şi adesea subiective.

Din cauza unei variaţii largi în ceea ce priveşte luarea în considerare a acestora în aprecierea calităţii software (permisă de o lipsă de standardizare), firma Hewlett- Packard (HP) a creat un set de criterii de măsurare a calităţii.

Propunerea lor este de a certifica produsele software realizate şi gata de livrare (Grady, 1992).

Cerinţele pentru a realiza acest obiectiv au rezultat din următoarele considerente:

⇒ Furnizează o măsură de proces consistentă pentru produsul testat;

⇒ Furnizează "piese" cuantificabile pentru a evalua progresul până la realizarea produsului;

⇒ Permite funcţionalitatea pe faze;

⇒ Încurajează automatizarea unui număr cât mai mare de teste;

⇒ Măreşte fiabilitatea produsului care funcţionează la utilizatori.

Aceste criterii au condus la un set echilibrat de teste şi mărimi de bază măsurabile, în raport cu care să poată fi judecată calitatea.

Standardele de testare includ cerinţele următoare:

- **întindere**: extinderea testării atât la ceea ce este accesibil utilizatorului, cât şi peste funcţiile interne;
- **adâncime**: testare acoperitoare a ramificaţiilor;
- **fiabilitate**: operare continuă sub "stress";
- **stabilitate**: abilitate de acoperire a condiţiilor pentru producerea erorilor;
- **densitatea** defectelor remanente la livrare.

Testarea sistemelor mari în standard HP include multiple cicluri de test.

Se utilizează de obicei trei stadii pentru completare şi stabilitate.

Fiecare din ele au cerinţe ridicate de întindere, adâncime, operare continuă şi densitate de defect.

Aceste cerinţe permit integrarea unui mare număr de componente cu nivel de calitate cunoscut.

Un prim indicator pentru management a fost rata defectelor care apar în sistem.

Fig. 10.7. prezintă 3 curbe a defectelor care apar (în cazul cererilor de service) (Grady, 1992).

Numărul defectelor în fiecare caz este normalizat de cantitatea de cod, în mii de linii sursă fără comentarii (KNCSS).

Vârful curbei reprezintă mai multe produse care ori n-au fost certificate ori au fost realizate fără să li se aplice criteriile de certificare.

Minimul curbei este media a 12 produse certificate, iar linia medie arată că rata defectelor, chiar a celor mai proaste produse certificate, a fost mult mai bună decât a produselor care nu au fost executate după standarde.

Acest grafic sugerează că un bun proces de testare contribuie la o rată mai scăzută şi mai stabilă a erorilor care apar (Fig.10.7).

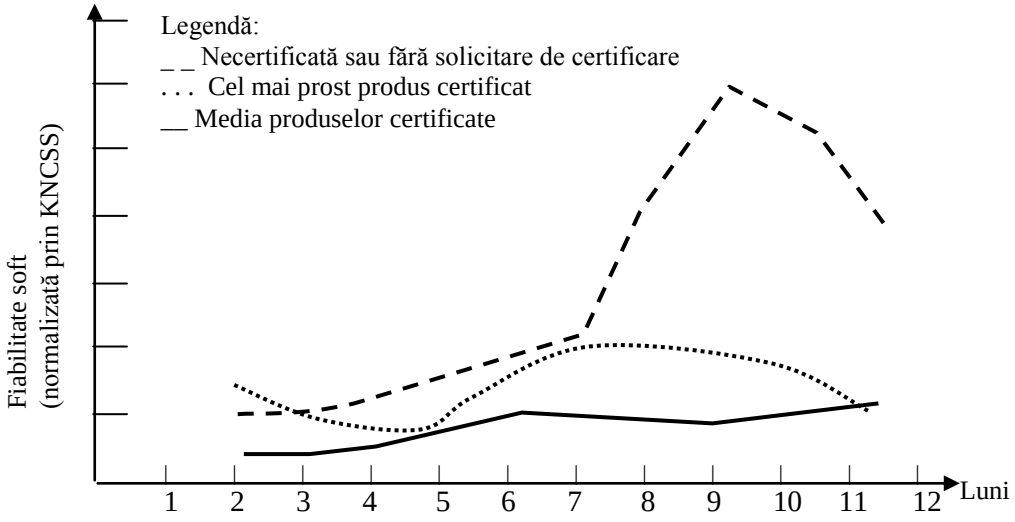

Fig.10.7. Spectrul tehnic al unui program

10.6.1. Inspecţii, acoperire cu cod şi complexitate

Procesul de testare implică de regulă numeroşi ingineri, care trebuie să consulte, să semnaleze şi corecteze sistemul.

