Conținut:

Bibliografie:

1. Cornelia Novac Ududec, Ingineria sistemelor de programe - *Ingineria programării, Ediție adăugită şi revizuită*, Editura Alma Mater, Bacău, 2011;
2. Rotar Dan, Ingineria programelor, Editura Alma Mater, Bacău, 2007
3. Pilat Florin, s.a., Metode, tehnici şi instrumente în ingineria programării, EdituraTehnică, Bucureşti 1985
4. Vaduva Ilie, Baltac Vasile, Florescu Vasile, s.a., Ingineria programării. Vol I, II – Editura Academiei, Bucureşti, 1986

1. Sistemele informatice. Probleme şi perspective

1.1 Introducere

Următoarele exemple oferă o imagine mai completă a complexității la care a ajuns software-ul în zilele noastre:

- Sistemul de rezervare a biletelor pentru compania aeriană KLM conținea, în anul 1992, 2.000.000 de linii de cod în limbaj de asamblare;
- Sistemul de operare System V versiunea 4.0 (UNIX) a fost obținut prin compilarea a 3.700.000 linii de cod;
- Sistemele de programe pentru controlul navetelor spațiale NASA au circa 40 de milioane de linii de cod;
- Pentru realizarea sistemului de operare IBM OS360 au fost necesari 5000 de ani-muncă.om.

Se face o paralelă între ingineria software şi ingineria construcțiilor:

- cuşcă pentru câine, putem să mergem în grădină, să căutam lemne şi cuie, să luăm un ciocan şi să începem să lucrăm. Avem şanse destul de bune să reuşim, mai ales dacă suntem

îndemânatici. Dacă totuşi nu ne iese, putem încerca a doua zi din nou cu alte lemne şi alte cuie. Şi totuşi, dacă câinele nu încape în cuşcă, putem să ne cumpărăm alt câine.

- când dorim să construim o casă:
  - angajăm un arhitect care să ne facă un proiect, sau să cumpărăm un proiect standard de casă.
  - negociem cu o firmă de construcții: prețul, durata de realizare, calitatea finisajelor.
  - Nu ne permitem să riscăm economiile familiei pe o construcție care se va dărâma la o rafală de vânt. În plus, dacă membrilor familiei nu le place orientarea ferestrelor sau peisajul, nu îi putem schimba cu alții (în cel mai rău caz, ne schimbă ei pe noi).

Inginerii constructori întocmesc:

- planuri,
- construiesc machete,
- studiază proprietățile materialelor folosite
- fac rapoarte privind progresul operațiunilor.

"Constructorii" de aplicații software trebuie să procedeze la fel pentru ca dezvoltarea programelor să nu mai fie un proces impredictibil.

Un raport prezentat de către o companie de software, în care erau analizate diverse proiecte şi stadiile lor de finalizare, a constatat că:

- 2% din sistemele software contractate au funcționat de la predare;
- 3% din sistemele software au putut funcționa după câteva modificări;
- 29% au fost predate dar n-au funcționat niciodată;
- 19% au fost folosite dar au fost abandonate;
- 47% au fost plătite dar niciodată predate.

Prima definiție dată ingineriei software a fost formulată astfel (F. L. Bauer):

Ingineria software este stabilirea şi utilizarea de principii inginereşti solide pentru a obține în mod economic programe sigure şi care funcționează eficient pe maşini de calcul reale.

În IEEE Standard Glossary of Software Engineering Technology (1983) ingineria software este definită după cum urmează:

Ingineria software, ingineria sistemelor de programe (ingineria programării), reprezintă abordarea sistematică a dezvoltării, funcționării, întreținerii, și retragerii din funcțiune a programelor.

A doua definiție adaugă însă referiri la perioade importante din viața unui program, ce urmează creării şi funcționării sale, şi anume întreținerea şi retragerea din funcțiune.

Se consideră că ingineria software are următoarele caracteristici importante:

- este aplicabilă în producerea de sisteme de programe mari;
- este o ştiință inginerească;
- scopul final este îndeplinirea cerințelor clientului.

Statitstici:

- în SUA se cheltuiesc anual 250 de miliarde de dolari pentru producția de software, dar 33% dintre proiectele informatice eşuează, adică 80 de miliarde de dolari se pierd.
- 83% dintre proiectele informatice au probleme.

- că proiectarea şi realizarea aplicațiilor informatice nu pot fi făcute oricum ci trebuie să respecte normele şi metodologiile standardizate, în primul rând PMI (Project Managemant Institute) şi apoi pe cele specifice aplicațiilor software.

Un program este ***fiabil*** dacă funcționează si continuă să funcționeze fără întreruperi şi erori un interval de timp. = rezistența la condițiile de funcționare.

Un sistem de operare trebuie să fie fiabil pentru că este obligatoriu să funcționeze o perioadă suficient de lungă de timp fără să clacheze, indiferent de programele care rulează pe el, chiar dacă nu totdeauna la performanțe optime.

Programul este ***sigur*** dacă funcționează corect, fără operații nedorite. Un program pentru un automat bancar trebuie să fie sigur, pentru a efectua tranzacțiile în mod absolut corect, chiar dacă funcționarea sa poate fi întreruptă din când în când. Atunci când funcționează însă, trebuie să funcționeze foarte bine.

Un program are o ***eroare*** dacă nu se comportă corect. Se presupune că dezvoltatorul ştia ce ar fi trebuit să execute programul, iar comportamentul greşit nu este intenționat.

Ingineria software are ca scop obținerea de sisteme funcționale chiar şi atunci când teoriile şi instrumentele disponibile nu oferă răspuns la toate provocările ce apar. Inginerii fac ca lucrurile să meargă, ținând seama de restricțiile organizației în care lucrează şi de constrângerile financiare.

Problema fundamentală a ingineriei software este ***satisfacerea cerințelor utilizatorului***. Aceasta trebuie realizată nu punctual, nu imediat, ci într-un mod flexibil şi pe termen lung.

Ingineria software se ocupă de toate etapele dezvoltării programelor, de la achiziția cerințelor de la client până la întreținerea şi retragerea din folosință a produsului livrat. Pe lângă cerințele funcționale, clientul doreşte (de obicei) ca produsul final să fie realizat cu costuri de producție cât mai mici. De asemenea, este de dorit ca aceasta să aibă performanțe cât mai bune (uneori direct evaluabile), un cost de întreținere cât mai mic, să fie livrat la timp, şi să fie sigur.

Rezumând, atributele cheie ale unui produs software se referă la:

- ***Mentenabilitate****,* posibilitatea de a putea fi *întreținut*: Un produs cu un ciclu de viață lung este supus deseori modificărilor, de aceea el trebuie foarte bine documentat;
- ***Fiabilitate***: produsul trebuie să se comporte după cerințele utilizatorului şi să nu „cadă” mai mult decât e prevăzut în specificațiile sale;

- ***Eficiență***: produsul nu trebuie să folosească în pierdere resursele sistemului ca memoria sau timpul de procesare;
- ***Interfața*** potrivită pentru utilizator: interfața trebuie să țină seama de capacitatea şi cunoştințele utilizatorului.

Optimizarea tuturor acestor atribute e dificilă deoarece unele se exclud pe altele (de exemplu, o mai bună interfață pentru utilizator poate micşora eficiența produsului).

În cazurile în care eficiența este critică, acest lucru trebuie specificat explicit încă din faza de preluare a cerințelor utilizatorului, precum şi compromisurile pe care ea le implică privind ceilalți factori.

Trebuie spus că ingineria software nu rezolvă toate problemele care apar atunci când se scriu programe dar, în momentul de față, ea ne poate spune sigur ce să *nu* facem.

1.2 Probleme ale software-ului

Una dintre cele mai întâlnite probleme ale sistemelor software este "proiectarea greşită" (în engl. ”bad design”). Pentru a rezolva problema unui design greşit trebuie mai întâi definită această problemă şi analizate cauzele ei.

Definiția unei ”proiectări greşite”

O piesă software care satisface cerințele impuse, legate de funcționalitate, are un "design greşit", dacă îndeplineşte cel puțin una din condițiile de mai jos:

1. Este dificil de modificat, pentru că orice modificare implică modificări în multe alte părți ale sistemului, această caracteristică numindu-se **rigiditate;**
2. Dacă se face o modificare, alte părți ale sistemului nu mai funcționează; această caracteristică se numeşte **fragilitate**;
3. Este dificil de refolosit în altă aplicație pentru că nu poate fi detaşată de aplicația curentă. Această caracteristică se numeşte **imobilitate**.

Cauze ale unui ”design greşit"

Una din cauzele **rigidității**, **fragilității** şi **imobilității** unei proiectări este **interdependența** modulelor implicate în acel design.

Un design este **rigid** dacă nu poate fi modificat cu uşurință.

Rigiditate este cauzată de faptul că o singură modificare într-un modul produce o cascadă de modificări în modulele dependente de acesta, datorită interdependenței.

Când numărul de modificări (care survin ca urmare a primei modificări) nu poate fi prevăzut, atunci impactul modificării nu poate fi estimat, şi implicit nici costul modificării.

Fragilitatea este tendința unui program de a nu mai funcționa atunci când se face o modificare.

În acest caz, survin o serie de noi probleme; cel mai adesea, acestea sunt de natură diferită decât a modificării inițiale. Rezolvarea acestora conduce la alte probleme. O mare fragilitate conduce la un software de o calitate scăzută.

Un design este **imobil** atunci când un modul din program, care se doreşte a fi detaşat şi folosit într-o altă aplicație, este dependent de detaliile din restul aplicației. Adesea, costul pentru separarea modulului dorit de restul aplicației este mult mai mare decât dacă se realizează un nou modul, acesta fiind unul dintre motivele pentru care se renunță la re-folosirea / re-utilizarea piesei de software.

Un sistem software este un sistem dinamic, care de-a lungul ciclului de viață se modifică inevitabil.

Modificarea cerințelor/specificațiilor pentru un sistem software poate duce la dezvoltarea unui design greşit.

Specificarea cerințelor se realizează într-un document scris care trebuie să poată fi citit şi înțeles atât de beneficiar cât şi de proiectant. Altfel, beneficiarul nu poate să-l avizeze.

Motive pentru care cerințele se schimbă:

- Beneficiarul (clientul) vine cu noi cerințe, adăugării / modificări de funcții;
- Analistul sau persoana care se ocupă de proiect (şi a formulat inițial cerințele aplicației) a interpretat greşit cererile clientului;

- În timpul procesului de discuții cu beneficiarul, unele din cerințele inițiale ale clientului pot fi "uitate". În faza de feedback de la client, aceste cerințe sunt reconsiderate. Ca urmare, apar modificări ale programului;
- alte cauze: o lege nouă, o nouă politică de aprovizionare a firmei, o reorganizare sau o fuziune a firmei pentru care este dezvoltată aplicația, etc.
- Cu cât durata de viață a proiectului este mai lungă, cu atât este mai vulnerabil la modificări de acest tip.

Atât sistemele cu un design bun cât şi cele cu design greşit sunt supuse modificărilor. Diferența dintre ele este că proiectarea bună este stabilă când sunt supuse modificării.

1.3 Satisfacerea cerințelor utilizatorilor şi costul software-ului

Nici un produs, deci nici produsele software, nu vor fi solicitate şi utilizate dacă ele nu răspund unor nevoi ale utilizatorilor.

Cu cât sunt mai bine acoperite cerințele şi cu cât produsul va răspunde mai bine acestor solicitări, cu atât cererea pentru sistemul (produsul) respectiv va fi mai mare.

Statisticile arată că în țările puternic industrializate, ponderea ocupată de costul software-ului în produsul național brut este în continuă creştere.

Costul este influențat, şi determinat totodată, de:

- productivitatea de programare (10-20 instr / zi),
- predicția timpului în care se va realiza produsul final,
- costul hardware-ului în raport cu cel al software-ului,
- utilizarea generatoarelor de programe, etc.

Costul software-ului este dat în principal de salarii.

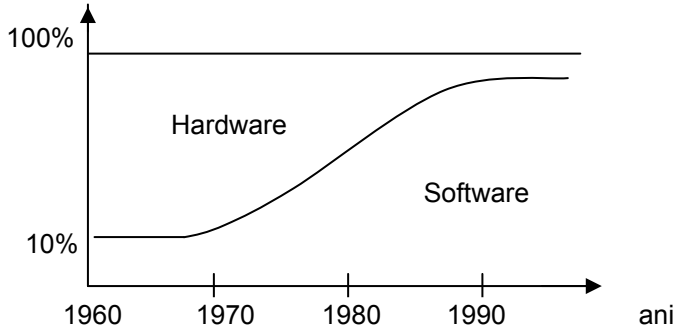


Fig.1.1. Relația între costul hardware-ului şi software-ului în timp

Rezumând, se poate spune că software-ul este scump, în primul rând din cauza productivității scăzute a programatorilor.

Astfel se naşte, în mod natural, întrebarea : Cum poate fi scăzut costul? Este interesant de văzut care parte din dezvoltarea unui produs software costă mai mult. Fig. 1.2 ilustrează acest lucru.

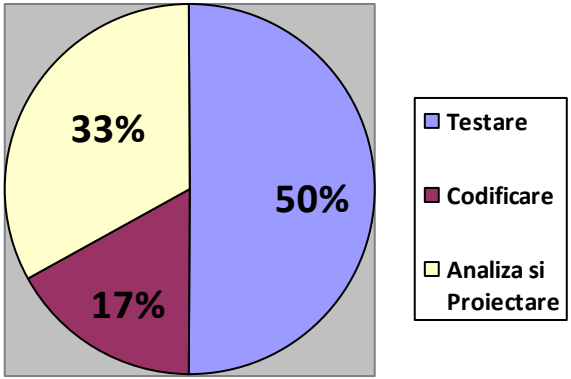


Fig. 1.2 Costul relativ în diferite stadii de dezvoltare ale software-ului

Dacă totuşi erorile sunt o problemă majoră, atunci, când apar ele?

Fig.1.3 arată numărul relativ de erori comise în diferite stadii de evoluție a software-ului.

- Dar problema este puțin mai dificilă şi constă în a stabili cât costă depistarea unei erori.
- Se ştie că o eroare nedescoperită costă mai mult decât ar fi costat ea dacă ar fi fost descoperită şi înlăturată la timp.

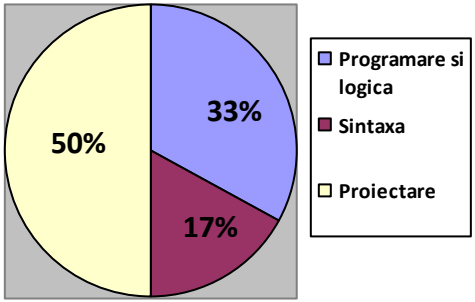


Fig. 1.3. Numărul relativ de erori de-a lungul stadiilor de evoluție ale software-ului

- Erorile de sintaxă sunt descoperite automat de către compilatoare, la prima compilare, şi pot fi corectate cu uşurință.

- Din contră, erorile de proiectare pot fi detectate de abia în faza de testare, ceea ce implică o activitate, câteodată destul de laborioasă, de reproiectare.

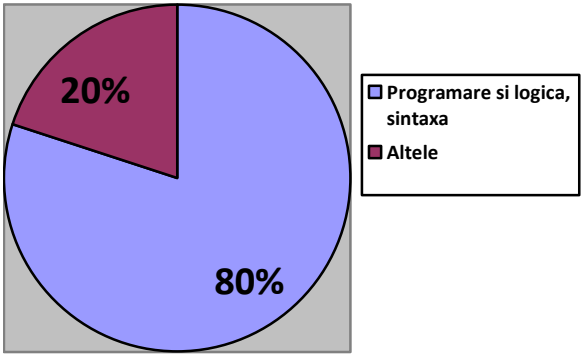


Fig.1.4 Costul relativ pentru depistarea diferitelor tipuri de erori software

1.4 Performanța, portabilitatea şi mentenanța software-ului

Performanța unui program este numită adesea *eficiență*.

Acestă terminologie datează din vremea când viteza hardware-ului era destul de mică iar costul calculatorului relativ mare, ceea ce însemna că efortul trebuia îndreptat spre utilizarea cât mai judicioasă a memoriei şi procesorului central, printr-un program cât mai eficient care conducea în final, la scurtarea timpului de execuție şi deci la o performanță mai bună a sistemului.

În momentul de față, prin performanță se subînțelege:

- un răspuns al sistemului într-o perioadă de timp rezonabilă;
- obținerea unui semnal de control la ieşire (într-un timp rezonabil);

Timpul scurt de execuție şi utilizarea unei memorii mici sunt două cerințe mutual contradictorii.

Mentenanța este termenul folosit pentru orice activitate de întreținere a unei "piese" software, după ce ea a fost dată în exploatare. Sunt două tipuri de mentenanță:

a). *corectivă* - prin care se înlătură erorile apărute în timpul exploatării;

b). *adaptivă* - care apare ca urmare a schimbărilor intervenite în solicitările utilizatorilor, în sistemul de operare sau în limbajele de programare.

O idee despre cât reprezintă activitățile de mentenanță raportate la întregul produs software se poate ilustra în fig. 1.5.

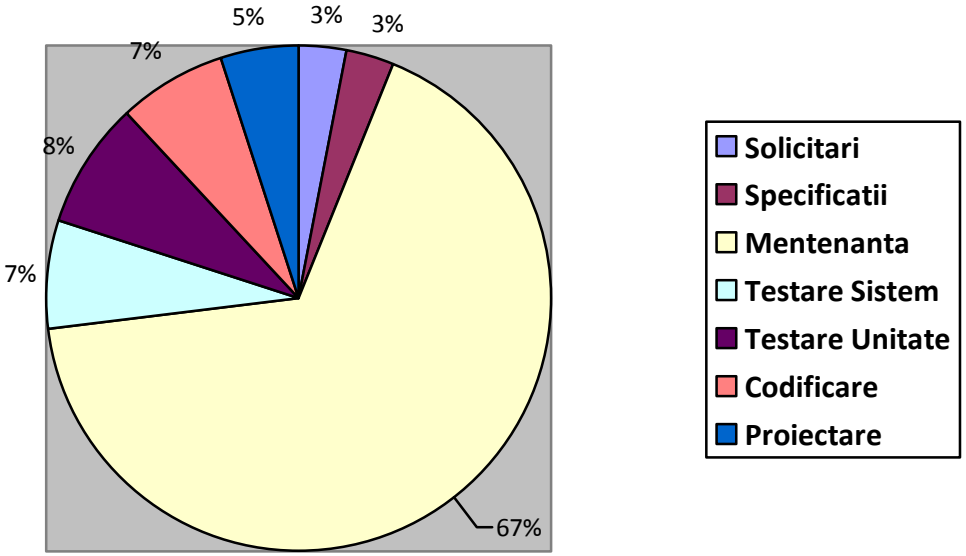


Fig.1.5 Proporția activităților în cadrul realizării unui produs software

1.5 Fiabilitatea

În termeni generali fiabilitatea software reprezintă capacitatea sistemului de a răspunde cerințelor utilizatorului (de a-şi executa misiunea), în conformitate cu specificațiile sale de proiectare.

În mod curent, testarea este principala tehnică care dă certitudinea că software-ul lucrează corect.

Dar problema se complică atunci când trebuie stabilit timpul în care este testată o piesă software pentru a avea certitudinea că este corectă.

1.6 Cerințe pentru ingineria sistemelor de programe

Aşa cum a reieşit din cele expuse până acum există câteva cerințe obligatorii pentru ca un produs informatic, un sistem software să fie utilizat şi anume:

- satisfacerea cât mai completă a cerințelor utilizatorului;
- cost de producție cât mai scăzut;
- performanță ridicată;
- portabilitate;
- cost de întreținere scăzut (mentenanță bună);

- fiabilitate ridicată;

- livrare la timp.

Cele mai multe dintre acestea, sunt în conflict, adică nu pot fi satisfăcute simultan la maximum.

În consecință, în funcție de domeniul de aplicație şi de cerințele problemei se va face o ierarhie a cerințelor acordând prioritate maximă celor care sunt critice pentru sistem.