În contrast cu aceasta, inspecţiile pot fi startate chiar şi de un singur inginer şi, mai mult, există rezultate mai bine documentate pentru acest gen de activitate decât pentru orice altă practică din ingineria software.

Deşi inspecţiile sunt în primul rând tehnici de detecţie a erorii, experienţele în domeniu au arătat că şi activităţile de pregătire a inspecţiei au drept rezultat prevenirea erorii.

În cazurile în care "producţia" de software nu se face într-o forma standardizată, pregătirea pentru o inspecţie forţează o clarificare a cerinţelor.

Inspecţiile pot ajuta la "a acoperi" multe defecte înainte ca ele să devină foarte costisitoare sau înainte ca ele să fie foarte integrate în sistem, ceea ce le-ar face dificil de înlăturat.

Ca o remarcă la fel de importantă, **inspecţia impune disciplină**.

Trebuie create produse inspectabile la intervale regulate de timp, ceea ce va avea drept rezultat obţinerea diferitelor imagini ale părţilor produsului final.

Produsele au din proiect, de regulă, mai multe părţi, iar inspectarea acestora va produce un feedback măsurabil şi obiectiv.

În final, inspecţiile vor constitui o cale de educare a echipei de software şi de încurajare a celor mai bune practici, acestea fiind aspectele unei bune inginerii software.

O modalitate importantă de a păstra inspecţiile eficiente este de a produce rezultate măsurabile.

Obiectul unei bune inspecţii este de a găsi, mai repede decât pe alte căi, unde se afla hibele.

Se poate măsura procentajul de defecte găsite şi timpul în care au fost depistate.

Adesea inginerii software cred că au făcut toate testările posibile, când de fapt nu este aşa.

Reflectarea erorilor în codificare este măsurată prin utilizarea unui instrument care inserează instrucţiuni în codul sursă precompilat.

Când se rulează testul, "extracodul" marchează segmentele care au fost executate şi care nu.

10.7. Utilizarea datelor eronate pentru îmbunătăţirea deciziilor

La sfârşitul anului 1986 Consiliul de Metrici Software HP a făcut cunoscute definiţiile categoriilor standard de cauze ale defectelor.

Fig. 10.10 reprezintă modelul definiţiilor prezentate în lucrarea lui Grady (1992).

Modelul este utilizat în scopul selectării unui descriptor pentru origini, tipuri şi moduri, pentru fiecare raport de defectare care este rezolvat.

De exemplu, un defect ar putea fi o eroare de proiectare care face parte din interfaţa cu utilizatorul dar care lipseşte ca descriere din specificaţiile interne.

Un alt defect ar putea fi o eroare de codificare, atunci când ceea ce este logic, este eronat.

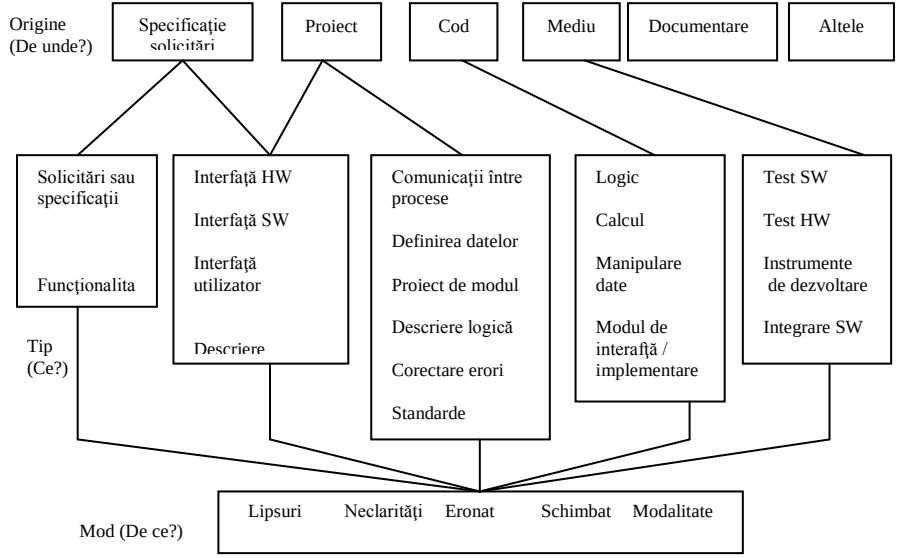

Fig.10.10. Categoriile surselor de defecte software

Se pare că există mai multe atribute care ajută în obţinerea succesului în ceea ce priveşte calitatea software şi anume:

- O diagramă conceptuală simplă, cum este cea din fig. 10.10 ajută evaluatorii să înteleagă ce este cel mai simplu de făcut, şi îi ghidează în vederea proporţionării cantităţii de efort necesare raportărilor;

- Definiţiile standard pentru tipurile de defecte ajută la rezolvarea pe diferite căi de raţionament a problemelor similare;

- Furnizând un cadru pentru raportarea analizelor de erori care au fost rezolvate se încurajează interesul altor grupuri în a întreprinde astfel de acţiuni;

- Pregătirea în colectarea datelor despre erori şi analizarea lor este de asemenea,foarte valoroasă.

10.8. Reducerea erorilor datorită măsurătorilor

Cunoscând care erori apar cel mai frecvent în testare sau mai târziu, se pot concentra eforturile de îmbunătăţire a acestei situaţii.

Echipele HP, de care s-a mai amintit deja, au cules date pentru specificaţii, proiect şi inspecţiile de cod. Toate acestea sunt prezentate în Fig. 10.12.

Trebuie manifestată puţină precauţie în interpretarea acestor date specifice, atât timp cât ele n-au fost colectate în mod uniform.

Linia orizontală din mijlocul figurii indică valorile pentru defectele diferite care au fost găsite în aceeaşi fază.

Barele verticale reprezintă ocaziile favorabile de a reduce, semnificativ, aceste surse de defecte.

Barele de sub linie arată valorile pentru defectele găsite în fazele imediat următoare în care au fost create.

Barele verticale sunt aici, surse, atât pentru o prevenire mai bună, cât şi ocazii de detectare mai precoce a erorilor.

De exemplu, solicitările, funcţionalitatea şi descrierea funcţională a defectelor se combină în a sugera că proiectele trebuie schimbate datorită unei definiri (specificaţii) iniţiale neadecvate a produsului. În asemenea situaţii ar fi util de folosit prototipurile.