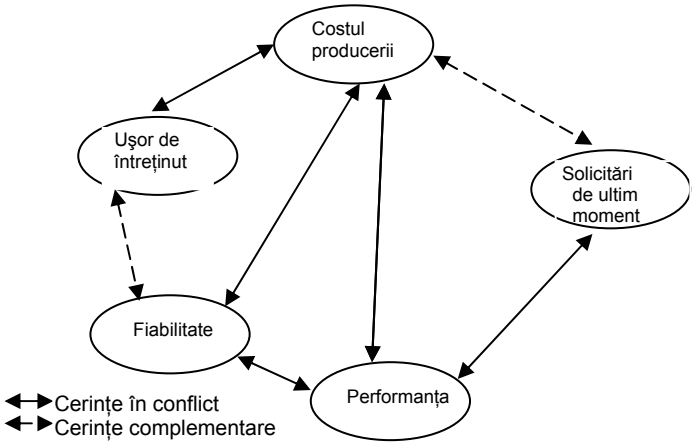


Fig. 1.6 Cerințe complementare şi în conflict în ingineria software

1.7 Teoremă pentru ingineria software

Dacă M(P) este măsura problemei P,
C(P) = costul scrierii programului P
Pt 2 probleme P şi Q, pt care M(P) > M(Q) atinci C(P) > C(Q).

Dacă combinăm cele 2 probleme, P+ Q , atunci

M(P+Q) > M(P) + M(Q).

C(P+Q) > C(P) + C(Q).

Rezultă că este mai uşor de creat două programe mici decât unul mare care cumulează funcțiile ambelor programe.

Pe măsură ce complexitatea creşte, pot apare şi mai multe erori, generate şi de interacțiunea dintre programe.

Fenomenul menționat este caracteristic tuturor domeniilor în care se rezolvă probleme.

Astfel pentru cazul proiectării programelor se poate obține o curbă a erorilor de felul celei prezentate în fig. 1.7.

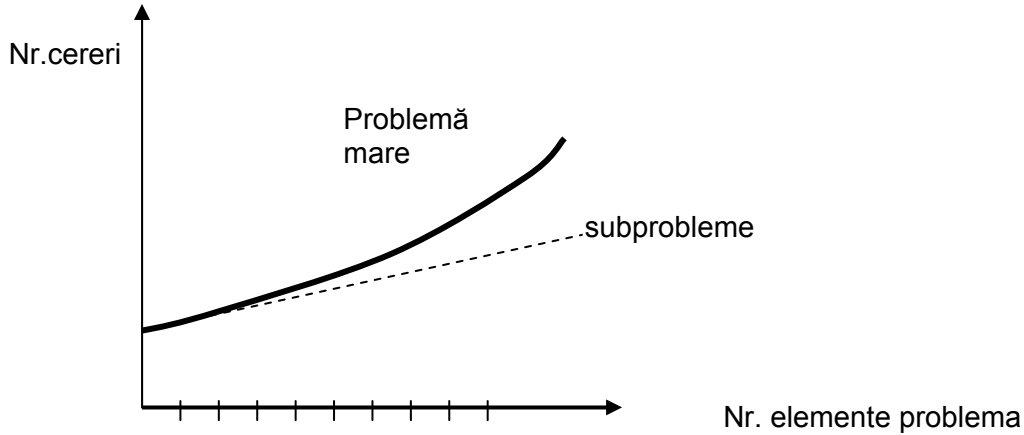


Fig.1.7. Curba erorilor la descompunerea problemei în subprobleme

Acest efect este determinat de limitările ființei umane în prelucrarea informațiilor. Se pare că suntem capabili la un moment dat să prelucrăm simultan şi complet informații referitoare numai la aproximativ şapte obiecte, entități sau concepte distincte.

Peste 7± 2 entități distincte ce trebuie folosite simultan, numărul de erori comise creşte mult mai repede.

în cazul problemelor mari pentru a învinge complexitatea cu un cost mai redus trebuie să se recurgă la descompunerea problemei în module mai mici şi cvasiindependente.

Făcând o substituție în relația de mai sus se obține:

$$C(P) > C(P') + C(P'')$$

numită teorema fundamentală a ingineriei programării, în care P' şi P'' sunt două subprobleme independente prin a căror rezolvare se soluționează la un cost mai scăzut problema P.

Legendă: 1- Cost realizare module
2- Cost realizare interfețe între module.
3- Cost total

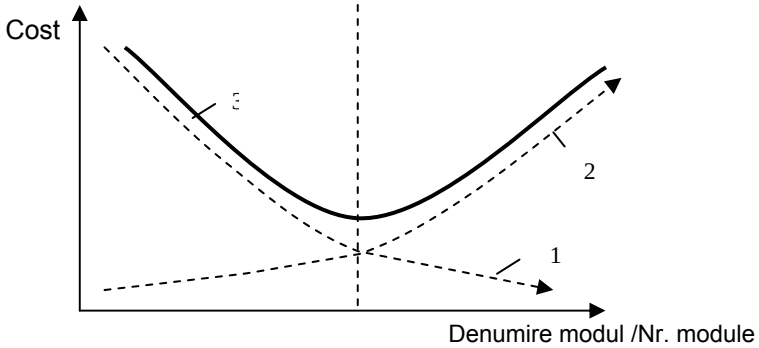


Fig.1.8. Relațiile cost-dimensiune modul şi cost-număr module

Existentă un cost optim de realizare care determină o dimensiune optimă de modul şi un număr optim de module în care proiectul poate fi descompus.

Evident acest optim nu poate fi precizat cu exactitate, ci trebuie căutat în fiecare caz în parte.

Soluțiile la problemele software sunt :

- dezvoltarea sistematică în toate stadiile de dezvoltare a unei piese software ;
- asistența calculatorului pentru dezvoltarea software-ului (limbaje de generația a patra, medii pentru dezvoltare software, proiectare asistată pentru inginerie software – instrumente software (CASE), UML, etc.) ;
- concentrarea efortului pentru depistarea cât mai exactă a dorințelor utilizatorilor ;
- utilizarea specificării formale pentru cerințele sistemului ;
- demonstrația făcută în fața beneficiarilor printr-o versiune timpurie a sistemului (prototipizare) ;
- utilizarea de limbaje de programare noi ;
- creşterea efortului pentru asigurarea că software-ul nu are erori .

Una dintre ideile dominante în abordarea dezvoltării software-ului este dezvoltarea pe baza ciclului de viață sau modelul « cascadă ».

În acest context se împarte dezvoltarea unui produs informatic în paşi independenți, aflați într-o anumită succesiune.

Paşii succesivi sunt:

stabilirea cerințelor

↓

specificarea

↓

proiectarea

↓

implementarea

↓

testarea

↓

operarea şi întreținerea

Specialiştii au idei diferite despre ce trebuie să conțină exact fiecare pas în parte, dar principiile modelului ciclului de viața sunt :

1. Există o serie succesivă de paşi ;
2. Fiecare pas este bine definit ;
3. Fiecare pas creează un produs definit (adesea doar o foaie de hârtie) ;
4. Corectitudinea fiecărui pas trebuie verificată cu grijă.

Utilizarea explicită a unei discipline de conducere a proiectului este factorul cheie în obținerea unei calități software ridicate.

1.9. Documentația sistemelor de programe

O parte extrem de importantă a oricărui produs software este documentația, de care va depinde ulterior utilizarea, întreținerea sistemului şi eventual extensibilitatea.

În mod normal, documentația este elaborată în cu două scopuri.

Primul este de a descrie pachetul software şi modul lui de utilizare.

Această documentație este cunoscută sub numele de **documentație de utilizare** şi este destinată utilizatorului sistemului, având un caracter mai puțin tehnic.

Cel de al doilea scop urmărit de documentație este de a descrie sistemul astfel încât acesta să fie întreținut pe durata celorlalte perioade de viață.

Documentația de acest tip este cunoscută sub numele de **documentație de sistem** şi are în mod inevitabil un caracter mult mai tehnic decât documentația de utilizare.

Astfel documentarea sistemului începe cu dezvoltarea specificațiilor inițiale şi continuă pe toată durata de viață a produsului software.

Documentația va consta în final din toate documentele elaborate în faza de dezvoltare a sistemului, inclusiv specificațiile conform cărora a fost verificat sistemul, diagramele de flux de date şi de relații între entități, dicționarul de date şi schemele de sistem, care reflectă structura sa modulară.

De o mare importanță sunt versiunile sursă ale tuturor programelor din sistem, care trebuie prezentate într-un format lizibil. De aceea, specialiştii încurajează limbajele de nivel înalt, bine proiectate, includerea în programe a comentariilor şi proiectarea modulară, care permite prezentarea fiecărui modul în parte ca un întreg corect.

Faptul că documentarea sistemului trebuie să fie un proces permanent duce la apariția unor conflicte între principiile ingineriei software şi natura umană.

1.10. Dreptul de proprietate şi garanții

Din păcate, problemele referitoare la dreptul de proprietate asupra produselor software nu se încadrează foarte clar în legile existente privind protecția drepturilor de autor şi a brevetelor.

Deşi aceste legi au fost elaborate tocmai pentru a permite fabricantului unui "produs" să-şi facă public produsul, păstrându-şi totodată drepturile de proprietate asupra lui, particularitățile produselor software au pus adesea instanțele în mare dificultate în eforturile lor de aplica şi în acest caz legile generale privind proprietatea intelectuală.

Inițial, drepturile de autor au fost elaborate pentru a proteja operele literare. În acest caz, valoarea nu stă în ideile exprimate, ci mai degrabă în modul în care sunt redate aceste idei. Valoarea unei opere literare rezidă din stil, formă şi nu neapărat din subiect.

În cazul unui produs software, ideea se exprimă în algoritm.

Spre deosebire însă de o poezie sau un roman, valoarea unui produs software nu constă în maniera particulară în care este exprimat algoritmul, ci chiar din algoritm.

În multe cazuri, costurile fazei de dezvoltare a unui pachet software se concentrează tocmai în descoperirea unor algoritmi, şi mai puțin în reprezentarea acestora.

În domeniul drepturilor de proprietate asupra pachetelor software, utilizarea brevetelor se loveşte de mai multe obstacole, dintre care cel mai important este principiul general care statuează că nimeni nu poate fi proprietarul unor fenomene naturale, cum ar fi legile fizicii sau formulele matematice.

În general, instanțele au decis că algoritmii intră şi ei în această categorie, astfel că şi în acest caz legea lasă neprotejată cea mai valoroasă componentă a programului – algoritmul. În plus, obținerea unui brevet este un proces costisitor şi îndelungat, a cărui durată poate fi de ordinul anilor. În acest timp, un produs software se uzează moral, iar până la obținerea brevetului de invenție solicitantul nu are nici un drept de exclusivitate.

Legile dreptului de autor şi cele legate de brevete au rolul de a încuraja atât creşterea numărului de invenții, cât şi schimbul liber de idei, în beneficiul societății.

Atunci când drepturile le sunt protejate, creatorii şi inventatorii sunt mai tentați să îşi aducă realizările la cunoştința publicului.

Adesea, producătorii de software precizează pentru produsele lor un set de garanții prin care îşi stabilesc limitele de responsabilitate.

Acestea iau forme ca: “În nici o situație compania X nu îşi asumă răspunderea pentru pagubele de orice fel provocate prin utilizarea acestui software” sau “Compania Y nu garantează că acest software corespunde exact cerințelor dumneavoastră”.

Etapele de dezvoltare a sistemelor de programe

2.1. Ciclul de viaţă

Există patru faze fundamentale ale metodologiilor ingineriei software:

- analiza (ce dorim să construim);
- proiectarea (cum vom construi);
- implementarea (construirea propriu-zisă);
- testarea (asigurarea calităţii).

Deşi aceste faze se referă în mod special la *ciclul de viaţă* al produsului software, ele pot fi aplicate şi altor stadii de existenţă prin care trece un program de la „naştere” până la „moarte”: lansare, întreţinere, ieşire din uz.

2.1.1. Faza de analiză

- defineşte *cerinţele* sistemului, independent de modul în care acestea vor fi îndeplinite.
- se defineşte problema pe care clientul doreşte să o rezolve.

- Rezultatul = *documentul care conţine specificarea cerinţelor* şi care trebuie să precizeze clar *ce* trebuie construit.

> formularea problemei, aşteptările clientului sau criteriile pe care trebuie să le îndeplinească produsul.

Documentul cerinţelor, numit *specificarea cerinţelor* poate fi realizat într-o manieră formală, bazată pe logică matematică, sau poate fi exprimat în limbaj natural.

el descrie *obiectele* din sistem şi *acţiunile* care pot fi realizate cu ajutorul obiectelor.

Descrierea obiectelor şi acţiunilor trebuie să fie generală si să nu depindă de o anumită tehnologie.

Desigur, într-o abordare POO, descrierile vor lua forma obiectelor şi metodelor, însă în alte abordări, obiectele pot fi de exemplu servicii care accesează baze de date.

Descrierile nu implică proiectarea arhitecturii aplicaţiei, ci enumerarea părţilor componente şi a modului în care acestea se comportă.

Mai târziu, în faza de proiectare, acestea vor fi transformate în primitive informatice, precum liste, stive, arbori, grafuri, algoritmi şi structuri de date.

Mai concret, documentul trebuie să conţină descrieri pentru următoarele categorii:

- ***Obiecte****:* se defineşte mai întâi ontologia sistemului, care este bazată pe construcţii substantivale pentru identificarea pieselor, părţilor componente, constantelor, numelor şi a relaţiilor dintre acestea;
- ***Acţiuni****:* se definesc acţiunile pe care trebuie să le îndeplinească sistemul şi care sunt sugerate în general de construcţii verbale. Exemple de acţiuni sunt: metodele, funcţiile sau procedurile;
- ***Stări****:* Sunt definite ca mulţimi de setări şi valori care disting sistemul între două ipostaze spaţio-temporale. Exemple de stări sunt: starea iniţială, cea finală sau stările de eroare.
- ***Scenarii tipice****:* Un scenariu este o secvenţă de paşi urmaţi pentru îndeplinirea unui scop. Când sistemul este terminat şi aplicaţia este disponibilă, clientul trebuie să poată utiliza, într-o

manieră cât mai facilă si clar specificată, toate scenariile tipice ale aplicaţiei. Scenariile tipice trebuie să reprezinte majoritatea scenariilor de utilizare ale aplicaţiei.

- ***Scenarii atipice****:* Un scenariu atipic trebuie să fie îndeplinit de sistem numai în cazuri speciale. Clientul poate să spere, de exemplu, că o eroare neprevăzută este un eveniment atipic. Totuşi, sistemul trebuie să gestioneze un număr cât mai mare de categorii de erori, prin tehnici stabilite, precum tratarea excepţiilor, monitorizarea proceselor etc.;
- ***Cerinţe incomplete sau nemonotone***: O enumerare completă a cerinţelor, pentru toate situaţiile care pot apărea în condiţii de lucru reale, nu este posibilă. Procesul de stabilire a cerinţelor are o natură iterativă si nemonotonă. Noile cerinţe pot infirma soluţiile vechi. Pe măsură ce un sistem creşte în dimensiuni şi complexitate, stabilirea cerinţelor devine din ce în ce mai dificilă, mai ales când procesul de colectare a cerinţelor este distribuit, fiind realizat de indivizi cu specializări diferite.

2.1.2. Faza de proiectare

Pe baza cerinţelor din faza de analiză, acum se stabileşte ***arhitectura*** sistemului: componentele sistemului, ***interfeţele*** şi modul lor de ***comportare***:

- ***Componentele*** sunt elementele constructive ale produsului. Acestea pot fi create de la zero sau reutilizate dintr-o bibliotecă de componente. Componentele rafinează si capturează semnificaţia detaliilor din documentul cerinţelor;
- ***Interfeţele*** ajută la îmbinarea componentelor. O interfaţă reprezintă graniţa dintre două componente, utilizată pentru comunicarea dintre acestea. Prin intermediul interfeţei, componentele pot interacţiona;
- ***Comportamentul***, determinat de interfaţă, reprezintă răspunsul unei componente la stimulii acţiunilor altor componente.

Se identifică detaliile privind limbajele de programare, mediile de dezvoltare, dimensiunea memoriei, platforma, algoritmii, structurile de date, definiţiile globale de tip, interfeţele etc.

- trebuie indicate şi *priorităţile critice* pentru implementare. Acestea sugerează sarcinile care, dacă nu sunt executate corect, conduc la eşecul sistemului. Totuşi, chiar dacă priorităţile critice sunt îndeplinite, acest fapt nu duce automat la succesul sistemului, însă creşte nivelul de încredere că produsul va fi o reuşită.

Folosind scenariile tipice şi atipice, trebuie realizate compromisurile inerente între performanţă si complexitatea implementării.

Analiza performanţelor presupune studierea modului în care diferitele arhitecturi conduc la diferite caracteristici de performanţă pentru fiecare scenariu tipic.

În funcţie de frecvenţa de utilizare a scenariilor, fiecare arhitectură va avea avantaje şi dezavantaje.

Un răspuns rapid la o acţiune a utilizatorului se realizează deseori pe baza unor costuri de resurse suplimentare: indecşi, managementul cache-ului, calcule predictive etc.

Dacă o acţiune este foarte frecventă, ea trebuie realizată corect şi eficient.

O acţiune mai rară trebuie de asemenea implementată corect, dar nu este evident care e nivelul de performanţă necesar în acest caz.

O situaţie în care o astfel de acţiune trebuie implementată cu performanţe maxime este închiderea de urgenţă a unui reactor nuclear.

Planul de implementare stabileşte programul după care se va realiza implementarea şi resursele necesare (mediul de dezvoltare, editoarele, compilatoarele etc.).

Planul de test defineşte testele necesare pentru stabilirea calităţii sistemului. Dacă produsul trece toate testele din planul de test, este declarat terminat.

Cu cât testele sunt mai amănunţite, cu atât este mai mare încrederea în sistem şi deci creşte calitatea sa. Un anume test va verifica doar o porţiune a sistemului. *Acoperirea testului* este procentajul din produs verificat prin testare. În mod ideal, o acoperire de 100% ar fi excelentă, însă ea este rareori îndeplinită. De obicei, un test cu o acoperire de 90% este simplu, însă ultimele 10% necesită o perioadă de timp semnificativă.

În general, este suficient ca testele să cuprindă scenariile tipice şi atipice, fără să verifice întregul sistem, cu absolut toate firele de execuţie. Acesta poate conţine ramificaţii interne, erori sau întreruperi care conduc la fire de execuţie netestate. Majoritatea sistemelor sunt pline de bug-uri

nedescoperite. De obicei, clientul participă în mod logic la testarea sistemului şi semnalează erori care vor fi îndepărtate în versiunile ulterioare.

2.1.3. Faza de implementare

În această fază, sistemul este construit, ori plecând de la zero, ori prin asamblarea unor componente pre-existente. Pe baza documentelor din fazele anterioare, echipa de dezvoltare ar trebui să ştie exact ce trebuie să construiască, chiar dacă rămâne loc pentru inovaţii şi flexibilitate.

Echipa trebuie să gestioneze problemele legate de calitate, performanţă, biblioteci şi depanare. Scopul este producerea sistemului propriu-zis.

O problemă importantă este *îndepărtarea erorilor critice*.

Într-un sistem există trei tipuri de erori:

- *Erori critice:* Împiedică sistemul să satisfacă în mod complet scenariile de utilizare. Aceste erori trebuie corectate înainte ca sistemul să fie predat clientului şi chiar înainte ca procesul de dezvoltare ulterioară a produsului să poată continua;

- *Erori necritice:* Sunt cunoscute, dar prezenţa lor nu afectează în mod semnificativ calitatea observată a sistemului. De obicei aceste erori sunt listate în notele de lansare şi au modalităţi de ocolire bine cunoscute;
- *Erori necunoscute:* Există întotdeauna o probabilitate mare ca sistemul să conţină un număr de erori nedescoperite încă. Efectele acestor erori sunt necunoscute. Unele se pot dovedi critice, altele pot fi rezolvate cu patch-uri sau eliminate în versiuni ulterioare.

2.1.4. Faza de testare

Calitatea produsului software este foarte importantă. Multe companii nu au învăţat însă acest lucru şi produc sisteme cu funcţionalitate extinsă, dar cu o calitate scăzută. Totuşi, e mai simplu să-i explici clientului de ce lipseşte o anumită funcţie decât să-i explici de ce produsul nu este performant. Un client satisfăcut de calitatea produsului va rămâne loial firmei şi va aştepta noile funcţii în versiunile următoare.

În multe metodologii ale ingineriei software, faza de testare este o fază separată, realizată de o echipă *diferită* după ce implementarea s-a terminat. Motivul este faptul că un programator nu-şi poate descoperi foarte uşor propriile greşeli.

O persoană nouă care priveşte codul poate descoperi greşeli evidente care scapă celui care citeşte şi reciteşte materialul de multe ori.