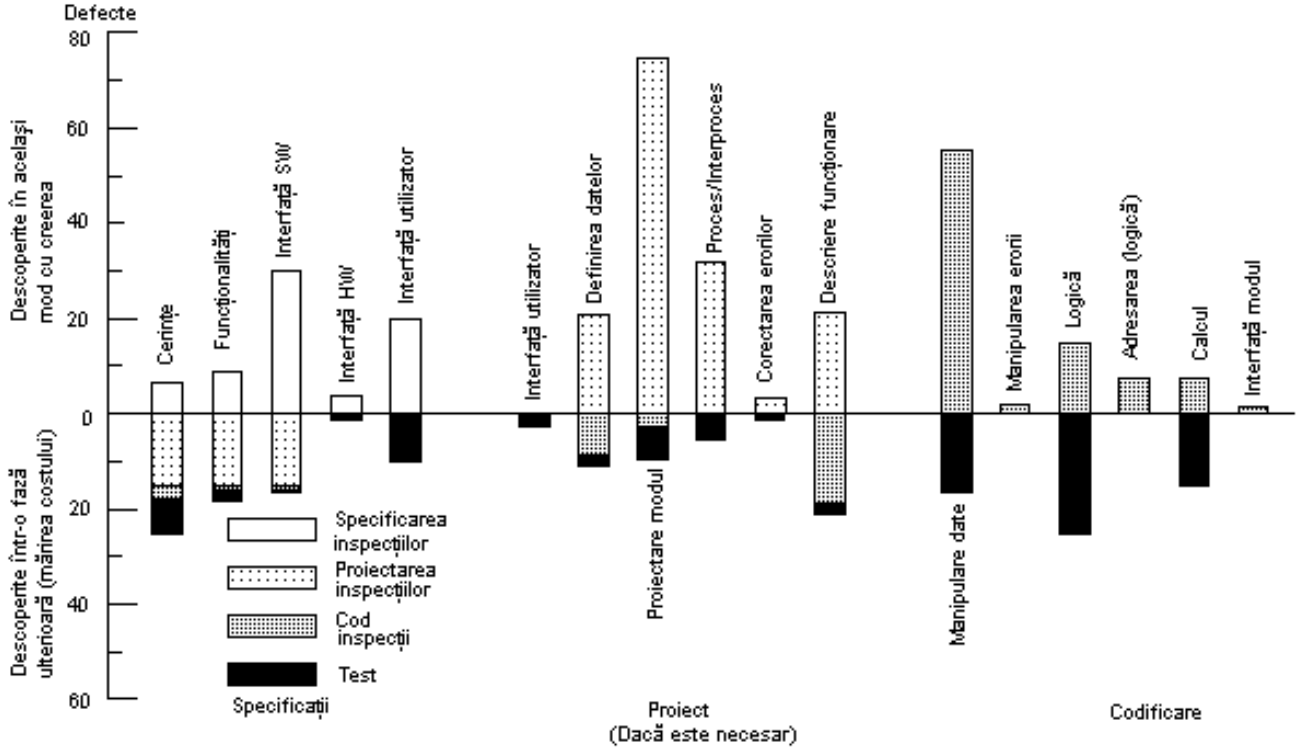

Fig.10.12. Profilul erorilor

Aceste tipuri de date creează posibilitatea de a lua decizii mai bine informate şi indică o cale de evaluare a rezultatelor schimbărilor cu o precizie mai bună decât în trecut.

Introducând noi proceduri standard pentru corectarea erorilor în cazul a trei produse realizate s-au obţinut rezultatele prezentate în fig. 10.13.

S-a început cu realizarea produsului în care erau 58 de erori, apoi s-a validat ceea ce era proiectat şi efectiv s-a înlocuit cea mai mare parte a erorilor rezultând cele două produse B şi C (Grady, 1992).

În concluzie, o bună calitate software este rezultatul unei bune inginerii software.

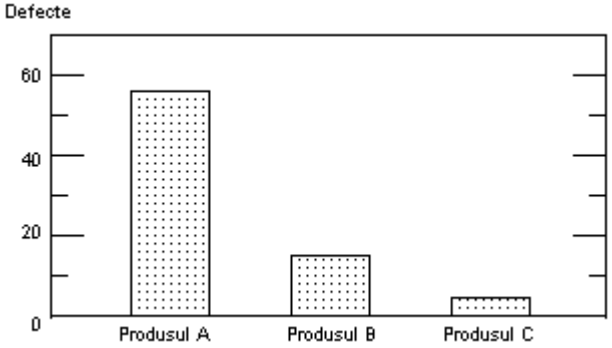

Fig.10.13. Erori de manevrare

Toate organizaţiile trebuie îndrumate să adopte practici de inginerii software atât pentru considerente de afaceri, cât şi pentru cerinţe profesionale, aceasta fiind singura formulă de obţinere a unui produs software cât mai aproape de necesităţile utilizatorilor.

10.9. Aplicarea analizei cauzale procesului de modificare a software-ului

Producerea sistemelor software de înaltă calitate pe scară largă în cadrul unor restricţii de proiect şi de buget a devenit o competiţie în ingineria software.

Modificarea acestor sisteme pentru a încorpora noul şi posibilităţile de schimbare supun tot timpul sistemul la o competiţie din ce în ce mai mare.

Această activitate de modificare trebuie să fie executată fără a afecta calitatea existentă a sistemului.

Din păcate, acest obiectiv este rareori atins, modificările software introducând adesea efecte laterale nedorite şi conducând la reducerea calităţii.

O abordare tradiţională în dezvoltarea produsului software de înaltă calitate constă în aplicarea unei metodologii de dezvoltare, cu sublinieri accentuate pe detecţia erorilor.

Acest proces de detectare a erorilor constă într-o căutare continuă, prin inspectare şi testare la diverse niveluri.

O abordare mai eficace pentru dezvoltarea unui produs de înaltă calitate este o accentuare pe prevenirea erorilor, care se poate face prin aplicarea analizei cauzale.

Analiza cauzală constă în colectarea şi analiza datelor de defect software în ordinea identificării cauzelor lor.

Din momentul în care cauzele sunt identificate, pot fi aduse îmbunătăţiri proiectului pentru a preveni apariţiile viitoare ale erorilor.

O lucrare de specialitate de la firma IBM prezintă o metodologie de abordare a proiectului care utilizează analiza cauzală şi feedback-ul ca mijloace pentru obţinerea îmbunătăţirii calităţii şi, în final, prevenirea defecţiunilor.

Metodologia de prevenire a defectelor se bazează pe următoarele trei concepte:

1) Proiectanţii şi-au evaluat propriile greşeli;

2) Analiza cauzală este parte din procesul de dezvoltare software;

3) Feedback-ul este parte din proces.

O privire generală a procesului de analiză cauzală propusă de Jones este ilustrată în Fig.10.14

Prima activitate constă din începerea activităţii echipei de analiză cu următoarele obiective:

a) Revederea datelor eronate de intrare;

b) Revederea liniilor directoare din metodologie;

c) Revederea listelor de verificare potrivite;

d) Identificarea cerinţelor echipei.

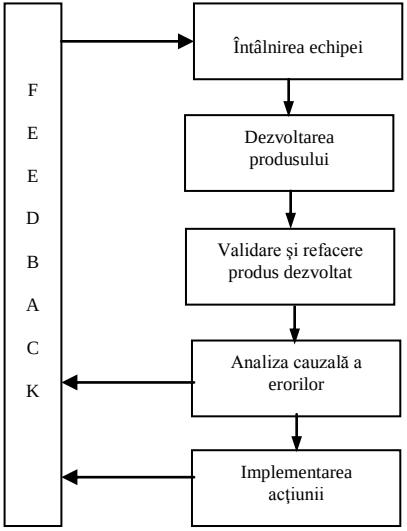

Fig.10.14. Procesul de analiză cauzală

Următoarea activitate este dezvoltarea produsului care apare, folosind feedback-ul obţinut de la activitatea echipei întrunită în scopul prevenirii creării de noi erori.

Produsele rezultate sunt apoi validate şi refăcute acolo unde este necesar.

Activitatea de analiză cauzală este apoi efectuată, începând cu analiza erorilor, făcută de autorii erorilor. Aceasta implică analizarea fiecărei erori pentru a determina:

1) tipul erorii sau categoria;
2) faza în care a fost găsită;
3) faza în care a fost creată;
4) cauza (cauzele) erorii;
5) soluţia (soluţiile) pentru prevenirea apariţiei erorii în viitor.

Această informaţie este înregistrată într-o anumită formă în analiza cauzală şi folosită apoi ca dată de intrare într-o bază de date.

O altă echipă de analiză cauzală se întâlneşte apoi pentru analizarea datelor din baza de date.

În plus, grupul poate avea nevoie la un moment dat să se consulte cu alte persoane din afara echipei (proiectanţi, persoane care au efectuat testele, etc.), pentru a completa analiza.

Echipa de analiză cauzală este responsabilă pentru identificarea ariilor majore de probleme, privind datele eronate ca un întreg, în loc de analizarea unei erori particulare la un moment dat.

Echipa foloseşte metoda de rezolvare a problemelor pentru: a analiza datele, a determina punctele asupra cărora trebuie să lucreze, cauzele grupurilor de probleme şi pentru a dezvolta

planuri de implementare şi recomandări care să prevină apariţia tipului sau tipurilor similare de probleme în viitor.

Aceste recomandări sunt apoi supuse unei acţiuni în echipă, care are următoarele responsabilităţi:

a) Evaluarea şi prioritizarea recomandărilor;

b) Implementarea recomandărilor;

c) Răspândirea feedback-ului.

Echipa de acţiune trebuie să se întâlnească periodic (de exemplu, o dată pe lună) pentru a revedea orice nou plan de implementare recepţionat de la echipa de analiză cauzală şi să verifice stadiul planurilor de implementare prevăzute, precum şi numărul acţiunilor.

Starea acţiunii este de asemenea păstrată în baza de date a analizei cauzale şi monitorizată de acţiunile echipei.