Din păcate, această practică poate determina o atitudine indiferentă faţă de calitate în echipa de implementare.

Tehnicile de testare sunt abordate preponderent din perspectiva producătorului sistemului. În mod ideal, şi clientul trebuie să joace un rol important în această fază.

Testele de regresiune (engl. „regression test") sunt colecţii de programe care testează una sau mai multe trăsături ale sistemului. Rezultatele testelor sunt adunate şi dacă există erori, eroarea este corectată. Un test de regresiune valid generează rezultate verificate, numite „standardul de aur".

Validitatea rezultatului unui test ar trebui să fie determinată de documentul cerinţelor. În practică, echipa de implementare este responsabilă de interpretarea validităţii.

Testele sunt colectate, împreună cu rezultatele standardelor de aur, într-un pachet de test de regresiune. Pe măsură ce dezvoltarea continuă, sunt adăugate mai multe teste noi, iar testele vechi pot rămâne valide sau nu. Dacă un test vechi nu mai este valid, rezultatele sale sunt modificate în standardul de aur, pentru a se potrivi aşteptărilor curente. Pachetul de test este rulat din nou şi

generează noi rezultate. Acestea sunt comparate cu rezultatele standardelor de aur. Dacă sunt diferite, înseamnă că în sistem a apărut o eroare. Eroarea este corectată si dezvoltarea continuă. Acest mecanism detectează situaţiile când starea curentă de dezvoltare a produsului invalidează o stare existentă. Astfel, se previne regresiunea sistemului într-o stare de eroare anterioară.

Există patru puncte de interes în testele de regresiune pentru asigurarea calităţii.

Testarea internă tratează implementarea de nivel scăzut. Fiecare funcţie sau componentă este testată de către echipa de implementare. Aceste teste se mai numesc teste „clear-box” sau „white-box”, deoarece toate detaliile sunt vizibile pentru test.

Testarea unităţilor testează o unitate ca un întreg. Aici se testează interacţiunea mai multor funcţii, dar numai în cadrul unei singure unităţi.

Testarea este determinată de arhitectură. De multe ori sunt necesare aşa-numitele „schele”, adică programe special construite pentru stabilirea mediului de test.

Numai când mediul este realizat se poate executa o evaluare corectă.

Programul schelă stabileşte stări şi valori pentru structurile de date şi asigură funcţii externe fictive.

De obicei, programul schelă nu are aceeaşi calitate ca produsul software testat şi adesea este destul de fragil.

O schimbare mică în test poate determina schimbări importante în programul schelă.

Aceste teste se mai numesc teste „black-box” deoarece numai detaliile interfeţei sunt vizibile pentru test.

Testarea internă si a unităţilor poate fi automatizată cu ajutorul instrumentelor de acoperire (engl. „coverage tools”), care analizează codul sursă şi generează un test pentru fiecare alternativă a firelor de execuţie.

Depinde de programator combinarea acestor teste în cazuri semnificative care să valideze rezultatelor fiecărui fir de execuţie.

De obicei, instrumentul de acoperire este utilizat într-un mod oarecum diferit: el urmăreşte liniile de cod executate într-un test şi apoi raportează procentul din cod executat în cadrul testului.

Dacă acoperirea este mare şi liniile sursă netestate nu prezintă mare importanţă pentru calitatea generală a sistemului, atunci nu mai sunt necesare teste suplimentare.

Testarea aplicaţiei testează aplicaţia ca întreg şi este determinată de scenariile echipei de analiză.

Aplicaţia trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziţia clientului.

Spre deosebire de testarea internă si a unităţilor, care se face prin program, testarea aplicaţiei se face de obicei cu scripturi care rulează sistemul cu o serie de parametri şi colectează rezultatele.

În trecut, aceste scripturi erau create manual. În prezent, există instrumente care automatizează si acest proces. Majoritatea aplicaţiilor din zilele noastre au interfeţe grafice (GUI).

Testarea interfeţei grafice pentru asigurarea calităţii poate pune anumite probleme.

Cele mai multe interfeţe, dacă nu chiar toate, au bucle de evenimente, care conţin cozi de mesaje de la mouse, tastatură, ferestre etc.şi care asociate cu fiecare eveniment sunt coordonatele ecran.

Testarea interfeţei presupune deci memorarea tuturor acestor informaţii şi elaborarea unei modalităţi prin care mesajele să fie trimise din nou aplicaţiei, la un moment ulterior.

Testarea la stres determină calitatea aplicaţiei în mediul său de execuţie.

Ideea este crearea unui mediu mai solicitant decât cel în care aplicaţia va rula în mod obişnuit.

Aceasta este cea mai dificilă si complexă categorie de teste.

Sistemul este supus unor cerinţe din ce în ce mai numeroase, până când acesta cade.

Apoi produsul este reparat şi testul de stres se repetă până când se atinge un nivel de stres mai ridicat decât nivelul aşteptat de pe staţia clientului.

Deseori apar aici conflicte între teste. Fiecare test funcţionează corect atunci când este făcut separat.

Când două teste sunt rulate în paralel, unul sau ambele teste pot eşua. Cauza este de obicei managementul incorect al accesului la resurse critice.

Mai apar şi probleme de memorie, când un test îşi alocă memorie şi apoi nu o mai eliberează.

Testul pare să funcţioneze corect, însă după ce este rulat de mai multe ori, memoria disponibilă se reduce iar sistemul cade.

2.2. Cerinţe – Specificaţii

În mod logic, prima etapă în realizarea unui produs software constă în stabilirea cu precizie a ceea ce utilizatorul doreşte de la sistem.

Dominanta în această etapă o constituie comunicarea dintre beneficiar şi inginerul software.

Inginerul care se ocupă de stabilirea cerinţelor utilizatorului, ceea ce de fapt reprezintă analiza sistemului, se va numi simplu *analist.*

utilizatorului colaborează cu analistul pentru definirea cerinţelor şi specificaţiilor de proiectare ale sistemului.

Utilizatorul poate să aibă o idee foarte vagă despre ceea ce doreşte sau, din contra, el poate să ştie foarte exact.

În mod foarte categoric, stabilirea specificaţiilor de proiectare este o etapă deosebit de importantă în dezvoltarea ulterioară a sistemului.

2.2.1. Noţiunea de "cerinţă"

Cerinţa reprezintă *ceea ce doreşte* de fapt utilizatorul, fără a se preciza de fapt cum se realizează acel lucru.

Una dintre marile controverse din Computer Science se enunţă astfel:

« Este sau nu este necesar să se specifice şi *cum* realizează sistemul o anumită solicitare». Aceasta este relaţia dintre specificare şi implementare.

De cele mai multe ori, utilizatorul nu agreează să i se explice cum va fi implementat sistemul, sau mai mult, nici nu-l interesează.

Pentru analist însă acest lucru este important pentru că, în funcţie de varianta aleasă, ştie cum să-şi orienteze investigaţiile şi, de asemenea, ce restricţii are.

Analistul trebuie:

- să verifice dacă o anumită solicitare este tehnic posibilă.
- este vital să ia în considerare implementarea pentru a putea estima costul şi data de livrare a sistemului.

- dacă specificaţiile pot să constituie un ghid privind modul în care sistemul răspunde dorinţelor utilizatorilor.

Câteodată, specificaţiile suferă de deficienţe majore, cum ar fi: sunt incomplete, nu există nici o menţiune despre cost sau termen de livrare.

2.2.2. Procesul de alegere a cerinţelor

Activitatea de selectare a cerinţelor presupune colaborarea şi lucrul împreună atât al analistului cât şi al utilizatorului.

Se pot distinge trei feluri de activităţi în cadrul acestui proces de specificare a cerinţelor, şi anume:

- ascultarea (sau selectarea cererilor);
- analizarea cererilor;
- scrierea (sau definirea cererilor).

Selectarea presupune ascultarea nevoilor utilizatorilor, punând întrebări astfel încât utilizatorii să-şi poată clarifica cât mai bine cererile, ca în final să se înregistreze punctul de vedere al utilizatorului privind specificaţiile sistemului.

Analiza este etapa în care efectiv inginerul software se gândeşte cum poate transforma punctul de vedere al utilizatorului, privind sistemul, într-o reprezentare care să poată fi implementată în sistem.

Acest lucru poate fi adesea dificil din cauza existenţei unor puncte de vedere diferite ale diverşilor utilizatori.

Definirea cererilor înseamnă scrierea într-o manieră clară, adesea în limbaj natural, a ceea ce sistemul trebuie să furnizeze utilizatorului. Această informaţie se numeşte ***specificarea cererilor***.

Rezumând, rolul analistului este:

1. Să achiziţioneze şi să selecteze cererile de la utilizatori.
2. Să ajute la rezolvarea diferenţelor posibile dintre utilizatori.
3. Să avizeze utilizatorul despre ceea ce este tehnic posibil sau imposibil.
4. Să clarifice cererile utilizatorilor.

5. Să documenteze solicitările (se va vedea în secţiunea următoare).

6. Să negocieze şi în final să pună de acord diferitele opinii al utilizatorilor pentru formularea specificaţiilor.

2.2.4. Specificarea cerinţelor

Specificarea este documentul de referinţă prin care este asigurată dezvoltarea sistemului.

Cei trei factori importanţi care sunt luaţi în consideraţie sunt:

1) nivelul de detaliere - ; nevoia de a restrânge specificaţiile, pe cât posibil numai la ceea *ce* "face sistemul" şi nu la *cum* va realiza cerinţa respectivă
2) cui este adresat documentul - utilizatori şi proiectanţi;
3) notaţiile utilizate.

Utilizatorii preferă o descriere netehnică, exprimată în limbaj natural, improprie pentru specificaţii precise, consistente şi neambiguii.

Pe de altă parte, analistul având o orientare tehnică, va dori probabil să utilizeze o notaţie precisă (chiar matematică) pentru a specifica sistemul.

Există câteva notaţii pentru scrierea specificaţiilor:

- *informale* - scrise în limbaj natural, utilizate cât mai clar şi cu cât mai multă grijă posibilă;
- *formale* - utilizând notaţii matematice, cu rigoare şi consistenţă;
- *semiformale* - utilizând o combinaţie de limbaj natural cu diferite diagrame şi notaţii tabelare.

În momentul de faţă, cele mai multe specificaţii sunt scrise în limbaj natural, la care se adaugă notaţii formale, cum ar fi diagramele de flux, diagramele UML pentru a clarifica textul.

Varianta modernă este de a elabora două documente şi anume:

1. Specificaţiile cererilor scrise în primul rând pentru utilizatori, pentru a descrie punctul lor de vedere asupra sistemului, exprimat în limbaj natural. Acestea constituie substanţa contractului dintre utilizatori şi proiectanţi.
2. O specificare tehnică, care este folosită în primul rând de proiectanţi, exprimată într-o formă de notaţii formale şi descriind numai o parte a informaţiei în toate specificaţiile cererilor.

O lista de verificare a conţinutului pentru specificaţiile solicitărilor trebuie să arate astfel:

- solicitări funcţionale;
- solicitări de date;
- restricţii;
- ghid de aplicare.

Solicitări funcţionale

sunt esenţa specificaţiilor cererilor. Ele indică sistemului "ceea ce trebuie să facă" şi sunt caracterizate de verbe şi realizează acţiuni.

Exemple:

- Sistemul va afişa titlurile şi toate cărţile scrise de acelaşi autor.
- Sistemul va afişa permanent temperaturile tuturor maşinilor.

Solicitări de date - au două componente:

1. Date de intrare sau ieşire pentru sistem, cum ar fi de exemplu dimensiunile ecranului;
2. Date care sunt memorate în sistem, de obicei în fişiere pe disc. De exemplu, informaţia despre cărţile dintr-o bibliotecă publică.

Restricţiile

Acestea au de regulă influenţe asupra implementării sistemului. Exemple:

"Sistemul trebuie scris în limbajul de programare ADA", " Sistemul va răspunde utilizatorului în timp de o secundă". Principalele restricţii care pot apare sunt:

1). Costul
2). Termenul de livrare
3). Echipamentul hardware necesar
4). Capacitatea internă a memoriei
5). Capacitatea memoriei externe
6). Timpul de răspuns
7). Limbajul de programare care trebuie utilizat
8). Volumul de date (ex. sistemul trebuie să memoreze informaţii despre 10000 angajaţi)
9). Nivelul de încărcare pentru terminale pe unitatea de timp
10). Fiabilitatea solicitărilor (ex. sistemul trebuie să aibă un anumit timp mediu între defecţiuni pentru o perioadă de şase luni).

Restricţiile se adresează adesea implementării, de aceea trebuie făcute cu mare grijă.

Ghidul de aplicare

- furnizează informaţii utile pentru implementare, în situaţia în care există mai multe strategii de implementare.

De exemplu: “ Timpul de răspuns al sistemului la cererile sosite de la tastatură trebuie să fie minimizat. “ Sau o altă alternativă: “Utilizarea memoriei interne trebuie să fie cât mai redusă. “

deficienţe:

- imprecizia informaţiei. De exemplu: “ Interfaţa va fi prietenoasă.”
- contradicţiile: “ Rezultatele vor fi memorate pe suport magnetic” , sau, “ Sistemul va răspunde în mai puţin de o secundă. “
- omisiunea sau incompletitudinea : tratarea datelor de intrare eronate, furnizate de utilizator.

În concluzie, realizarea cu succes a specificaţiilor este o activitate necesară şi care solicită implicarea cu competenţă a unui mare număr de persoane.

2.3. Concepte ale specificaţiilor de programare

Orice program se exprimă în două forme:

- una fiind o mulţime obişnuită de comenzi şi structuri de control caracteristice limbajului de programare utilizat, care exprimă CUM se rezolvă problema;
- a doua este o serie de propoziţii (secvenţe) într-un formalism matematic, care exprimă CE face programul.

Presupunând că una dintre aceste forme este o reprezentare satisfăcătoare a intenţiilor programatorului, dacă se poate demonstra că ele exprimă acelaşi lucru, sau au aceeaşi semnificaţie, atunci se poate concluziona că programul este corect. Această abordare a verificării programului nu implică nici un alt concept de corectitudine absolută, ci doar demonstrează consistenţa celor două descrieri ale aceleiaşi probleme.

Atunci când se solicită descrierea semnificaţiei unei piese de software, de regulă se descrie interacţiunea programului cu mediul utilizatorilor. Exprimarea semnificaţiei programului în termenii utilizatorilor prin descrierea execuţiilor sale posibile se numeşte **descriere operaţională.** Această descriere nu este unică, o altă cale fiind descrierea în termenii entităţilor unui limbaj de programare.

Utilizarea unui formalism pentru descrierea programului permite verificarea mai multor implementări ale aceleaşi probleme.

Exprimarea semnificaţiei programului (ale specificaţiilor sale) într-un formalism al calculului cu predicate de ordinul întâi, chiar dacă raţionamentul este încă în termenii entităţilor utilizate de limbajul de programare, este efectiv independentă de orice concept al execuţiei comenzilor pe un calculator real sau abstract. Acest concept este important deoarece permite ca, în principiu să se exprime semnificaţia programului într-o astfel de manieră încât consistenţa mai multor implementări să fie verificată.

Se ştie că una dintre cele mai importante surse de erori în programe este generată de absenţa unei porţiuni din proiectul logic, deci un program trebuie exprimat (descris) în aşa fel încât să se reducă această sursă de erori.

2.3.1.Variabilele şi stările unui program

Dacă examinăm un program scris în orice limbaj de programare putem identifica două aspecte: unul este **controlul** execuţiei instrucţiunilor (if-then, etc.), al doilea este **execuţia** comenzilor (ca asignări de valori rezultate în urma unor evaluări).

Ne putem astfel imagina că pentru fiecare variabilă a programului există o stare care conţine informaţia curentă la orice moment de timp. Această stare este identificată de valoarea variabilei care poate fi schimbată de execuţia unei comenzi.

Să presupunem că fiecărei variabile a unui program îi asociem una din axe într-un spaţiu cartezian multidimensional, care se va numi **spaţiul stărilor unui program.** Un punct în acest spaţiu este o stare a programului. Execuţia unei comenzi elementare este vizualizată în acest spaţiu ca un segment ce leagă starea programului înainte de execuţie de cea de după execuţie. În acest context, **spaţiul stărilor** este o mulţime a tuturor valorilor posibile ale variabilelor programului.

Expresia condiţiilor care trebuie satisfăcute, în spaţiul stărilor, pentru o execuţie corectă a programului se numeşte **precondiţie.** Specificaţiile stărilor la terminarea programului se numesc **postcondiţii.**

Operaţiile dintr-un program ne permit să reorganizăm diferitele reprezentări ale aceluiaşi concept. Un tip de date este o mulţime de valori şi operaţii definite pe ea însăşi.

Exemple:

" mai mare decât " : G x G $\rightarrow$ {true, false}

>

"adunare" : G x G $\rightarrow$ G

+

În acest context, un sistem este o colecţie de mecanisme care realizează unul sau mai multe tipuri de date.

Tipurile de date pot fi definite prin **enumerare**, prin **operaţii** sau prin **proceduri.** Depinde de limbajul de programare modul în care acestea sunt implementate în compilatoare.

În conformitate cu Dijkstra programatorii pot utiliza trei modalităţi de proiectare a programelor: enumerarea, inducţia matematică şi abstractizarea.

Instrumentul mental al enumerării este principala metodă folosită de programatori. Utilizăm gândirea enumerativă ori de câte ori vrem să instruim pe cineva despre execuţia mai multor activităţi.

A doua metodă indicată de Dijkstra, inducţia matematică, este foarte utilizată pentru proiectarea structurilor repetitive (cicluri). De remarcat că, inducţia matematică nu este un proces mental natural ca enumerarea.

În general, principiile inducţiei pot fi definite astfel:

1. **Baza inducţiei** este de a stabili că un anumit element ***p*** al unei mulţimi ***S*** satisface o relaţie ***R***.
2. **Pasul inducţiei** este de a arăta că un element generic ***y*** a lui ***S*** satisface ***R***, adică există ***T(y)***, unde ***T*** este orice fel de transformare în care alt element a lui ***S*** satisface ***R***.
3. Dacă baza şi pasul inducţiei sunt demonstrate, atunci orice element derivând din ***p*** printr-un număr de aplicaţii ale lui ***T***, de asemenea satisfac ***R***.

Principiul inducţiei este fundamentarea teoretică a metodei de verificare propusă de Floyd. Importanţa acestui concept de bază este aceea că nu cere o execuţie mentală a unei secvenţe de comandă, permiţând proiectarea şi verificarea ciclurilor care trebuie executate de mai multe ori, în funcţie de valori particulare ale datelor.

Atunci când inducţia matematică este aplicată asupra stărilor descrise de relaţia R, această metodă este un instrument efectiv pentru proiectarea secvenţelor de iteraţii în general şi executarea lor de ori câte ori.

În proiectarea programelor, aşa cum am amintit deja, se utilizează şi limbajul logicii predicatelor. Fiecare predicat defineşte un subset al spaţiului stărilor pentru care el este adevărat.

Precondiţia unui program este un predicat care defineşte un subset al stărilor acceptat ca intrare legitimă. Constanta predicat T, care este adevărată peste tot, defineşte toate stările posibile ale spaţiului stărilor. Constanta predicat F, care este falsă peste tot, nu defineşte nici o stare în spaţiul stărilor.

Cu alte cuvinte, dacă spaţiul stărilor reprezintă întregul univers al obiectelor care pot fi manipulate de program, T defineşte în întregime acest univers , iar F este mulţimea vidă. Dacă un program a fost specificat cu o precondiţie care acceptă orice stare din spaţiul stărilor, T este în mod clar predicatul care reprezintă corect această condiţie.

Predicatele definesc submulţimi ale spaţiului stărilor. În acest context, verificarea corectitudinii programului în totalitate poate fi definită.

Dându-se un program S şi o precondiţie r, vom găsi "cea mai slabă precondiţie", adică condiţia care defineşte numai şi numai acele stări iniţiale care asigură terminarea în stările care satisfac r.

Această precondiţie, "cea mai slabă" este un predicat funcţie atât de S cât şi de r, indicat prin ***wp(S,r)***.

Bineînţeles că se poate defini şi problema duală. Dându-se o precondiţie p a unui program S, vom găsi "cea mai puternică" postcondiţie, adică predicatul care defineşte numai şi numai acele stări pentru care programul va satisface terminarea când este iniţializat într-o stare care satisface p.

Această "cea mai puternică" postcondiţie este o funcţie de S şi p definită ca ***Sp(S,p)***.