Cele mai multe eforturi raportate la activităţile de analiză cauzală au fost focalizate pe dezvoltarea procesului.

Aceasta reprezintă, din păcate, un procentaj ridicat de efort consumat în timpul activităţilor de modificare software. Modificările software apar pe măsura adăugării de noi posibilităţi sau pe măsura

modificării celor existente ale sistemului. Modificarea în linii mari a unui sistem complex este o mare consumatoare de timp şi o activitate succeptibilă la erori. Această sarcină devine şi mai dificilă în timp, pe măsură ce sistemul se măreşte şi structurile sale se deteriorează.

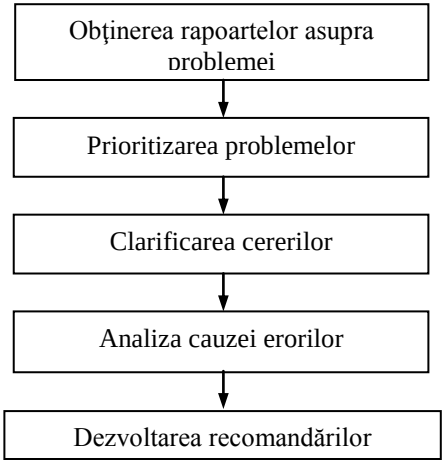

Fig.10.15. Analiza cauzală în procesul de modificare software

Analiza cauzală

Paşii analizei cauzale sunt prezentaţi sumar în Fig.10.15.

Pasul 1. Obţinerea rapoartelor asupra problemei

Prima activitate care trebuie efectuată este colectarea rapoartelor asupra problemei care rezultă din procesele de modificare ce au suferit scrutinul analizei cauzale. Aceste rapoarte pot fi generate intern prin testare, sau extern, de la un client.

Pasul 2. Prioritizarea problemelor

Prioritizarea erorilor permite o analiză a cauzelor care contribuie la creşterea priorităţii problemei şi constă în împărţirea erorilor în trei categorii, şi anume:

1) Critice;

2) Majore;

3) Minore.

Erorile critice slăbesc funcţionalitatea şi interzic testarea viitoare. Erorile majore slăbesc parţial funcţionarea, iar erorile minore nu afectează în mod normal operarea sistemului.

Pasul 3. Clasificarea erorilor

După prioritizarea erorilor urmează clasificarea lor. Primul obiectiv al acestei clasificări este facilitatea care se creează astfel pentru a lega erorile de cauzele lor. De-a lungul anilor au fost propuse numeroase clasificări ale erorilor. Deşi clasificarea nu este neapărat necesară în efectuarea unei analize cauzale, datorită faptului că sistemele bazate pe cunoştinţe nu sunt produse software obişnuite, nu este lipsit de interes o astfel de încercare, care să adauge la categoriile de erori din software-ul tradiţional, clase noi, specifice sistemelor de inteligenţă artificială (Novac 1994).

Clasificarea erorilor furnizează de asemenea informaţii utile atunci când se determină eficacitatea din punct de vedere al costului eliminării erorilor care ar putea să apară.

Categoriile de erori sunt :

A) Erori de proiectare;

B) Erori provenite din validarea cunoaşterii;

C) Erori de interfaţă incompatibilă;

D) Sincronizare incorectă dintre proiectele paralele;

E) Resturi corectate de obiecte incorecte;

F) Epuizarea resurselor sistemului.

A) Erori de proiectare

Această categorie reflectă erorile software cauzate de o transpunere improprie a cerinţelor în proiect, de-a lungul perioadei de modificare.

Sunt incluse proiectul la toate nivelurile, achiziţia şi structurarea cunoaşterii, realizarea prototipului. Exemple tipice de erori de proiectare sunt:

A1) ***Erori logice***: Condiţii eronate logic în reguli, transpunere logică greşită a cunoaşterii în reguli, etc.

A2) ***Erorile de calcul*** sunt legate de inacurateţea şi greşelile făcute în implementare legate de operaţia de calcul, în special în cazul în care se folosesc cunoştinţe care rezultă din algoritmi de calcul, cum ar fi algoritmii precompilaţi care studiază rezistenţa navei în cazul unui sistemului expert.