Conceptele "cea mai slabă" precondiţie şi "cea mai puternică" postcondiţie au fost introduse de Dijkstra în 1976. Acestea au fost necesare pentru a descrie tehnica de proiectare a lui Dijkstra numită **calculul programării.**

Ideea de bază este de a utiliza tehnici de demonstrare a corectitudinii **nu** prin verificarea programului deja creat printr-un mijloc sau altul , ci prin a ghida pas cu pas proiectarea programului în mod explicit şi conştient.

Calculul Dijkstra este direct îndreptat spre realizarea unei tehnici care, la sfârşitul programelor, să furnizeze verificarea proiectării lor printr-o analiză "intelectuală" înainte de testarea produsului.

Pe scurt, calculul Dijkstra constă în construirea de expresii scrise în limbajul calculului predicatelor care să descrie precondiţiile şi postcondiţiile problemei şi care să se verifice ca adevărate în contextul problemei. Puterea acestui calcul este evidentă în cazul problemelor greu de algoritmat.

O tehnică destul de comună în programare este abstractizarea procedurală. Ea constă din abstractizarea unei piese din program (de regulă nescrisă încă) prin substituirea în textul său a precondiţiei şi postcondiţiei. În acest fel întreaga structură a programului poate fi definită şi se demonstrează corectitudinea lui prin utilizarea unui text mai scurt.

Această tehnică poate fi efectiv aplicată pentru algoritmi a căror complexitate şi dimensiune nu sunt prea mari. Nu este de imaginat să se aplice această metodă pentru proiectarea unui sistem software mare. Totuşi, este posibil de conceput că fiecare sistem mare este compus din mai multe părţi componente, fiecare dintre acestea nefiind prea mare, care poate fi proiectată folosind abstractizarea procedurală.

Problemele specifice limbajelor de programare, cum ar fi pointerii şi transmiterile de parametri între module, limitează aplicabilitatea calculului Dijkstra. Totuşi, exprimarea specificaţiilor programelor în termenii limbajului calculului predicativ ajută la reducerea erorilor.

2.4. Specificarea formală

Obiectivul acestui capitol este de a înţelege şi scrie specificaţiile unui sistem software. S-a amintit deja că ieşirile din activitatea de analiză sunt constituite din specificaţii de proiectare, care conţin atât solicitări funcţionale cât şi solicitări de date, cu alte cuvinte se detaliază cu precizie ceea ce sistemul va executa având în vedere restricţiile practice de care proiectantul va ţine cont. Ne vom referi la componenta funcţională a specificaţiilor de proiectare ca la o *specificaţie software* .

Comportarea unui sistem este influenţată de dezvoltarea lui în cele trei faze, şi anume: specificarea, implementarea şi verificarea. O abordare convenţională este specificarea sistemului în limbaj natural (în limba româna), scrierea programului într-un limbaj de programare (ex. Pascal) şi verificarea prin testare. S-a argumentat că o astfel de abordare este susceptibilă la tot felul de erori şi în ultimă instanţă generează software incorect.

Utilizând abordarea convenţională se pune întrebarea: cum şi de ce au fost scrise programe incorecte? Există trei cauze esenţiale:

1. Specificarea este greşită - ea poate fi ambiguă, vagă, inconsistentă şi/sau incompletă.

2. Programul este greşit - nu execută ceea ce există în specificaţii.

3. Verificarea este incompletă - testarea nu este exhaustivă, mai mult decât atât, nici nu poate fi exhaustivă .

Există următoarele aserţiuni pentru cele trei faze importante din dezvoltarea software-ului :

1). Specificarea este o *descriere* - adică spune CE reprezintă problema;

2). Problema este o *realizare* a specificaţiei - adică spune CUM poate fi rezolvată problema;

3). Verificarea este o *justificare* - spune DE CE realizarea satisface specificaţia.

Specificarea într-o notaţie formală, mai mult decât într-un limbaj natural, aduce beneficii imediate. "Formal" înseamnă scrierea în întregime într-un limbaj cu o sintaxă explicită şi precisă şi cu o semantică definită. Limbajul matematic este mai apropiat pentru acest scop.

Avantajele utilizării unei notaţii formale pot fi rezumate astfel:

- Specificaţiile formale pot fi studiate matematic - cu alte cuvinte, o specificaţie poate fi judecată utilizând tehnicile matematice. De exemplu, diverse forme de inconsistenţă sau incompletitudine în specificaţii pot fi detectate automat.
- Specificaţiile formale pot fi întreţinute cu mai multă uşurinţă decât dacă ar fi scrise în limbaj natural. Aceasta face ca specificaţiile să prezinte mai multă siguranţă şi să fie asigurat controlul posibilelor consecinţe ale schimbărilor.
- Notaţia utilizată pentru exprimarea specificaţiilor formale este extensibilă.

Utilizând un limbaj de specificare formală avem posibilitatea de a mecaniza transformarea specificaţiilor în programe. Prototipizarea sistemelor (în general ineficientă) poate fi generată automat din specificaţii mai adecvate pentru viabilitatea sistemului propus.

Şi mai important este că în acest fel, avem posibilitatea potenţială de a dovedi corectitudinea programului în raport cu specificaţiile sale.

2.5. Exemplu de specificare formală

În mod curent, cele mai cunoscute limbaje de specificare formală sunt Z şi VDM. Ambele utilizează o abordare pe bază de model, adică reprezintă tipurile de date şi structurile necesare pentru a putea descrie problema folosind entităţi cum ar fi: mulţimi, funcţii şi propoziţii.

Limbajul Z a fost prima dată introdus de Jean-Raymond Abriel în 1979 şi dezvoltat în cadrul Programming Research Group la Universitatea Oxford. Partea "puternică" a lui Z este aceea că specificaţiile structurate pot fi achiziţionate folosind "schema de calcul" ; aceasta permite specificaţiilor să fie construite în trei blocuri mai mici, iar programele pot fi proiectate top-down sau bottom-up, pentru componentele lor procedurale.

Metoda VDM (Vienna Development Method) a fost iniţiată în laboratoarele de cercetare IBM din Viena, în 1970. Această metodă se bazează pe notaţiile semantice, o abordare a definirii limbajelor de programare făcută de Scott şi Strachey. VDM reprezintă mai mult decât un limbaj semantic, el punând la dispoziţia celor care îl utilizează reguli şi proceduri care vor fi urmărite în diferite stadii de dezvoltare a sistemului.

Z şi VDM au notaţii vaste şi complexe. Este interesant poate de subliniat că un set relativ mic de instrumente permite specificarea cu uşurinţă a unei clase largi de probleme tipice.

Pentru a putea înţelege mai bine exemplul următor, să reluăm câteva notaţii matematice cum ar fi: mulţimile, secvenţele şi funcţiile.

O *mulţime* este o colecţie de obiecte, exprimată de obicei prin enumerarea elementelor incluse între acolade. De exemplu:

{Anton, Vasile, Ioana, Alice}

este o mulţime de nume. Se poate utiliza semnul "$\in$" pentru a indica apartenenţa unui element la o mulţime.

Astfel Anton $\in$ {Anton, Vasile, Ioana, Alice}, dar

Sandu $\notin$ {Anton, Vasile, Ioana, Alice}.

O *secvenţă* este o colecţie ordonată de obiecte exprimate în mod normal prin enumerarea elementelor sale, incluse în paranteze drepte. De exemplu:

[Anton, Vasile, Ioana, Alice]

este o secvenţă de nume. *Capătul (capul)* secvenţei este Anton, iar *corpul* este [Vasile, Ioana, Alice]. Dacă S este o secvenţă, atunci elementele lui S marchează mulţimea de elemente din secvenţa S. Dacă S este o secvenţă a mulţimii, atunci elemente lui S = {Anton, Vasile, Ioana, Alice}. O secvenţă poate avea elemente care se repetă, în timp ce o mulţime, nu.

O *funcţie* exprimă o relaţie între două elemente ale unei mulţimi. De exemplu, *director* poate asocia nume cu numere de telefon. Dacă vom utiliza *Nume* pentru a reprezenta mulţimea tuturor numelor posibile şi *Număr* pentru a reprezenta mulţimea tuturor numerelor de telefon posibile, atunci putem descrie *director* astfel:

director : Nume →Număr

Vom spune că *director* este o funcţie de tip *Nume* → *Număr* . Orice funcţie de acest tip va avea ca argument un membru din *Nume* şi va returna un membru din mulţimea *Număr.* Totuşi, *directorul* nostru particular este parţial, în sensul că este definit numai pentru unele elemente din *Nume* deoarece nu toate persoanele au telefon.

Submulţimea (subsetul) pe care este definită funcţia se numeşte *domeniul de definiţie al funcţiei.* De exemplu, să presupunem că *Nume* este mulţimea {Anton, Vasile, Ioana, Alice} şi *Număr* este mulţimea {421563, 123456}. În acest caz *directorul* va fi definit explicit astfel:

director(Anton) = 421563

director(Alice) = 123456

Dacă Vasile şi Ioana nu au telefon atunci *director (Vasile)* şi *director (Ioana)* nu sunt definite. Domeniul funcţiei este mulţimea {Anton, Alice}, care este un subset al mulţimii *Nume* .

Funcţia *director* mai poate fi definită prin explicitarea enumerativă a *Numelui* legat de un *Număr,* astfel:

director = {Anton $\rightarrow$ 421563, Alice $\rightarrow$ 123456}

Anton $\rightarrow$ 421563 şi Alice $\rightarrow$ 123456 se numesc "corespondenţe" (maplets). *Director* este o funcţie finită deoarece domeniul său este finit, cu alte cuvinte conţine un număr finit de corespondenţe.

Funcţiile exprimă mai mult decât o relaţie. Astfel, dacă luăm ca model funcţia telefon, mai multe persoane pot avea acelaşi număr de telefon, dar o anumită persoană nu poate avea decât un singur număr de telefon.

Să considerăm acum următorul exemplu pentru a descrie specificaţiile formale simple pentru o problemă descrisă informal astfel :

O bancă are numai un ghişeu unde în mod normal clienţii aşteaptă la coadă. Fiecare client se identifică prin numărul de cont şi sunt permise numai tranzacţii de adăugare sau scoatere din cont. Clientului i se refuză eliberarea sumei cerute dacă contul său este vid. Dacă el doreşte mai mult decât are în cont atunci va primi numai suma pe care o mai are în cont. După ce s-a efectuat tranzacţia, clientul părăseşte banca.

Primul lucru care trebuie făcut este să se stabilească un model matematic adecvat pentru sistemul propus. Conturile din bancă pot fi modelate printr-o funcţie parţială care face corespondenţa între numerele de cont şi balanţa băncii:

bancă : Cont $\rightarrow$ *Balanţă*

Cont este mulţimea de numere de conturi posibile, iar *Balanţă* este mulţimea tuturor balanţelor (disponibilului) băncii. Deci o funcţie *bancă* poate arăta astfel :

{a1$\rightarrow$ 23, a2 $\rightarrow$ 87, a3 $\rightarrow$ 45},

care indică de exemplu, că persoana care are numărul de cont a1 are 45000 lei în cont, ş.a.m.d. Domeniul funcţiei *bancă* (scris dom_bancă) este o submulţime {a1,a2,a3}, care identifică conturile.

Coada la ghişeul băncii poate fi modelată ca o secvenţă de numere de cont astfel:

coadă : secvCont

În acest fel [a2,a5,a1] reprezintă o coadă tipică. Vom adopta convenţia că prima persoană din linie reprezintă *capul* cozii. De notat că putem avea elemente care se repetă, adică o persoană poate să apară de mai multe ori în coadă. Ne vom asigura că acest lucru nu este posibil, introducând restricţii suplimentare ori de câte ori se schimbă sau actualizează coada.

Avem acum specificarea formală a operaţiilor care se vor efectua în sistemul modelat. Fiecare operaţie va fi specificată prin trei componente:

1. Intrările operaţiei;

2. Condiţia în care operaţia poate fi aplicată (*precondiţia*);

3. Expresia care arată relaţia dintre starea sistemului înainte de operaţie şi starea sistemului după operaţie (*post-condiţia*).

Vom adopta convenţia de a folosi numele obişnuit pentru a desemna starea iniţială şi &nume, pentru starea finală. Exemplu : coadă reprezintă starea cozii înainte de sosirea unui client, iar &coadă, după.

Să luăm acum specificaţia care introduce un client în coadă:

sosire (client : Cont)

precondiţie

client $\in$ *dom_bancă*

client $\notin$ *elem_coadă*

post-condiţie

&coadă = adaug_ coadă [client]

Numele operaţiei este *sosire.* Ea are o intrare numită *client*, de tip *Cont.* Precondiţia indică faptul că atunci când un client soseşte la coadă el trebuie să aibă cont în bancă, altfel nu va fi acceptat în coada de aşteptare. Post-condiţia indică faptul că după operaţie coada se măreşte cu o persoană. (Observaţie: *adaug* este un operator de concatenare a două secvenţe).

De exemplu, să presupunem:

bancă = {a1→ 23, a2→ 87, a3 → 45}

coadă = [a2]

client = [a1]

atunci

client ∈ {a1,a2,a3}

client ∉ {a2}

şi

&*coadă* = [a2,a1]

Vom scrie acum o operaţie care acordă credit unui client care are cont şi se află în capul cozii (este primul în coadă):

credit (*suma* : *Balanţă*)

precondiţia

coadă ≠ [ ]

post-condiţia

&bancă = *bancă* ⊕ {*cap coadă* → *bancă(cap coadă*) + *suma*}

&coadă = elimin coadă

Operaţia *credit* are o intrare numită ***sumă***, de tip ***Balanţă*** şi o precondiţie care precizează că lista (coada) nu trebuie să fie vidă, adică trebuie să fie măcar o persoană la coadă.

⊕ este un operator de scriere adiţională, peste ceea ce era înainte. El creează o nouă funcţie din alte două date ca operanzi. De exemplu, să presupunem că:

bancă = {a1 → 23, a2 → 87, a3 → 45}

coadă = [a2,a1]

suma = 10

Atunci

capul cozii = a2

bancă(*capul cozii*) = 87

astfel încât

&bancă = {a1 → 23, a2 → 87, a3 → 45} ⊕ {a2 → 87+10} = {a1 → 23, a2 →97, a3 →45}

Corespondenţa care are a2 în stânga a fost scrisă peste vechea corespondenţă, definind astfel o nouă funcţie care modelează noua stare a băncii.

Ultima componentă a post-condiţiei arată că după efectuarea tranzacţiei, clientul din faţă părăseşte coada :

&*coadă* = *elimin* [a2 a1]

= [a1]

În final, vom defini operaţia care permite efectuarea retragerii de bani din cont :

debit (*suma* : *Balanţă*)

precondiţie

coadă $\neq$ [ ]

bancă (*capul cozii*) $\geq$ *suma*

post-condiţia

&*bancă* = *bancă* $\oplus$ { *capul cozii* $\rightarrow$ *bancă*(*capul cozii*) - *suma*}

&*coadă* = *elimin coadă*

Prima componentă din precondiţie arată mai întâi că nu trebuie să fie vidă coada şi apoi presupune că:

bancă = {a1 $\rightarrow$ 23, a2 $\rightarrow$ 87, a3 $\rightarrow$ 45}

coadă = [a2,a1]

suma = 10

Capul cozii este a2 iar *bancă* (*capul cozii*) = 87

A doua componentă a precondiţiei specifică faptul că balanţa relativă la contul clientului trebuie să fie mai mare sau egală cu suma pe care acesta intenţionează s-o scoată din bancă.

În contextul validării specificaţiilor (ceea ce înseamnă de fapt înglobarea cerinţelor), vom specifica acum un *invariant* .

Un *invariant* este o aserţiune despre componentele modelului matematic, pe care se vor baza apoi specificaţiile noastre. Dacă invariantul se află înainte de operaţiile pe care le-am specificat, atunci el trebuie să apară şi după ce operaţia este completă.

El exprimă ceva ce este întotdeauna adevărat. De exemplu, există un invariant pentru bancă:

($\forall$ a $\in$ dom_bancă) *bancă*(a) $\geq$ 0

elem_coadă $\subseteq$ dom_bancă

coadă $\leq$ 10

Acest invariant stipulează că :

1. Oricare ar fi *a* din domeniul băncii, valoarea lui *bancă*(a) este mai mare sau egală cu zero. Aceasta înseamnă că toţi clienţii băncii au credit.

2. Mulţimea de persoane aflată în coadă este o submulţime a domeniului băncii. Aceasta înseamnă că fiecare persoană din coadă are un cont în bancă.

3. Numărul de elemente din coadă este mai mic sau egal cu 10, considerând că nu vor fi niciodată mai mult de 10 persoane la coadă.

Intuitiv, se poate vedea că debitul şi creditul conservă acest invariant, dar *sosire* nu, de aceea nu a existat nici o restricţie cu privire la lungimea cozii. Totuşi *sosire* va conserva invariantul dacă vom adăuga un extra predicat : # *coadă* < 10

în precondiţia sa.

Invarianţii furnizează un instrument de valoare pentru asigurarea consistenţei peste operaţiile care se fac în cadrul specificaţiilor.

Concluzii:

Rezumând, un mediu ideal de inginerie software (cu sublinierea cuvântului inginerie), ca scenariu tipic pentru dezvoltarea unui program, trebuie să arate astfel:

- Să existe o specificare formală, derivată dintr-o descriere informală, care să ajute mai bine la înţelegerea problemei.
- Să existe o dovadă care să demonstreze că specificarea reală se poate realiza printr-un algoritm.
- Să existe un prototip (chiar dacă este ineficient) care să poată fi prezentat celui care trebuie să valideze proiectul.
- Să se scrie programul sau programele derivate din specificaţii.
- Să existe o dovadă că programul este corect prin prisma specificaţiilor sale.

Dezvoltarea unor instrumente integrate de software care să susţină fiecare dintre aceste faze este considerată vitală pentru acceptarea unui astfel de scenariu.

2.6. Exerciţii

1) Care sunt informaţiile necesare care trebuie colectate şi înregistrate ca specificaţii software?

2) Explicaţi dificultăţile limbajului natural pentru descrierea specificaţiilor software?

3) Luaţi ca exemplu un sistem informatic mic (care doriţi dv.). Identificaţi componentele funcţionale şi de date din sistem. Identificaţi problemele legate de specificaţiile software cum ar fi: ambiguitate, inconsistenţă şi informaţii vagi.

4) Există solicitarea pentru un sistem informatic care să gestioneze informaţiile despre cărţile existente într-o mică bibliotecă de departament.

s1 - sistemul trebuie să funcţioneze pe un calculator standard PC;

s2 - pentru fiecare carte informaţiile standard sunt:

- titlul
- autorul
- codul de clasificare (cota cărţii)
- anul apariţiei
- dacă este împrumutată
- data solicitării

s3 - calculatorul trebuie să memoreze informaţiile pentru 1000 de cărţi

s4 - sistemul trebuie să răspundă la următoarele comenzi de la tastatură:

(a) solicitarea de împrumut a unei cărţi

(b) înapoierea unei cărţi împrumutate

(c) crearea unei înregistrări pentru o carte nou achiziţionată;

s5 - comenzile vor fi accesibile printr-o selecţie cu ajutorul cursorului dintr-un meniu;

s6 - calculatorul trebuie să răspundă în maximum 30 de secunde de la formularea cererii;

s7 - calculatorul trebuie să fie capabil să tipărească întregul catalog al cărţilor, cu un cap de tabel corespunzător. Acest lucru trebuie să se facă în paralel cu altă comandă primită;

s8 - măsuri de securitate: sistemul va iniţializa informaţiile din bibliotecă numai dacă el conţine zero cărţi;

s9 - când o carte este ştearsă sistemul va afişa informaţiile despre ea;

s10 - sistemul trebuie livrat la data D, trebuie să nu depăşească costul C, să fie în întregime documentat şi uşor de întreţinut.

5. " Scrieţi un program de sortare ". Este această modalitate, adecvată ca specificaţie ?

6. Un aspect necesar al specificaţiilor este discuţia cu potenţialii utilizatori. Cei mai mulţi dintre ei nu înţeleg notaţiile matematice. Este acesta un dezavantaj? Ce aduc în plus notaţiile formale pentru a face această muncă mai productivă?

7. Enumeraţi câteva cerinţe non-funcţionale din implementarea programului de împrumut la bancă.

8. Scrieţi specificaţiile pentru deschiderea şi închiderea unui cont bancar.

9. Modificaţi invariantul pentru bancă astfel încât să ilustreze faptul că o persoană nu poate să apară de mai multe ori în coadă.