A3) *Lipsa excepţiei de manipulare*. Aceste erori includ greşeala de a lucra cu condiţii unice sau cazuri unice chiar şi atunci când există excepţii şi acestea au fost prevăzute în specificaţiile proiectului.

A4) *Erori de timp*. Acest tip de erori reflectă proiectarea incorectă a timpului critic software. De exemplu, sistemul încearcă să facă o căutare "forward" cu multe inferenţe atunci când răspunsul sistemului trebuie să fie foarte rapid, luând în considerare numai cunoştinţe de suprafaţă.

A5) *Erori de manipulare a datelor*. Aceste erori includ greşelile făcute la iniţializarea datelor înainte de utilizare, utilizări improprii ale constantelor din sistem, control neadecvat al fişierelor de date adiţionale sistemului, etc.

A6) *Erorile în conceptele I/O* includ forme de intrare sau ieşire în/din fişiere eronate, erori de formatare, definirea unor înregistrări de dimensiune greşită, protocoale de I/O eronate sau improprii dispozitivelor sistemului.

A7) *Definirea greşită a datelor*. Această categorie reflectă proiectarea incorectă a structurilor de date care vor fi utilizate în diferitele module din baza de cunoştinţe, sau incorecta definire a structurilor globale. Se reflectă de asemenea aici abstractizarea incorectă a datelor.

B). Erori provenite din validarea cunoaşterii

Această categorie, specifică sistemelor bazate pe cunoştinţe, se referă la erorile care apar în oricare dintre fazele de prelucrare a cunoaşterii, începând cu achiziţia primară a cunoştinţelor şi continuând cu mentenanţa sistemelor (vezi 8.4 Validarea cunoaşterii).

C). Interfaţă incompatibilă

Erorile legate de interfaţă apar atunci când interfeţele dintre două sau mai multe tipuri diferite de componente modificate nu sunt compatibile. Se pot da următoarele exemple de acest gen:

1) Se transmit între diversele module parametri diferiţi, faţă de cei care sunt aşteptaţi;

2) Se efectuează alte operaţii de către modulele apelate decât cele care s-au gândit în proiect. Aceste situaţii pot să apară şi din cauza propagării erorilor de la modulele eronate;

3) Dezacordul dintre baze de date şi cod atunci când baza a fost proiectată;

4) Rutine de execuţie sau alte rutine, inexistente;

5) Dezacord între software şi hardware;

6) Dezacord între software şi firmware (microprogramare) atunci când anumiţi parametri trebuie obţinuţi prin microprogramare.

D) Sincronizarea incorectă dintre proiectele paralele

Pentru o evoluţie mai largă a unui sistem, ciclurile de dezvoltare software ale mai multor realizări se pot suprapune aşa cum se prezintă în fig. 10.16.

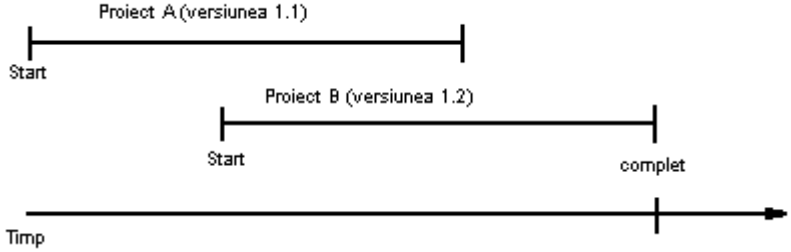

Fig.10.16. Evoluţia sistemului software

Această categorie de erori apare atunci când schimbările, modificările din proiectul anterior A, sunt incorect continuate în proiectul B, care este proiectul curent.

E). "Resturi" corectate de obiecte incorecte

Într-un efort mai mare de mentenanţă resturile de obiecte sunt câteodată folositoare. Aceste resturi pot rezulta din mai multe revizii ale sistemului. Erorile comise atunci când se modifică modulele cu resturi (părţi) de obiecte fac parte din această categorie.

F). Epuizarea resurselor sistemului

Acest tip de erori apare atunci când resursele sistemului, cum ar fi memoria şi timpul real devin insuficiente. Exemple de acest tip sunt: epuizarea spaţiului în stiva dinamică de memorie (heap), a unor registre, sau a ceasului de timp real.