10. Pentru cine sunt mai importante atributele unei specificaţii, pentru programator sau pentru cel care implementează programul ?

Paradigmele de dezvoltare a sistemelor de programe

3.1. Etapele dezvoltării programelor

Când pornim la dezvoltarea unui program avem nevoie de:

- înţelegere clară a ceea ce se cere;
- un set de metode şi instrumente de lucru;
- un plan de acţiune.

Planul de acţiune se numeşte ***metodologie de dezvoltare***.

- Dezvoltarea unui anumit program constă într-un set de paşi ce se fac pentru a-l realiza.
- Luând în considerare tipul paşilor ce se efectuează, se creează un model de lucru, ce poate fi aplicat unei serii mai largi de proiecte.
- Acesta este motivul pentru care planul de acţiune este numit **model**: el poate fi privit ca un şablon al dezvoltării de programe.
- În timpul dezvoltării programelor s-a constatat că există anumite tipuri de activităţi care trebuie făcute la un moment dat:

- ***Analiza cerinţelor**:* Se stabileşte ce anume vrea clientul ca programul să facă.
  - înregistrarea cerinţelor într-o manieră cât mai clară (lipsa ambiguităţilor) şi mai fidelă (cuvânt cu cuvânt);
- ***Proiectarea arhitecturală**:* - împarte sistemul într-un număr de module mai mici şi mai simple, care pot fi abordate individual;
- ***Proiectarea detaliată**:* Se realizează proiectarea fiecărui modul al aplicaţiei, în cele mai mici detalii;
- ***Scrierea codului**:* Proiectul detaliat este transpus într-un limbaj de programare. De obicei, aceasta se realizează modular, pe structura rezultată la proiectarea arhitecturală;
- ***Integrarea componentelor**:* Modulele programului sunt combinate în produsul final. Rezultatul este sistemul complet.
  - În modelul numit ***big-bang*** componentele sunt dezvoltate şi testate individual, după care sunt integrate în sistemul final.

- Chiar dacă componentele aufost testate individual şi funcţionează, la asamblare apar erori sau conflicte între anumite componente (de exemplu, conflicte de partajare a resurselor).
- Astfel timpul de testare explodează, proiectul devenind greu de controlat; aceasta justifică denumirea de „big-bang”.

▪ ***Modelul incremental*** propune crearea unui nucleu al aplicaţiei şi integrarea a câte o componentă la un moment dat, urmată imediat de testarea sistemului obţinut. Astfel, se poate determina mai uşor unde anume apare o problema în sistem. Acest tip de integrare oferă de obicei rezultate mai bune decât modelul big-bang;

- ***Acceptarea****:* În procesul de acceptare ne asigurăm că programul îndeplineşte cerinţele utilizatorului. Întrebarea la care răspundem este: construim produsul corect?
  - ▪ Clientul spune dacă este mulţumit cu produsul sau dacă mai trebuie efectuate modificări;
- ***Verificarea****:* ne asigurăm că programul este stabil şi că funcţionează corect din punctul de vedere al dezvoltatorilor. Întrebarea la care răspundem este: construim corect produsul?
- ***Întreţinerea****:* gestionarea : erorilor, schimbarea specificaţiilor, îmbunătăţiri.

Se poate constata uşor că aceste activităţi sunt în strânsă legătură cu cele patru faze ale ingineriei programării: analiza, proiectarea, implementarea şi testarea.

3.2. Paradigmele de dezvoltare software

Paradigmele de dezvoltare a sistemelor de programe se constituie din două categorii de metodologii şi anume:

3.2.1. Metodologii generice	**3.2.2. Metodologii concrete.**
▪ Metodologia secvenţială ▪ Metodologia ciclică ▪ Metodologia hibridă ecluză	▪ Metodologia cascadă ▪ Metodologia spirală ▪ Metodologia spirală WinWin ▪ Prototipizarea ▪ Metodologia Booch ▪ Metode formale ▪ Metoda V ▪ Programarea extremă ▪ Metodologia Open Source ▪ Reverse Engineering ▪ Metodologia de dezvoltare Offshore ▪ Metodologii orientate obiect

3.2.1. Metodologii generice

3.2.1.1. Metodologia secvenţială

În metodologia secvenţială (fig.3.1), cunoscută şi sub numele de metodologia „cascadă”, are loc mai întâi faza de analiză, apoi cea de proiectare, urmată de cea de implementare, iar în final se realizează testarea. Echipele care se ocupă de fiecare fază pot fi diferite, iar la fiecare tranziţie de fază poate fi necesară o decizie managerială.

Fig. 3.1. Metodologia secvenţială

Avantaje

- este potrivită când complexitatea sistemului este mică iar cerinţele sunt statice.

- mai întâi trebuie să ne gândim ce trebuie construit, apoi să stabilim un plan de lucru şi apoi să realizăm proiectul, ţinând cont de standardele de calitate.

- se aliniază metodelor de inginerie hardware. Forţează menţinerea unei discipline de lucru care evită presiunea scrierii codului înainte de a cunoaşte precis ce produs va trebui de fapt construit.

De multe ori, echipa de implementare se află în situaţia de a programa înainte de finalizarea analizei, ceea ce conduce inevitabil la descoperirea unor părţi de cod inutile sau care contribuie foarte puţin (poate chiar şi ineficient) la funcţionalitatea produsului final. Totuşi, acest cod devine un balast foarte costisitor: dificil de abandonat şi greu de schimbat. Această metodologie forţează analiza şi planificarea înaintea implementării, o practică foarte nimerită în multe situaţii.

Un mare număr de sisteme software din trecut au fost construite cu o metodologie secvenţială. Multe companii îşi datorează succesul acestui mod de realizare a programelor. Trebuie spus totuşi şi că presiunea de schimbare din partea surselor externe era destul de limitată la momentul respectiv.

Dezavantaje

- acordă o foarte mare importanţă fazei de analiză.

- Membrii echipei de analiză ar trebui să fie probabil clarvăzători ca să poată defini *toate* detaliile aplicaţiei încă de la început.

- Greşelile nu sunt permise, deoarece nu există un proces de corectare a erorilor după lansarea cerinţelor finale.

- Nu există nici feedback de la echipa de implementare în ceea ce priveşte complexitatea codului corespunzător unei anumite cerinţe.

- O cerinţă simplu de formulat poate creşte considerabil complexitatea implementării.

- În unele cazuri, este posibil să fie chiar imposibil de implementat cu tehnologia actuală.

- Dacă echipa de analiză ar şti că o cerinţă nu poate fi implementată, ei ar putea-o schimba cu o cerinţă diferită care să satisfacă cele mai multe dintre necesităţi şi care să fie mai uşor de efectuat.

Comunicarea dintre echipe este o problemă: cele patru echipe pot fi diferite iar comunicarea dintre ele este limitată.

-- Modul principal de comunicare sunt documentele realizate de o echipă şi trimise următoarei echipe cu foarte puţin feedback.

- Echipa de analiză nu poate avea toate informaţiile privitoare la calitate, performanţă şi motivare.

Într-o industrie în continuă mişcare, metodologia secvenţială poate produce sisteme care, la vremea lansării, să fie deja învechite.

Accentul atât de mare pus pe planificare nu poate determina răspunsuri suficient de rapide la schimbare.

Ce se întâmplă dacă clientul îşi schimbă cerinţele după terminarea fazei de analiză?

Acest lucru se întâmplă însă frecvent; după ce clientul vede prototipul produsului, el îşi poate schimba unele cerinţe.

3.2.1.2. Metodologia ciclică

Metodologia ciclică (fig.3.2), cunoscută şi sub numele de metodologia „***spirală***”, încearcă să rezolve unele din problemele metodologiei secvenţiale.

Şi această metodologie are patru faze, însă în fiecare fază se consumă un timp mai scurt, după care urmează mai multe iteraţii prin toate fazele.

Ideea este de fapt următoarea: gândeşte un pic, planifică un pic, implementează un pic, testează un pic şi apoi ia-o de la capăt.

În mod ideal, fiecărei faze trebuie să i se acorde atenţie şi importanţă egale.

Documentele de la fiecare fază îşi schimbă treptat structura şi conţinutul, la fiecare ciclu sau iteraţie.

Pe măsură ce procesul înaintează, sunt generate din ce în ce mai multe detalii.

În final, după câteva cicluri, sistemul este complet şi gata de lansare.

Procesul poate însă continua pentru lansarea mai multor versiuni ale produsului.

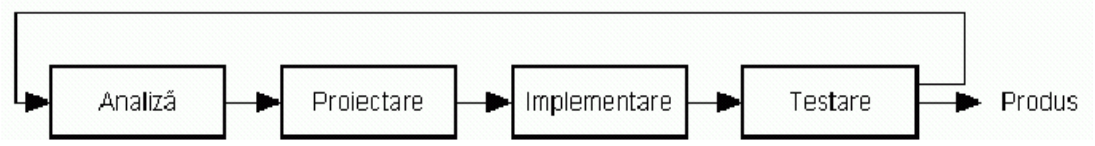


Fig.3.2. Metodologia ciclică

Avantaje

Metodologia ciclică se bazează pe ideea perfecţionării incrementale ale metodologiei secvenţiale.

Permite feedback-ul de la fiecare echipă în ceea ce priveşte complexitatea cerinţelor.

Există etape în care pot fi corectate eventualele greşeli privind cerinţele.

Clientul poate arunca o privire asupra rezultatului şi poate oferi informaţii importante mai ales în faza dinaintea lansării produsului.

Echipa de implementare poate trimite echipei de analiză informaţii privind performanţele şi viabilitatea sistemului.

Acesta se poate adapta mai bine progresului tehnologic: pe măsură ce apar noi soluţii, ele pot fi încorporate în arhitectura produsului.

Dezavantaje

Metodologia ciclică nu are nici o modalitate de supraveghere care să controleze oscilaţiile de la un ciclu la altul.

În această situaţie, fiecare ciclu produce un efort mai mare de muncă pentru ciclul următor, ceea ce încarcă orarul planificat şi poate duce la eliminarea unor funcţii sau la o calitate scăzută.

Lungimea sau numărul de cicluri poate creşte foarte mult.

De vreme ce nu există constrângeri asupra echipei de analiză să facă lucrurile cum trebuie de prima dată, acest fapt duce la scăderea responsabilităţii.

Echipa de implementare poate primi sarcini la care ulterior se va renunţa.

Echipa de proiectare nu are o viziune globală asupra produsului şi deci nu poate realiza o arhitectură completă.

Nu există termene limită precise.

Ciclurile continuă fără o condiţie clară de terminare.

Echipa de implementare poate fi pusă în situaţia nedorită în care arhitectura şi cerinţele sistemului sunt în permanenţă schimbare.

3.2.1.3. Metodologia hibridă ecluză

Metodologia ecluză, propusă de Ronald LeRoi Burback (1998), separă aspectele cele mai importante ale procesului de dezvoltare a unui produs software de detaliile mai puţin semnificative şi se concentrează pe rezolvarea primelor.

- Pe măsură ce procesul continuă, detaliile din ce în ce mai fine sunt rafinate, până când produsul poate fi lansat.

- Această metodologie hibridă (fig.3.3) preia natura iterativă a metodologiei spirală, la care adaugă progresul sigur al metodologiei cascadă.

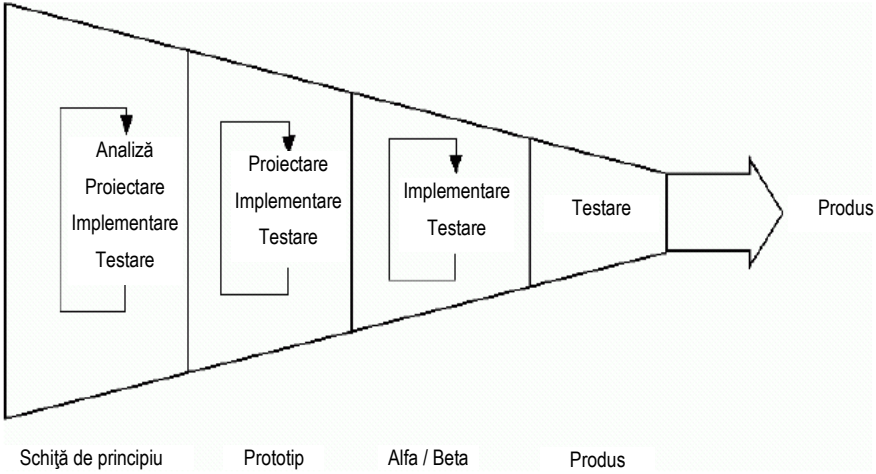


Fig. 3.3. Metodologia hibridă ecluză

- La început, într-un proces iterativ, fazele de analiză, proiectare, implementare şi testare sunt împărţite în mai multe sarcini potenţiale, fiecăruia atribuindu-i-se o prioritate.

- La fiecare moment se execută sarcina cu prioritate maximă.

- În funcţie de dimensiunea echipelor, mai multe sarcini pot fi realizate în paralel.

- Sarcinile rămase, de prioritate minimă, sunt păstrate pentru examinare ulterioară.

- Descompunerea problemei este foarte importantă.

- Cu cât descompunerea şi stabilirea priorităţilor sunt mai bune, cu atât mai eficientă este metodologia.

- Priorităţile se stabilesc pe baza unei *funcţii de prioritate*, care depinde atât de domeniul problemei şi cât şi de normele firmei.

- Ea trebuie să realizeze un compromis între cantitate şi calitate, între funcţionalitate şi constrângerile privind resursele, între aşteptări şi realitate.

-Toate funcţiile de prioritate ar trebuie să aibă ca prim scop lansarea produsului.

- funcţia mai trebuie să gestioneze sarcinile conflictuale şi nemonotone.

- Odată ce o componentă este terminată şi acceptată de echipă, schimbările asupra sa sunt îngheţate.

- Componenta va fi schimbată numai dacă modificările sunt absolut necesare iar echipa este dispusă să întârzie lucrul la restul sistemului pentru a le efectua.

- Schimbările trebuie să fie puţine la număr, bine justificate şi documentate.

Etapele principale ale metodei sunt:

- schiţa de principiu,
- prototipul,
- versiunile alfa/beta şi
- produsul final.

În prima etapă, ***schiţa de principiu***, echipele lucrează simultan la toate fazele problemei.

- Echipa de analiză sugerează cerinţele.

- Echipa de proiectare le discută şi trimite sarcinile critice de implementare echipei de implementare.

- Echipa de testare pregăteşte şi dezvoltă mediul de test în funcţie de cerinţe.

- Echipa de implementare se concentrează asupra sarcinilor critice, care în general sunt cele mai dificile. - contrastează cu practica curentă de realizare mai întâi a sarcinilor simple.

Dacă nu sunt respectate cu stricteţe etapele sistemele pot să eşueze.

Odată ce *componentele critice* au fost realizate, sistemul este gata de a face tranziţia către ***stadiul de prototip***.

Unul din scopurile aceste etapei este de a se convinge echipele că o soluţie poate fi găsită şi pusă în practică.

În cea de a doua etapă, de ***prototip***, cerinţele şi documentul cerinţelor sunt îngheţate.

- Schimbările în cerinţe sunt încă permise, însă ar trebuie să fie foarte rare şi numai dacă sunt absolut necesare, deoarece modificările cerinţelor în acest stadiu al proiectului sunt foarte costisitoare.

- Este posibilă totuşi ajustarea arhitecturii, pe baza noilor opţiuni datorate tehnologiei.

- După ce sarcinile critice au fost terminate, echipa de implementare se poate concentra pe extinderea acestora, pentru definirea cât mai multor aspecte ale aplicaţiei.

- Unul din scopurile acestei etape este de a convinge persoanele din afara echipelor că o soluţie este posibilă.

Acum produsul este gata pentru lansarea versiunilor ***alfa şi beta***.

- Arhitectura este îngheţată, iar accentul cade pe implementare şi asigurarea calităţii.

- Prima versiune lansată se numeşte în general *alfa*.

Produsul este încă imatur; numai sarcinile critice au fost implementate la calitate ridicată.

Numai un număr mic de clienţi sunt în general dispuşi să accepte o versiune alfa şi să-şi asume riscurile asociate.

O a doua lansare reprezintă versiunea *beta*.

Rolul său este de a convinge clienţii că aplicaţia va fi un produs adevărat şi de aceea se adresează unui număr mai mare de clienţi.

Când o parte suficient de mare din sistem a fost construită, poate fi lansat în sfârşit ***produsul***.

În această etapă, implementarea este îngheţată şi accentul cade pe asigurarea calităţii.

Scopul este realizarea unui produs competitiv.

În produsul final nu se acceptă erori critice.

Avantaje:

- Metodologia ecluză recunoaşte faptul că oamenii fac greşeli şi că nici o decizie nu trebuie să fie absolută.

- Echipele nu sunt blocate într-o serie de cerinţe sau într-o arhitectură imobilă care se pot dovedi mai târziu inadecvate sau chiar greşite.

- Totuşi, metodologia impune date de îngheţare a unor faze.

- Există timp suficient pentru corectarea greşelilor decizionale pentru atingerea unui nivel suficient de ridicat de încredere.

- Se pune mare accent pe comunicarea între echipe, ceea ce reduce cantitatea de cod inutil la care ar trebui să se renunţe în mod normal.

- Metodologia încearcă să mute toate erorile la începutul procesului, unde corectarea, sau chiar reînceperea de la zero a lucrului, nu sunt foarte costisitoare.

Dezavantaje

Metodologia presupune asumarea unor responsabilităţi privind delimitarea etapelor şi îngheţarea succesivă a fazelor de dezvoltare.

Ea presupune crearea unui mediu de lucru în care acceptarea responsabilităţii pentru o decizie care se dovedeşte mai târziu greşită să nu se repercuteze în mod negativ asupra individului.

Se doreşte de asemenea schimbarea atitudinii echipelor faţă de testare, care are loc încă de la început, şi faţă de comunicarea continuă, care poate fi dificilă, întrucât cele patru faze reprezintă perspective diferite asupra realizării produsului.

3.2.2. Metodologii concrete

3.2.2.1. Metodologia cascadă

Metodologia cascadă, propusă de Barry Boehm, este una din cele mai cunoscute exemple de metodologie de ingineria programării.

Există numeroase variante ale acestui proces.

Într-o variantă detaliată, metodologia cascadă cuprinde etapele prezentate în fig.3.4.

După fiecare etapă există un pas de acceptare.

Procesul „curge” de la etapă la etapă, ca apa într-o cascadă.

În descrierea originară a lui Boehm, există o întoarcere, un pas înapoi interactiv între fiecare două etape.

Astfel, metoda cascadă este de fapt o combinaţie de metodologie secvenţială cu elemente ciclice.

Totuşi, în practica inginerească, termenul „cascadă” este utilizat ca un nume generic pentru orice metodologie secvenţială.

Acesta este modelul după care de obicei sistemele sunt dezvoltate în practică.

Există o mare atracţie pentru acest model datorită experienţei, tradiţiei în aplicarea sa şi succesului pe care l-a implicat.

O sarcină complexă este împărţită în mai mulţi paşi mici, ce sunt mai uşor de administrat.

Fiecare pas are ca rezultat un produs bine definit (documente de specificaţie, model, etc.)

Modelul cascadă cu feedback propune remedierea problemelor descoperite în pasul precedent.

Problemele la pasul *i* care sunt descoperite la pasul *i* + 3 rămân neremediabile.

Un model realist ar trebui să ofere posibilitatea ca de la un anumit nivel să se poată reveni la oricare dintre nivelele anterioare.

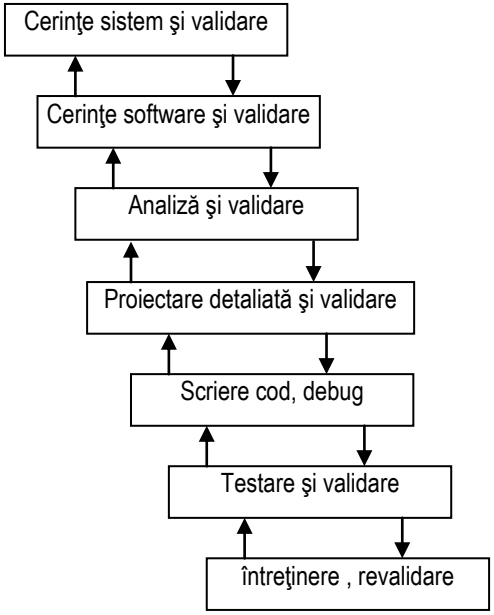


Fig. 3.4. Metodologia cascadă

Dezavantaje:

- clientul obţine o viziune practică asupra produsului doar în momentul terminării procesului de dezvoltare.

- modelul nu are suficientă putere descriptivă, în sensul că nu integrează activităţi ca managementul resurselor sau managementul configuraţiei. Aceasta face dificilă coordonarea proiectului.

- şi modelul cascadă impune îngheţarea specificaţiilor foarte devreme în procesul de dezvoltare pentru a evita iteraţiile frecvente

- reîntoarcerile în fazele anterioare atunci când în faza curentă s-au detectat erori:

  - în timpul analizei se descoperă erori de specificaţii,

  - în timpul implementării se descoperă erori de specificaţii/proiectare etc.,

astfel încât procesul poate implica multiple secvenţe de iteraţii ale activităţilor de dezvoltare).

- Îngheţarea prematură a cerinţelor conduce la :

- obţinerea unui produs prost structurat şi care nu execută ceea ce doreşte utilizatorul.

- obţinerea unei documentaţii neadecvate deoarece schimbările intervenite în iteraţiile frecvente nu sunt actualizate în toate documentele produse.

3.2.2.2. Metodologia spirală

Metodologia spirală, propusă tot de Boehm, este un alt exemplu bine cunoscut de metodologie a ingineriei programării.

Acest model încearcă să rezolve problemele modelului în cascadă, păstrând avantajele acestuia: planificare, faze bine definite, produse intermediare.

El defineşte următorii paşi în dezvoltarea unui produs:

- studiul de fezabilitate;
- analiza cerinţelor;
- proiectarea arhitecturii software;
- implementarea.

Modelul în spirală (fig. 3.5.) recunoaşte că problema principală a dezvoltării programelor este riscul.

Riscul nu mai este eliminat prin aserţiuni de genul: „în urma proiectării am obţinut un model corect al sistemului”, ca în modelul cascadă.

Aici riscul este acceptat, evaluat şi se iau măsuri pentru contracararea efectelor sale negative.

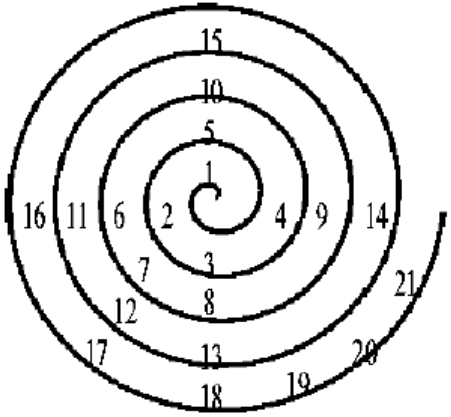


Fig. 3.5. Modelul spirală

Exemple de riscuri:

⇒ în timpul unui proces îndelungat de dezvoltare, cerinţele noi ale clientului sunt ignorate;

⇒ clientul schimbă cerinţele;

⇒ o firmă concurentă lansează un program rival pe piaţă;

⇒ un dezvoltator /arhitect părăseşte echipa de dezvoltare;

⇒ o echipă nu respectă un termen de livrare pentru o anumită componentă

Metodologia spirală cuprinde următoarele etape, grupate pe patru cicluri (Tabelul 3.1)

În modelul spirală se consideră că fiecare pas din dezvoltare conţine o serie de activităţi comune:

- pregătirea: se identifică obiectivele, alternativele, constrângerile;
- gestionarea riscului: analiza şi rezolvarea situaţiilor de risc;
- activităţi de dezvoltare specifice pasului curent (de exemplu analiza specificaţiilor sau scrierea de cod);
- planificarea următorului stadiu: termenele limită, resurse umane, revizuirea stării proiectului.

Procesul începe în centrul spiralei. Fiecare ciclu terminat reprezintă o etapă. Pe măsură ce spirala este parcursă, produsul se maturizează. Cu fiecare ciclu, sistemul se apropie de soluţia finală.

Deşi este considerată ca un exemplu generic pentru metodologia ciclică, metoda are şi elemente secvenţiale, puse în evidenţă de evoluţia constantă de la o etapă la alta.

Tabelul 3.1. Ciclurile metodologiei în spirală

Ciclul 1 – Analiza preliminară:
1. Obiective, alternative, constrângeri 2. Analiza riscului şi prototipul 3. Conceperea operaţiilor 4. Cerinţele şi planul ciclului de viaţă 5. Obiective, alternative, constrângeri 6. Analiza riscului şi prototipul
Ciclul 2 – Analiza finală:
7. Simulare, modele, benchmark-uri

8. Cerinţe software şi acceptare 9. Plan de dezvoltare 10. Obiective, alternative, constrângeri 11. Analiza riscului şi prototipul
Ciclul 3 – Proiectarea:
12. Simulare, modele, benchmark-uri 13. Proiectarea produsului software, acceptare şi verificare 14. Integrare şi plan de test 15. Obiective, alternative, constrângeri 16. Analiza riscului şi prototipul operaţional
Ciclul 4 – Implementarea şi testarea:
17. Simulare, modele, benchmark-uri 18. Proiectare detaliată 19. Cod

20. Integrarea unităţilor şi testarea acceptării
21. Lansarea produsului

3.2.2.4. Metodologia spirală WinWin

Această metodologie extinde spirala Boehm prin adăugarea unui pas de stabilire a priorităţii la începutul fiecărui ciclu din spirală şi prin introducerea unor scopuri intermediare, numite *puncte ancoră*.

- Modelul spirală WinWin identifică un punct de decizie.

- Pentru fiecare punct de decizie, se stabilesc obiectivele, constrângerile şi alternativele.

- Punctele ancoră stabilesc trei scopuri intermediare.

- Primul punct ancoră, numit *obiectivul ciclului de viaţă*, precizează cazurile sigure de funcţionare pentru întregul sistem, arătând că există cel puţin o arhitectură fezabilă (adică posibilă din punct de vedere practic) care satisface scopurile sistemului.

- Primul scop intermediar este stabilit când sunt terminate obiectivele de nivel înalt ale sistemului, arhitectura, modelul ciclului de viaţă şi prototipul sistemului.

- Această primă ancoră spune de ce, ce, când, cine, unde, cum şi estimează costul produsului.

După executarea acestor operaţii, este disponibilă analiza de nivel înalt a sistemului.

- Al doilea punct ancoră defineşte *arhitectura ciclului de viaţă*, iar al treilea – *capacitatea operaţională iniţială*, incluzând mediul software necesar, hardware-ul, documentaţia pentru client şi instruirea acestuia.

Aceste puncte ancoră corespund etapelor majore din ciclul de viaţă al unui produs: dezvoltarea iniţială, lansarea, funcţionarea, întreţinerea şi ieşirea din funcţiune.

3.2.2.5. Prototipizarea

O problemă generală care apare la dezvoltarea unui program este să ne asigurăm că utilizatorul obţine exact ceea ce vrea. Prototipizarea vine în sprijinul rezolvării acestei probleme. Încă din primele faze ale dezvoltării, clientului i se prezintă o versiune funcţională a sistemului.

Această versiune nu reprezintă întregul sistem, însă este o parte a sistemului care cel puţin funcţionează. Prototipul ajută clientul în a-şi defini mai bine cerinţele şi priorităţile. Prin intermediul

unui prototip, el poate înţelege ce este posibil şi ce nu din punct de vedere tehnologic. Prototipul este de obicei produs cât mai repede; pe cale de consecinţă, stilul de programare este de obicei (cel puţin) neglijent. Însă scopul principal al prototipului este de a ajuta în fazele de analiză şi proiectare şi nu folosirea unui stil elegant.

Se disting două feluri de prototipuri:

- jetabil sau de aruncat (throw-away);
- evoluţionar sau lent şi recuperabil.

În cazul realizării unui prototip jetabil, scopul este exclusiv obţinerea unei specificaţii. De aceea nu se acordă nici o importanţă stilului de programare şi de lucru, punându-se accent pe viteza de dezvoltare. Odată stabilite cerinţele, codul prototipului este „aruncat”, sistemul final fiind rescris de la început, chiar în alt limbaj de programare.

În cazul realizării unui prototip evoluţionar, scopul este de a crea un schelet al aplicaţiei care să poată implementa în primă fază o parte a cerinţelor sistemului. Pe măsură ce aplicaţia este dezvoltată, noi caracteristici sunt adăugate scheletului existent. În contrast cu prototipul de aruncat,

aici se investeşte un efort considerabil într-un design modular şi extensibil, precum şi în adoptarea unui stil elegant de programare.

Această metodă are următoarele avantaje:

- permite dezvoltatorilor să elimine lipsa de claritate a specificaţiilor;
- oferă utilizatorilor şansa de a schimba specificaţiile într-un mod ce nu afectează drastic durata de dezvoltare;
- întreţinerea este redusă, deoarece acceptarea se face pe parcursul dezvoltării;
- se poate facilita instruirea utilizatorilor finali înainte de terminarea produsului.

Dintre dezavantajele principale ale prototipizării amintim:

- deoarece prototipul rulează într-un mediu artificial, anumite dezavantaje ale produsului final pot fi scăpate din vedere de clienţi;
- clientul nu înţelege de ce produsul necesită timp suplimentar pentru dezvoltare, având în vedere că prototipul a fost realizat atât de repede;
- deoarece au în fiecare moment şansa de a face acest lucru, clienţii schimbă foarte des specificaţiile;

- poate fi nepopulară printre dezvoltatori, deoarece implică, în cazul prototipului jetabil, renunţarea la propria muncă.

Acest model, cunoscut de asemenea sub numele de prototip rapid, încearcă să rezolve neajunsul modelului în cascadă legat de faptul că până la construirea produsului final nu se ştie cu exactitate ce a dorit într-adevăr utilizatorul. Dezvoltarea evolutivă este similară abordării exploratorii, dar scopul acestei dezvoltări este de a stabili cerinţele utilizatorului, mai ales atunci când este dificil de făcut acest lucru, sau când documentele de specificaţii existente sunt ambigui sau incomplete. Această tehnică este în esenţă o metodă de analiză a specificaţiilor.

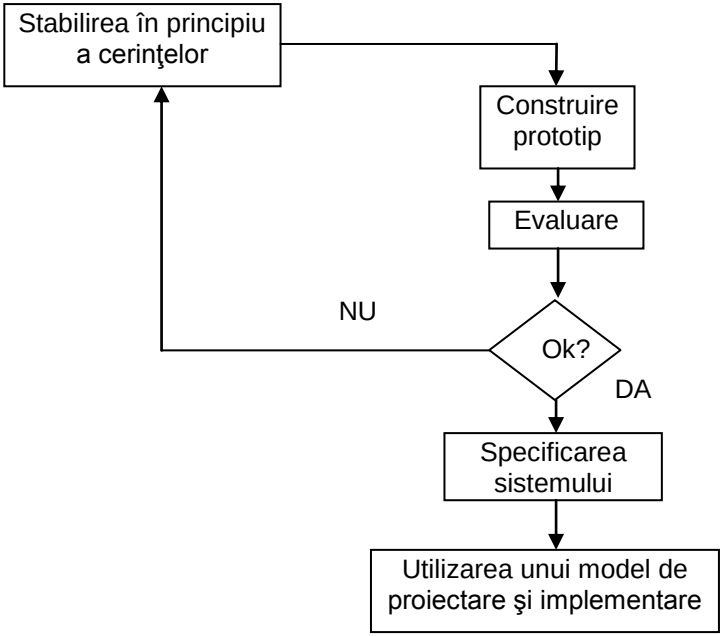


Fig.3.6. Modelul prototipului rapid

Plecând de la stabilirea în mare a specificaţiilor, inginerul software construieşte un prototip rapid şi lasă clientul să-l experimenteze. În momentul în care clientul este satisfăcut, proiectantul poate trece la elaborarea documentului cu specificaţiile produsului, care este apoi folosit în construcţia software-ului utilizând, de exemplu, un model în cascadă (fig.3.7).

Când se construieşte un prototip, proiectarea şi implementarea se realizează în faze diferite de timp.

Cele două abordări pot fi combinate în mod util aşa cum se arată în fig. 3.7. În acest caz, stadiul de analiză a cerinţelor din cadrul modelul în cascadă a fost înlocuit de stadiul necesar al prototipizării iar bucla de feedback a dispărut.

Punctul forte al acestui model combinat este acela că, în acest mod dezvoltarea software-ului devine, în mod esenţial, un proces complet liniar. Datorită completitudinii specificaţiilor utilizatorului, buclele de reacţie (feedback) din stadiile precedente par a fi, din ce în ce mai puţin, necesare.

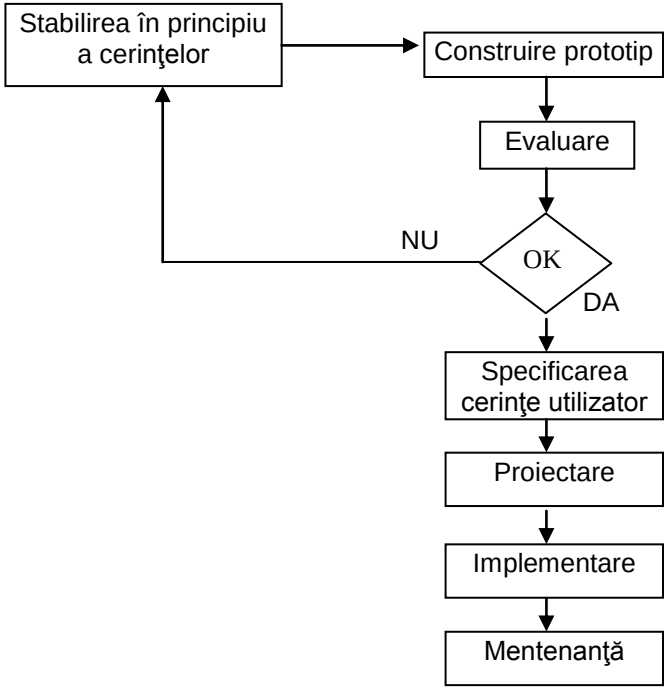


Fig.3.7. Modelul integrat “cascadă şi prototipizare”

3.2.2.6. Metode formale

În acest model de dezvoltare, sunt folosite formalismul şi rigoarea matematicii. În prima fază este construită o specificaţie în limbaj matematic. Apoi, această specificaţie este transformată în programe, de obicei într-un proces incremental.

Avantaje:

- precizia obţinută prin specificarea formală;
- păstrarea corectitudinii în timpul transformării specificaţiei în cod executabil;
- oferă posibilitatea generării automate de cod;
- sunt potrivite pentru sisteme cu cerinţe critice.

Dezavantaje:

- specificarea formală este de obicei o barieră de comunicare între client şi analist;
- necesită personal foarte calificat (deci mai scump);
- folosirea impecabilă a tehnicilor specificării formale nu implică neapărat obţinerea de programe sigure, deoarece anumite aspecte critice pot fi omise din specificaţiile iniţiale.

3.2.2.7. Programarea Extrema

Programarea Extrema (Kent Beck, 1996) este o metodologie care propune rezolvări originale pentru problemele care apar în dezvoltarea de programe.

Este o metodă de programare „agilă”, potrivită dezvoltării rapide de aplicaţii

Fiind o tehnologie nouă (şi extremă) are atât adepţi cât şi critici. XP consideră că dezvoltarea programelor nu înseamnă ierarhii, responsabilităţi şi termene limită, aşa cum se află acestea pe masa administratorului, ci înseamnă colaborarea oamenilor din care este formată echipa. Aceştia sunt încurajaţi să îşi afirme personalitatea, să ofere şi să primească cunoştinţe şi să devină programatori străluciţi.

De asemenea, XP consideră că dezvoltarea de programe înseamnă în primul rând scrierea de programe. Această sintagmă banală se pare că este uitată de multe companii care se ascund în spatele proceselor de dezvoltare stufoase, a şedinţelor şi a rapoartelor de activitate. XP ne aminteşte cu respect ca fişierele PowerPoint nu se pot compila.

De altfel, inspirarea proceselor de dezvoltare a programelor din ingineria construcţiilor se pare că nu este cea mai fericită alegere. Este adevărat că un inginer care vrea să construiască un pod peste un râu face mai întâi măsurători, realizează un proiect şi abia apoi trece la execuţie, toate acestea într-un mod secvenţial şi previzibil. Dar dezvoltarea de programe nu seamănă cu aşa ceva, oricât am vrea să credem asta. Dacă inginerului constructor respectiv i s-ar schimba cerinţele de rezistenţă şi i s-ar muta malurile chiar când a terminat de construit jumătate de pod, putem fi siguri că acel inginer şi-ar schimba modul de lucru. Din păcate însă, nu ştim (încă) cum. Iniţiatorii XP definesc următoarele două carte, ca bază filosofică pentru această metodologie.

Principiile metodelor agile

- Implicarea clientului

  - Clientul trebuie implicat pe tot parcursul procesului de dezvoltare. Rolul său este de a prioritiza noile cerinţe şi de a evalua iteraţiile sistemului

- Livrarea incrementală

  - Programul este dezvoltat incremental, clientul indicând cerinţele care trebuie incluse la fiecare iteraţie

- Oamenii nu procesul

  - Abilităţile echipei de dezvoltare trebuie recunoscute şi exploatate. Echipa trebuie lăsată să-şi contureze propriile modalităţi de lucru, fără a i se da reţete

- Acceptarea schimbării

  - Echipa trebuie să se aştepte ca cerinţele să se schimbe iar proiectarea sistemului trebuie făcută astfel încât să se adapteze uşor la aceste schimbări

- Menţinerea simplităţii

  - Concentrare pe simplitate atât în programele dezvoltate cât şi în procesul de dezvoltare. Oricând este posibil, trebuie eliminată complexitatea din sistem

Problemele metodelor agile

- Este dificilă menţinerea interesului clienţilor implicaţi în proces
- Membrii echipei pot fi incapabili să se adapteze la implicarea intensă caracteristică metodelor agile
- Când există mai mulţi factori de decizie, este dificilă prioritizarea schimbărilor
- Menţinerea simplităţii necesită lucru suplimentar

- Contractele pot fi o problemă în cazul dezvoltării iterative

Carta drepturilor dezvoltatorului:

- Ai dreptul să ştii ceea ce se cere, prin cerinţe clare, cu declaraţii clare de prioritate;
- Ai dreptul să spui cât îţi va lua să implementezi fiecare cerinţă, şi să îţi revizuieşti estimările în funcţie de experienţă;
- Ai dreptul să îţi accepţi responsabilităţile, în loc ca acestea să-ţi fie asignate;
- Ai dreptul să faci treabă de calitate în orice moment;
- Ai dreptul la linişte, distracţie şi la muncă productivă şi plăcută.

Carta drepturilor clientului:

- Ai dreptul la un plan general, să ştii ce poate fi făcut, când, şi la ce preţ;
- Ai dreptul să vezi progresul într-un sistem care rulează şi care se dovedeşte că funcţionează trecând teste repetabile pe care le specifici tu;
- Ai dreptul să te răzgândeşti, să înlocuieşti funcţionalităţi şi să schimbi priorităţile;
- Ai dreptul să fii informat de schimbările în estimări, suficient de devreme pentru a putea reduce cerinţele astfel ca munca să se termine la data prestabilită. Poţi chiar să te opreşti

la un moment dat şi să rămâi cu un sistem folositor care să reflecte investiţia până la acea dată.

Aceste afirmaţii, deşi par de la sine înţelese, conţin semnificaţii profunde. Multe din problemele apărute în dezvoltarea programelor pornesc de la încălcarea acestor principii. Enumerăm pe scurt câteva dintre caracteristicile XP:

- Echipa de dezvoltare nu are o structură ierarhică. Fiecare contribuie la proiect folosind maximul din cunoştinţele sale;
- Scrierea de cod este activitatea cea mai importantă;
- Proiectul este în mintea tuturor programatorilor din echipă, nu în documentaţii, modele sau rapoarte;
- La orice moment, un reprezentant al clientului este disponibil pentru clarificarea cerinţelor;
- Codul se scrie cât mai simplu;
- Se scrie mai întâi cod de test;
- Dacă apare necesitatea rescrierii sau eliminării codului, aceasta se face fără milă;
- Modificările aduse codului sunt integrate continuu (de câteva ori pe zi);

- Se programează în echipă (programare în perechi). Echipele se schimbă la sfârşitul unei iteraţii (1-2 săptămâni);
- Se lucrează 40 de ore pe săptămână, fără lucru suplimentar.