Pasul 4. Analiza şi determinarea cauzelor erorilor

Cel mai critic pas din realizarea analizei cauzale este determinarea cauzei fiecărei erori. Această sarcină este cel mai bine realizată de programatorul care a introdus eroarea. Pentru a facilita identificarea cauzei erorii, proiectanţii software care au introdus erorile trebuie intervievaţi, dar într-o manieră neoficială şi cu foarte mare grijă pentru a nu provoca reacţia lor de apărare. Analiza pentru prevenirea defectului trebuie subliniată clar ca fiind scop şi prioritate în acest interviu. Bazat pe informaţia obţinută de la proiectantul care a introdus eroarea se poate selecta o categorie cauzală de erori.

Considerăm că se pot identifica şapte categorii majore de cauze care conduc la erori în modificarea software-ului. Acestea diferă de altele prezentate în literatura de specialitate şi care sunt orientate direct către descoperirea cauzelor erorilor comise de-a lungul dezvoltării unui sistem

software nou. Aceste scheme de clasificare a erorilor nu se adresează în exclusivitate numai cauzei care a produs erorile apărute de-a lungul procesului de mentenanţă.

Cerinţa de determinare a categoriei cauzale pentru fiecare eroare este de a colecta datele necesare pentru stabilirea categoriei cauzale căreia îi corespunde cel mai bine eroarea. Acest lucru furnizează o metodă pentru evaluarea eficienţei costului implementării recomandărilor care elimină sau reduc cauzele erorii.

Categoriile cauzale obişnuite în activităţile de modificare sunt:

I) Cunoaşterea sistemului / experienţa

Această categorie cauzală reflectă lipsa cunoaşterii proiectului software al produsului sau a procesului de modificare. Se pot da câteva exemple:

1) Înţelegerea greşită (confuziile) a proiectului existent;

2) Înţelegerea greşită a procesului de modificare;

3) Înţelegerea neadecvată a mediului de programare. Mediul de programare incluzând personal, maşini, management, instrumente de dezvoltare şi alţi factori care afectează posibilitatea de dezvoltare şi modificare a sistemului;

4) Înţelegerea neadecvată a solicitărilor clientului.

II) Comunicaţia

Această categorie cauzală reflectă problemele de comunicare în ceea ce priveşte modificările care trebuie efectuate. Se identifică cauzele care nu au fost atribuite lipsei cunoaşterii sau experienţei şi care apar datorită comunicării incorecte sau incomplete.

Problemele de comunicare sunt de asemenea cauzate de confuzia în rândurile membrilor echipei în ceea ce priveşte responsabilitatea sau deciziile. Câteva exemple de probleme de erori în comunicare sunt:

1. Greşeala unui grup de proiectare datorită necomunicării ultimei modificări;

2. Eroare în scrierea unei rutine de tratare a erorilor deoarece fiecare membru al echipei a considerat că ea este scrisă de altul.

III) Impacturile software

Această categorie reflectă eroarea proiectantului software în considerarea tuturor implicaţiilor posibile ale modificării software-ului. Un exemplu este omiterea unei codificări de acoperire a erorii după adăugarea unei noi piese harware.

IV) Metode / standarde

Această categorie reflectă încălcările de metode şi/sau standarde în module, de asemenea, limitări ale metodelor existente sau ale standardelor care contribue la defectări. Un exemplu ar fi omiterea revizuirii codificării, înainte de testare.

V) Caracteristica desfăşurarii

Această categorie reflectă problemele legate de inabilitatea software-ului, hardware-ului, componentelor bazei de date, de a se integra la un moment dat. Este caracterizată de o lipsă a planurilor de prevenire care să evite problemele cauzate de lipsa componentelor.

VI) Instrumente de susţinere

În această categorie se regăsesc problemele legate de instrumentele care introduc erori. Un exemplu îl constituie un instrument de validare a cunoaşterii care se utilizează pentru evaluarea bazelor de cunoştinţe şi care introduce erori în cunoştinţe.

VII) Eroarea umană

Această categorie cauzală reflectă erorile umane comise de-a lungul procesului de modificare care nu sunt atribuibile altor surse. Proiectantul a ştiut ce face şi a înţeles totul de la un cap la altul, dar a greşit pur şi simplu.