Concluzii

Au fost prezentate aici cele mai importante metodologii de dezvoltare a programelor, mai puţin metodologia orientată obiect care a devenit în momentul de faţă, suverană şi despre care vom vorbi în continuare. Mai întâi au fost descrise metodologiile generice: secvenţială, ciclică şi hibridă, cu avantajele şi dezavantajele fiecăreia. Apoi s-au amintit câteva metode concrete de dezvoltate: modelul cascadă, modelul spirală, WinWin, prototipizarea, metodologia Booch, metodele formale şi aşa-numita „programare extremă”.

3.2.2.8. Metodologia Open Source

Metodologia Open Source înseamnă “sursă la vedere”. Este o abordare recentă, apărută ca urmare a dezvoltării mijloacelor de comunicaţie: FTP, e-mail, grupuri de discuţie Exemple clasice sunt: sistemul de operare Linux, browser-ul Netscape 5.

Codul sursă este transmis utilizatorului final într-o manieră non-proprietară (fără patent), pe baza unei licenţe open-source (gen GNU+ sistem de operare Open Source gen Unix).

Critici

- Nu există documente de proiectare sau alte documentaţii ale proiectului
- Nu se realizează testarea la nivel de sistem
- Nu există cerinţe ale utilizatorilor în afară de funcţionalitatea de bază
- Marketingul produsului este incomplet

O iteraţie tipică pentru o astfel de abordare este:

- Dezvoltatorul realizează proiectarea şi codarea (individual sau în echipă);
- Ceilalţi dezvoltatori sau comunitatea de utilizatori realizează debugging-ul şi testare;
- Noile funcţionalităţi dorite şi erorile depistate sunt trimise iniţiatorului proiectului;
- Se lansează o nouă versiune cu erorile corectate şi se analizează noile cerinţele;
- Se distribuie o listă de sarcini către comunitatea de utilizatori, căutându-se membri voluntari care să execute sarcinile de pe listă;

3.2.2.9. Reverse Engineering - Inginerie inversă

- Se analizează sistemele anterioare şi se încearcă reutilizarea componentelor existente:
  - Software, hardware
  - Documentaţie, metode de lucru
- Unele componente nu pot fi utilizate, altele trebuie modificate (în faza de proiectare);
- Componentele selectate sunt integrate direct în sistem.

3.2.2.10. Metodologia de dezvoltare Offshore

Offshore Înseamnă în limba engleză , "în larg". Companiile consideră ca profitabil outsourcing-ul pentru funcţiile care pot fi dezvoltate mai ieftin în altă ţară

Venituri din outsourcing IT în 2004 au fost:

- India – 43%
- Canada – 32%
- China – 5%
- Europa de Est – 5%

Motivarea externalizării (outsourcing) este:

- Reducerea costurilor;
- Creşterea eficienţei;
- Concentrarea asupra obiectivelor critice ale proiectului;
- Accesarea flexibilă a unor resurse care altfel nu ar fi accesibile (de exemplu personal înalt calificat)
- Dimensiunea mare a proiectului.

Etapele metodologiei sunt:

1. Iniţierea proiectului (local)
2. Analiza cerinţelor (local)
3. Proiectarea de nivel înalt (local)
4. Proiectarea detaliată (offshore)
5. Implementarea (offshore)
6. Testarea (offshore sau local)
7. Livrarea (local).

3.2.2.12. Metodologia orientată pe obiect

Metodologia proiectării orientate pe obiect “inversează” metodologiile anterioare, în special metodologiile funcţionale, aşa cum au fost ele concepute de Yourdan, în sensul că focalizează abordarea pe identificarea obiectelor din domeniul aplicaţiei, “potrivind” apoi procedurile în jurul acestor obiecte. La o eventuală modificare a cerinţelor , nu va mai fi necesar să fie schimbată toată structura obiectelor.

Din start, termenul *orientat pe obiect* înseamnă organizarea software-ului ca o colecţie de obiecte discrete, fiecare obiect încorporând atât structuri de date, cât şi comportament.

Ce este un obiect?

Un obiect poate fi considerat o entitate care încorporează atât structuri de date, numite atribute, cât şi comportament, denumit operaţii.

Un obiect trebuie să aibă caracteristicile următoare:

- **Identitate** : obiectul este o entitate discretă, care se distinge dintre alte entităţi. Exemple: o fereastră pe o staţie de lucru, un triunghi isoscel, o listă de persoane.

- **Clasificare** : obiectele cu aceleaşi atribute şi operaţii se grupează în clase. Fiecare obiect cu aceleaşi atribute şi operaţii poate fi considerat ca o instanţă a unei clase. Exemple: fereastră, triunghi, listă.

- **Polimorfism** : aceeaşi operaţie (cu acelaşi nume) poate să aibă comportament diferit în clase diferite. Exemple: *a muta* o fereastră, *a muta* un triunghi. Operaţia *a muta* înseamnă lucruri diferite, depinzând de obiectul asupra căruia se aplică. Implementarea concretă a unei operaţii într-o clasă se numeşte metodă.

- **Moştenire**: Este o caracteristică a abordării obiectuale şi se referă la transmiterea pe cale ierarhică a atributelor şi metodelor tuturor claselor descendente, într-o relaţie ierarhică.

Modele de lucru în OMT

Metodologia de proiectare orientată pe obiect s-a dezvoltat iniţial ca tehnică de modelare a obiectelor cunoscută şi sub numele de OMT, în engleză Object Modelling Technique şi a evoluat apoi printr-o tehnologie unificată numită UML.

OMT-ul foloseşte trei modele de bază şi anume: modelul obiectelor, modelul dinamic, modelul funcţional.

Modelul obiectelor:

- descrie structura statică a obiectelor din sistem şi relaţiile dintre ele;

- descrie ce se modifică în sistem (adică obiectele);

- este reprezentat cu ajutorul diagramelor de obiecte.

O diagramă de obiecte este un graf ale cărui noduri sunt *obiectele* şi ale cărui arce sunt *relaţiile* dintre obiecte.

Modelul dinamic:

- descrie acele aspecte ale sistemului care se schimbă în timp;

- specifică şi implementează partea de control a sistemului;

- descrie când se modifică sistemul;

- este reprezentat cu ajutorul diagramelor de stare.

Modelul funcţional:

- descrie transformările valorilor datelor în cadrul sistemului;

- descrie cum se modifică sistemul;

- este reprezentat cu ajutorul diagramelor de flux de date.

O diagramă de flux de date este un graf ale cărui noduri sunt procesele şi ale cărui arce sunt fluxurile de date.

Etapele aplicării metodologiei orientate pe obiect

Metodologia OO pentru dezvoltarea software-ului constă din construirea, în etapa de analiză a sistemului, a unui model complet al aplicaţiei, care va cuprinde cele trei modele prezentate anterior.

Ulterior se vor adăuga detaliile de implementare necesare. Etapele sunt: analiza, proiectarea sistemului, proiectarea obiectelor, implementarea sistemului.

- Analiza

Pornind de la specificarea cerinţelor, analistul va concepe un model cu obiecte din domeniul aplicaţiei şi nu al programării (ca de exemplu liste, arbori, etc).

- Proiectarea sistemului

Proiectantul de sistem va lua decizii generale asupra arhitecturii globale a sistemului, va alege o strategie de implementare şi o strategie de alocare a resurselor. De asemenea, va trebui să stabilească împărţirea sistemului mare în subsisteme.

- Proiectarea obiectelor

Proiectantul obiectelor va adăuga detalii de implementare modelului obţinut la analiză, în concordanţă cu strategia aleasă în etapa de proiectare a sistemului. Accentul se va pune acum pe algoritmii şi structurile de date folosite pentru a implementa clasele obţinute în etapa de analiză.

- Implementarea

În cele din urmă, clasele şi relaţiile dintre clase obţinute în celelalte etape vor fi traduse într-un limbaj de programare, într-o bază de date sau vor fi implementate hardware. Este important de remarcat că deşi analiza unui sistem poate să se efectueze folosind noţiuni de obiecte şi clase, nu este neapărat necesar ca implementarea să se facă într-un limbaj de programare orientat pe obiect (cum ar fi Java, C++, Smalltalk, Eiffel sau ADA). Implementarea poate fi făcută în orice limbaj de programare. Un exemplu în acest sens îl constituie biblioteca de elemente de interfaţă pentru X Window, numită Motif, care este scrisă în C, deşi a fost proiectată folosindu-se noţiuni de analiză orientată pe obiect.

Conceptele de bază

A. Abstractizarea

Accentul pentru un obiect se pune pe ce este acesta şi nu pe ce trebuie să facă, înainte de a stabili concret detaliile de implementare. De aceea, etapa esenţială în crearea unei aplicaţii orientate pe obiect este analiza şi nu implementarea, care poate deveni mecanică şi în mare parte automatizată, dacă se folosesc instrumente software specializate (gen CASE).

B. Încapsularea (ascunderea informaţiei)

Încapsularea constă în separarea aspectelor externe ale unui obiect, care sunt accesibile altor obiecte, de aspectele de implementare interne ale obiectului, care sunt ascunse celorlalte obiecte. Utilizatorul obiectului poate accesa doar anumite atribute şi operaţii, în scopul perfecţionării unui algoritm sau eliminării unor erori. Încapsularea ne va împiedica să modificăm toate caracteristicile obiectului, iar aplicaţiile care utilizează obiectul nu vor avea de suferit.

C. Împachetarea datelor şi a comportamentului în acelaşi obiect

Aceasta se referă tocmai la faptul că un obiect conţine atât structuri de date cât şi operaţii, şi permite să se ştie întotdeauna cărei clase îi aparţine o anumită metodă, chiar dacă există mai multe operaţii cu aceeaşi nume, în clase diferite.

D. Partajarea

Tehnicile orientate pe obiect promovează partajarea la diverse niveluri. Partajarea poate însemna transmiterea aceloraşi structuri de date şi respectiv, operaţii de-a lungul unor clase, dintr-o ierarhie de clase. Acest lucru are un efect benefic asupra economisirii spaţiului. Pe de altă parte,

partajarea oferă posibilitatea reutilizării proiectării, respectiv a codului anumitor clase, în proiectele ulterioare.

E. Accentul pus pe structura de obiect, nu pe cea de procedură

Este una dintre caracteristicile principale ale tehnicilor orientate pe obiect. Dacă în tehnicile funcţionale (sau procedurale) accentul în analiza sistemului cade pe descompunerea funcţională a acestuia, deci pe ceea ce trebuie să facă sistemul (în ultimă instanţă pe construirea diagramelor de flux), în tehnicile OOP accentul cade pe înţelegerea sistemului din punct de vedere al descompunerii acestuia în entităţi (obiecte) şi în stabilirea relaţiilor dintre acestea, deci pe ce este un obiect. Mult înainte de a stabili ce face sistemul, problema care se pune constă în a preciza cine execută şi care sunt relaţiile între cei care execută; deci mai importantă este diagrama obiectelor.

Analiza orientată pe obiecte

Analiza este primul pas al metodologiei tehnicii de modelare orientată pe obiecte şi are ca scop construirea unui model precis, concis şi concret al lumii reale.

În figura 3.8 este prezentată o vedere generală asupra procesului de analiză. Analistul, cel care realizează analiza problemei, începe cu definirea problemei, aşa cum este ea formulată de potenţialii

utilizatori. Analistul va construi un model, care poate fi incomplet, pe baza unor cerinţe incomplete. De aceea, modelul trebuie apoi rafinat.

Primul pas în definirea problemei este specificarea cerinţelor. Se va stabili **ceea ce** trebuie să facă aplicaţia şi **nu cum,** şi anume se vor defini:

- scopul problemei;
- necesităţile;
- contextul aplicaţiei;
- diverse presupuneri;
- performanţele necesare;

În partea de proiectare şi implementare se vor urmări:

- algoritmi;
- structurile de date;
- arhitectura;
- optimizările.

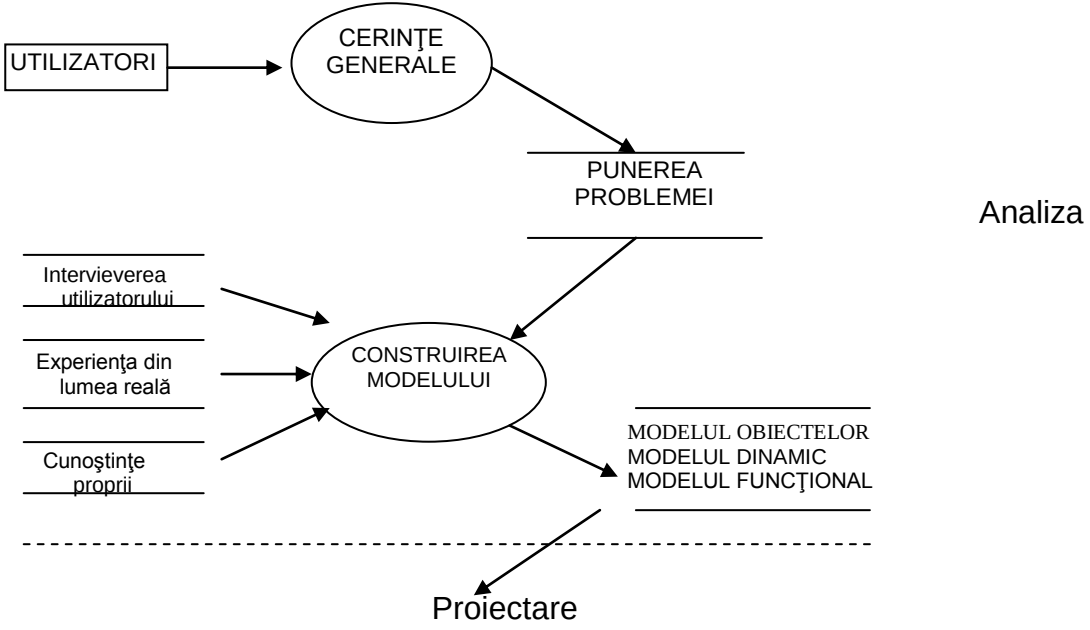


Fig.3.8. Procesul de analiza – vedere generală

Concluzii

1. Simpla adoptare a unei metodologii fără o analiză temeinică a cerinţelor şi contextului de lucru nu este fezabilă.
2. Fără sprijin din partea factorilor de decizie executivă, dezvoltarea proiectului este complexă şi de durată.
3. Metodologia este o tactică ce trebuie considerată numai după determinarea strategiei generale a companiei

UML- Limbaj unificat de modelare

4.1. Introducere în UML

Limbajul unificat de modelare, numit prescurtat UML se doreşte a fi un instrument care să ofere o metodologie unificată de modelare, analiză şi proiectare a sistemelor informatice.

El oferă o arhitectură de sistem pentru analiza şi proiectarea orientată obiect a sistemelor software, incluzând şi documentarea acestora, dar şi pentru modelarea altor sisteme non-software, cum ar fi modelarea afacerilor, de exemplu.

- Metode de proiectare orientate pe obiect
- Metode de proiectare structurate

} ⇨ Modele inconsistente

Scopurile principale sunt reutilizarea, portabilitatea şi interoperabilitatea software-lui orientat pe obiect.

Adoptarea specificaţiei UML a redus gradul de confuzie în cadrul industriei limbajelor de programare şi a permis schimbul reciproc între instrumentele vizuale de dezvoltare.

Prima sa versiune a apărut în Ianuarie 1997 şi a fost dezvoltat: Grady Booch, Jim Rumbaugh şi Ivar Jacobson.

UML a fost selectat ca standard al limbajelor de modelare orientate obiect (OOML- Object Oriented Modeling Language) de către OMG şi este utilizat de dezvoltatorii de produse software.

CE ESTE UML ?

Sunt multe definiţii şi nici una unanim acceptată sau standardizată.

Dar toate au în comun următoarele:

UML este un limbaj grafic pentru vizualizarea, specificarea, construrirea şi documentarea componentelor unui sistem software.

Mai mult se poate spune **că UML este un limbaj vizual de modelare, dar care nu are pretenţia de a fi un limbaj de programare vizual, el putând fi utilizat ca un limbaj universal pentru descrierea sistemelor**.

UML poate fi utilizat in toate domeniile ingineriei software oferind un mod standard de a scrie proiecte de sistem, incluzând obiectele conceptuale şi funcţiile de sistem precum şi obiecte concrete cum ar fi declaraţiile din limbajul de programare, scheme ale bazelor de date şi componente software reutilizabile.

Acest limbaj unificat reprezintă o arhitectură bazată pe patru niveluri de abstractizare definite în metamodelul UMLşi anume:

1. ***Meta-metamodelul*** – infrastructura pentru o arhitectură de modele;
2. ***Metamodelul*** – o instanţă a unui meta-metamodel şi defineşte semantica necesară pentru reprezentarea modelelor aplicaţiei;
3. ***Modelul*** – o instanţă a unui metamodel;
4. ***Obiecte utilizator*** – o instanţă a unui model, utilizate pentru descrierea unui domeniu specific de informaţie.

Metamodelul UML este un model logic şi are în componenţa sa trei pachete logice:

- *Elemente de comportament (Behavioral Elements);*
- *Elemente de bază (Foundation);*
- *Mecanisme generale (Model Management).*

Avantajele unui metamodel logic este acela că accentuează declarativele semantice, înlăturând detalile de implementare.

Implementările care utilizează metamodelul logic trebuie să se conformeze semanticilor sale şi să fie capabil să importe şi să exporte obiecte.

În cadrul fiecărui pachet, elementele unui element sunt definite astfel: sintaxă abstractă, reguli foarte bine formulate sau reguli de corectitudine şi semantică.

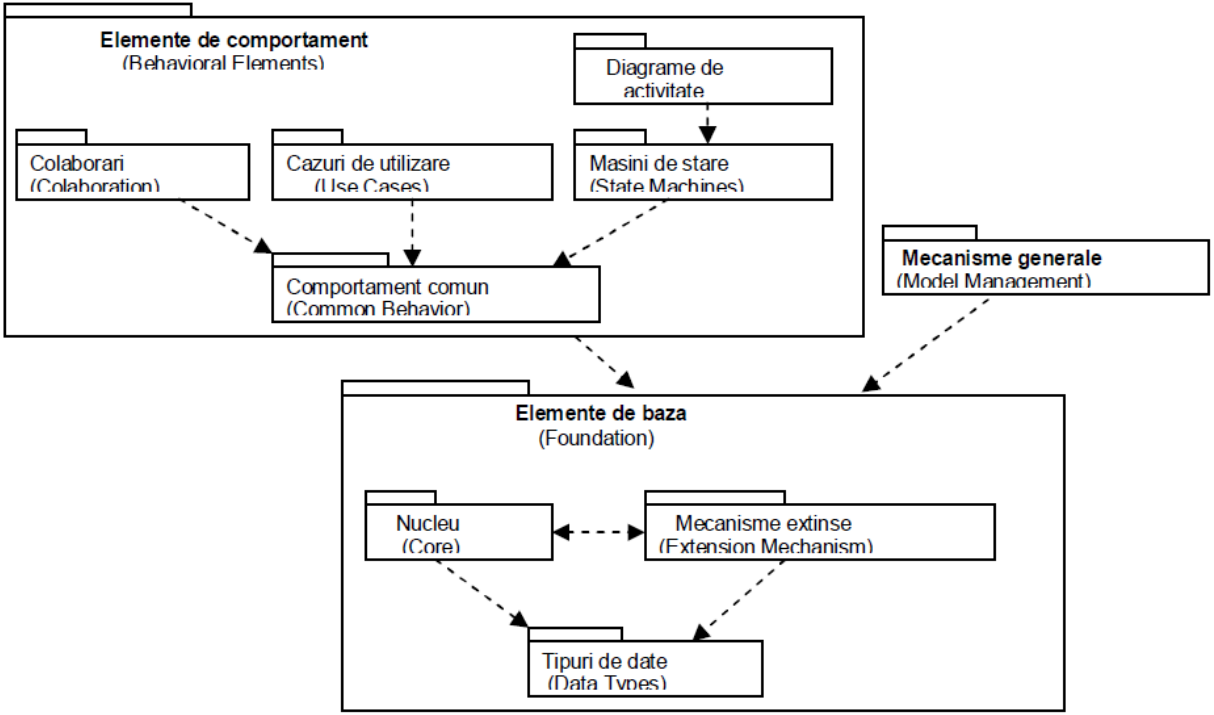


Fig. 4.1. Structura UML

Componentele de bază ale UML sunt elementele model.

Un ***model*** reprezintă o colecţie de *obiecte* (clase, pachete, actori, cazuri de utilizare, componente şi noduri), *relaţii* (de asociere, dependenţă, generalizări) şi *diagrame*.

Un model :

- Este entitatea de baza a proiectarii;
- Este individualizat;
- Este legat de alte modele prin legaturi;
- Este reprezentat grafic.

Modelele sunt de două tipuri:

- ***modele structurale*** *(statice)* care accentuează structura obiectelor din sistem, incluzând şi clasele lor, interfeţele, atributele şi relaţiile şi
- ***modele de comportament*** *(dinamice)*, care accentuează comportamentul obiectelor din sistem, incluzând şi metodele, interacţiunea, colaborările şi starea lor istorică.

Principalele obiective ale UML-ului sunt:

- Să pună la dispoziţia utilizatorului un limbaj vizual pentru dezvoltarea şi specificarea sistemului;
- Să suporte concepte de dezvoltare de nivel superior cum sunt componente, colaborări, modele;
- Să încurajeze utilizarea limbajelor de programare orientate pe obiect;
- Să furnizeze o bază formală pentru înţelegerea limbajului de modelare.

4.2. Diagrame şi concepte UML

UML defineşte patru tipuri de diagrame :

1. Diagrama de clase (Class Diagram);
   1.1. Diagrama de obiecte (Object Diagram);
2. Diagrama cazurilor de utilizare (Use Case Diagram);
3. Diagrame de comportament (Behavior Diagrams);
   3.3.1. Diagrama de stare, sau grafice de stări (Statechart Diagram);
   3.3.2. Diagrama de activitate (Activity Diagram);

3.3.3. Diagrama de interacţiuni (Interaction Diagrams);

3.3.4. Diagrama secvenţială (Sequence Diagram);

3.3.5. Diagrama de colaborare (Collaboration Diagram);

4. Diagrame de implementare (Implementation Diagrams);

4.1. Diagrama de componente (Component Diagram);

4.2. Diagrama de aplicaţie (de dezvoltare) (Deployment Diagram).

Aceste diagrame furnizează perspective multiple şi diferite asupra sistemului din punctul de vedere al analizei şi/sau dezvoltării.

4.2.1. Diagrama de clase (Class Diagram)

O diagrama a claselor este un graf ale cărui noduri sunt clasele din sistem şi ale cărui arce sunt relaţiile dintre clase.

Diagrama claselor este probabil cea mai importantă diagramă UML.

O astfel de diagramă poate conţine interfeţe, pachete, legături, şi chiar instanţe, cum ar fi obiecte. Clasele sunt tipuri abstracte de date. În cadrul programului se lucrează cu instanţieri ale claselor definite, numite *obiecte*.

Clasele de obiecte sunt categorii de obiecte cu **atribute** (structuri de date) şi **operaţii** (comportament) comune.

Fiecare obiect are propria sa:

- **stare** (definita de un set de valori păstrate în atributele sale);
- **identitate** (obiectele se disting între ele prin existenţă , nu prin atribute);
- **comportament** (felul în care un obiect acţionează şi reacţionează în termenii comunicării prin mesaje şi schimbării ale stării).

Scopul diagramei de clase este de a prezenta structura de clase din cadrul unui model.

În aplicaţiile orientate pe obiect clasele au atribute, operaţii şi relaţii cu alte clase.

Elementul fundamental al diagramei claselor este un dreptunghi care reprezintă o clasă, aşa cum este ilustrată în figura 4.2.

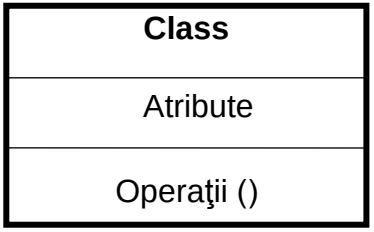


Fig. 4.2. Reprezentarea unei clase din diagrama de clase UML

Dreptunghiul care reprezintă clasa este împărţit în trei compartimente cu următoarele semnificaţii:

- Compartimentul de sus este cel mai important şi conţine numele clasei;
- Compartimentul din mijloc conţine o listă de atribute;
- Compartimentul de jos conţine o listă de operaţii.

În multe diagrame compartimentele de mijloc şi de jos sunt omise. Chiar şi atunci când sunt prezente, ele nu arată toate atributele şi operaţiile. Scopul este de a arata doar acele atribute şi operaţii care sunt esenţiale pentru diagrama respectivă.

Diagrama claselor poate conţine următoarele elemente model:

-  **Pachete.** Pachetele sunt utilizate pentru a structura modelul.
-  **Dependenţe între pachete.** Acestea arată că respectivele clase dintr-un pachet utilizează clasele pachetului de care depind.
-  **Colaborări între obiecte.**
-  **Interfeţe.** Interfeţele nu conţin atribute, ci doar operaţii.

- **Clase.** Clasele reprezintă cel mai important concept al programării orientată pe obiect şi al UML.
- **Relaţii de moştenire.** Acestea se pot regăsi între interfeţe sau între clase, dar niciodată între o interfaţă şi o clasă.
- **Relaţii de implementare.** Pot exista numai între interfeţe şi clase.
- **Relaţii de asociere.** Asocierile sunt relaţii între clase.

În construirea claselor care vor sta la baza proiectării programului trebuie clarificate entităţile care vor evolua în sistem şi operaţiile asociate.

Trebuie de asemenea definite responsabilităţile fiecărei clase în implementare şi colaborările între clasele modelate grafic prin legături de diverse tipuri şi cu diferite multiplicităţi.

Legăturile sunt simbolul relaţiilor între clase.

Un exemplu de reprezentare a unei clase şi anume clasa *Cerc* este prezentat în figura 4.3.

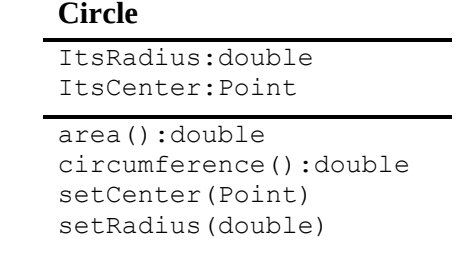


Fig. 4.3. Reprezentarea clasei *Circle*

Fiecare instanţă de tip *Circle* pare că are o instanţă de tip *Point.*

Aceasta este o relaţie cunoscută ca *relaţie de compoziţie* şi figurată în UML ca în fig. 4.4.

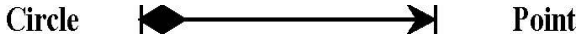


Fig. 4.4. Relaţie de compoziţie

Rombul negru reprezintă compoziţia. El este plasat în dreptul cercului deoarece cercul este compus din puncte. Punctul nu trebuie să ştie nimic despre cerc. Săgeata din partea cealaltă denotă faptul că relaţia poate fi parcursă numai într-un sens.

În UML se presupune că relaţiile sunt implicit bidirecţionale cu excepţia când se termină printr-o săgeată aşa cum am prezentat mai sus.

Dacă s-ar omite săgeta ar însemna că Punctul trebuie să cunoască Cercul, ceea ce la nivel de cod ar însemna să se include # *include " Cercle.h"* în *Point.h*.

Relaţiile de compoziţie sunt mai puternice decât cele de conţinut sau agregare.

Agregarea este întregul sau o parte din relaţie.

În cazul prezentat *Cercle* este întregul iar *Point* parte a *Cercle.*

Relaţia de compoziţie indică de asemenea că timpul de viaţă al *Point* depinde de *Cercle*. Aceasta înseamnă că, dacă Cercul este distrus va fi distrus şi Punctul.

În limbajul C++ această clasă se reprezintă astfel:

```
Class Circle {
   public:
    void SetCenter(const Point&);
    void SetRadius(double);
    double Area() const;
    double Circumference() const;
 private:
     double itsRadius;
   Point itsCenter;
};
```

În acest caz s-a reprezentat relaţia de compoziţie ca o variabilă membră. Se poate folosi la fel de bine un pointer care la sfârşit v-a fi şters de destructorul Cercului.

Clasificatori

Un *clasificator* este un element care descrie caracteristicile de comportament şi structurale ale sistemului.

Clasificatorii acceptaţi de UML sunt de mai multe tipuri şi includ: clase, tipuri de date, interfeţe, componente, semnale, noduri, cazuri de utilizare şi subsisteme.

Un clasificator declară un set de caracteristici care includ atribute, metode şi operaţii.

Numele unui clasificator este unic şi reprezintă o metaclasă abstractă.

UML admite următoarele tipuri speciale de clasificatori, numite *stereotipuri* :

- <<metaclasă>> - precizează că instanţele clasificatorului sunt clase;
- <<powertype>> - specifică un clasificator ale cărui obiecte sunt descendenţii unui anumit părinte;
- <<process>> – un clasificator care reprezintă un flux de control cu o interfaţă puternică;
- <<thread>> – un clasificator care reprezintă un flux de control;
- <<utility>> – specifică un clasificator care nu are instanţe;

Un *atribut* descrie un domeniu de valori pe care le pot lua instanţele unui clasificator.

El poate avea o valoare iniţială şi una dintre următoarele proprietăţi care se referă la posibilităţile de modificare a valorii, după ce obiectul a fost creat:

- *changeable* (modificabil)- nu se impune nici o restricţie asupra valorii atributului;

- *addOnly* – valabil numai pentru atributele cu ordin de multiplicitate mai mare ca unu; o valoare odată adăugată nu mai poate fi ştearsă sau modificată;
- *frozen* – valoarea atributului nu poate fi modificată după iniţializarea obiectului.

O **operaţie** este un serviciu care poate fi solicitat de către un obiect.

O operaţie în UML poate avea una dintre următoarele proprietăţi care se referă la simultaneitatea apelurilor concurente către o clasă pasivă:

- *isQuery* – indică dacă se schimbă sau nu starea sistemului. Dacă are valoarea *true* starea sistemului rămâne neschimbată.
- *sequential* – apelurile trebuie efectuate secvenţial;
- *guarded* – se pot produce simultan apeluri multiple către o instanţă dar numai unul are permisiunea să înceapă, celelalte fiind blocate până la finalizarea operaţiei.
- *concurrent* - se pot produce simultan apeluri multiple către o instanţă, fiecare dintre ele putând fi realizate în paralel.

Operaţiile pot avea parametri care sunt utilizaţi pentru specificare şi fiecare include un nume, un tip şi direcţia comunicării.

Direcţia se poate indica astfel:

- *in* – parametru de intrare care nu poate fi modificat;
- *out* – parametru de ieşire care poate fi modificat pentru a transmite informaţii către alte elemente;
- *inout* – parametru de intrare care poate fi modificat;
- *return* – valoare returnată de un apel.

Multiplicitatea unei clase specifică numărul de instanţe pe care le poate avea. Multiplicitatea se aplică şi la nivel de atribut.

O diagramă tipică de clase este prezentată în figura 4.5.

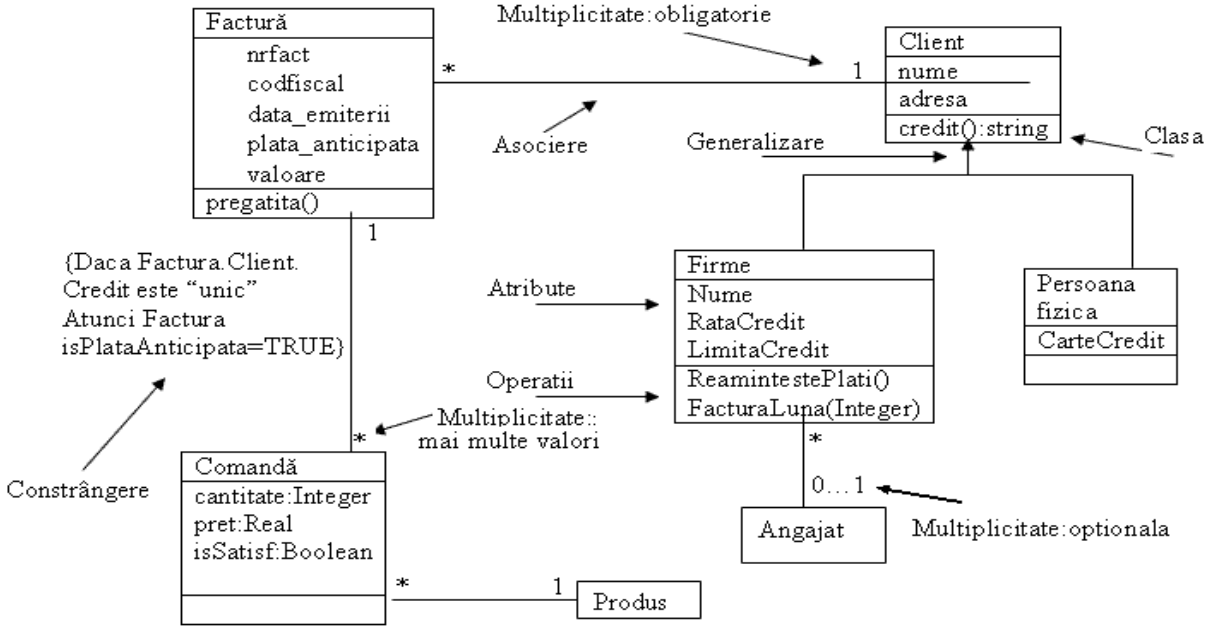


Fig. 4.5. Diagramă de clase

4.2.1.1. Diagrama de obiecte

Această diagramă evidenţiază un set de obiecte şi relaţiile cu alte obiecte la un moment dat.

O diagrama de obiecte poate fi considerată un caz special al diagramelor claselor sau al diagramei de colaborări şi este o instanţă a unei diagrame de clase.

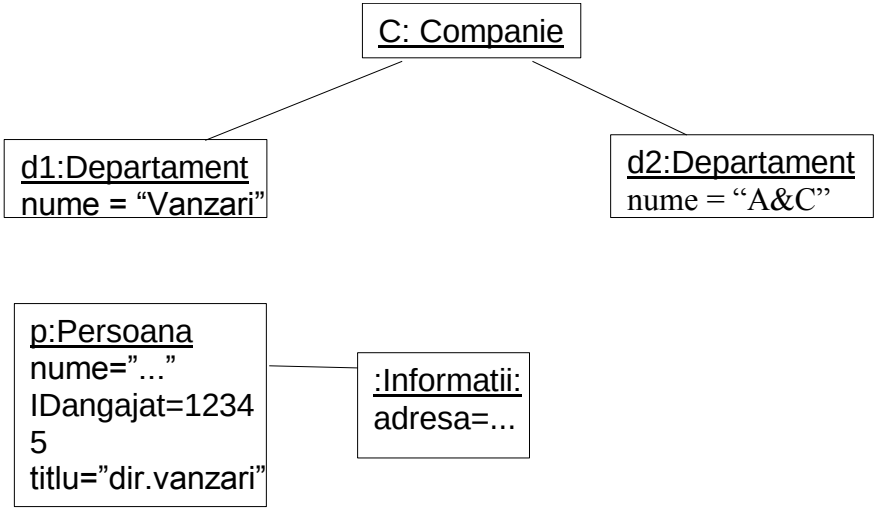
Ea arată o instanţă a unei stări a unui sistem la un moment dat şi conţine: obiecte, legături, eventual note şi constrângeri .

Obiectele pot fi :

- ***Actor*** este un obiect care operează asupra altor obiecte dar asupra căruia nu poate opera alt obiect;
- ***Server*** un obiect care poate opera pe alte obiecte;
- ***Agent*** un obiect care operează asupra altor obiecte şi asupra căruia pot opera alte obiecte. Un agent lucrează în numele unui actor sau altui agent.
- ***Legătura*** reprezintă o instanţă a unei relaţii de asociere şi defineşte o conexiune între instanţe. O legătură trebuie să aibă un furnizor (desemnează elementul care nu este afectat de o modificare) şi un client.
    - Un element furnizor poate participa în mai multe relaţii de legătură către diferiţi clienţi.

- Un element client poate participa numai într-o singură relaţie de legătură cu un furnizor.
- O legătură este o dependenţă unde furnizorul este un model şi clientul reprezintă instanţierea modelului care îndeplineşte substituirea parametrilor modelului;

- ***Note*** pentru formularea constrângerilor şi a altor comentarii sau alte explicaţii referitoare la clasificatorul utilizat;



Această diagramă foloseşte pentru vizualizarea, specificarea, construirea şi documentare structurii obiectelor şi în special pentru a arăta structurile de date.

Modelarea structurii obiectelor presupune:

- Identificarea mecanismului de modelat (o anumită funcţie sau comportamentul unei părţi a sistemului);
- Pentru fiecare mecanism, se identifică clasele, interfeţele şi alte elemente care participă la această colaborare;
- Se consideră un scenariu prin acest mecanism; se determină fiecare obiect care participă la mecanism;
- Selectarea obiectelor care au responsabilităţi de nivel înalt pentru fluxul lucrării;
- Identificarea precondiţiilor stărilor iniţiale şi postcondiţiilor stărilor finale;
- Specificarea activităţilor şi acţiunilor începând cu starea iniţială;
- Evidenţierea tranzacţiilor care conectează aceste activităţi şi acţiuni;
- Evidenţierea obiectelor importante implicate în fluxul de lucrări, cu evidenţierea schimbării valorilor obiectelor.

Modelarea operaţiilor presupune:

- Colectarea abstracţiilor implicate în operaţii (parametri, atribute ale claselor);
- Identificarea precondiţiilor stării iniţiale şi postcondiţiilor stării finale;
- Specificarea activităţilor şi acţiunilor începând cu starea iniţială;
- Folosirea ramificării dacă este necesar;
- Folosirea bifurcării şi reunirii pentru specificarea fluxurilor de control paralele.

4.2.2. Diagrama cazurilor de utilizare

Diagrama prezintă funcţionalitatea sistemului din punctul de vedere al interacţiunilor externe şi este utilizată în etapa de analiză.

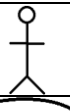
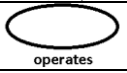
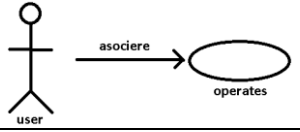
Elementele UML conţinute într-o astfel de diagramă sunt: cazuri de utilizare, actori, relaţii de utilizare (includere), relaţii de extindere, relaţii de generalizare şi de asociere.

Elementele din această diagramă sunt utilizate în primul rând pentru a defini comportamentul sistemului, a subsistemelor, a unei entităţi, fără a specifica structura internă.

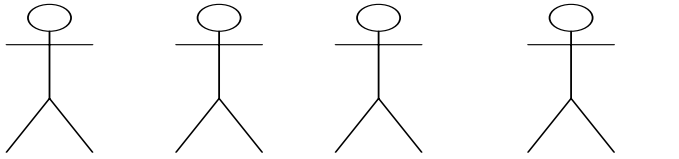
Diagrama cazurilor de utilizare prezintă relaţia dintre actori, cazuri de utilizare şi sistem.

Diagrama este un grafic care conţine actori, un set de cazuri de utilizare, posibile interfeţe şi relaţiile dintre acestea.

1. Componentele diagramei cazurilor de utilizare

Nr.	Denumire componentă	Simbol grafic
1	Actor	
2.	Cazul de utilizare	
3.	Relaţii	

1. Un actor este “cineva” sau “ceva” care interacţionează cu sistemul . Exemple:



Este o entitate internă sistemului (utilizator uman, dispozitiv fizic, un alt sistem), legată de acesta printr-un schimb de informaţie.

Defineşte un set al rolurilor pe care utilizatorii unei entităţi le pot avea în cadrul interacţiunii cu aceasta.

2. Cazuri de utilizare

- Orice caz de utilizare specifică o succesiune de acţiuni (evenimente), cu variantele lor, pe care entitatea le realizează atunci când interacţionează cu actorii săi.
- Comportamentul unui caz de utilizare este specificat prin descrierea setului de acţiuni.

Acces informaţii

Afişează comanda

Afişează cererea

3. Relaţii *Uses* şi *Extends*

Relaţiile de utilizare Uses (includere): Se folosesc atunci când există un comportament care se repetă identic în situaţia mai multor cazuri de utilizare.

Relaţiile de extensie - Extends : Sunt un tip special de generalizare pentru cazurile de utilizare folosite atunci când avem un caz de utilizare similar cu un alt caz, dar care face ceva în plus faţă de acesta.

Exemplul tipic pentru o astfel de diagramă este prezentat în figura 4.6. şi se referă la modelul cumpărării de produse online. Săgeţile pline din figuri reprezintă cazuri “copii” (⟶).

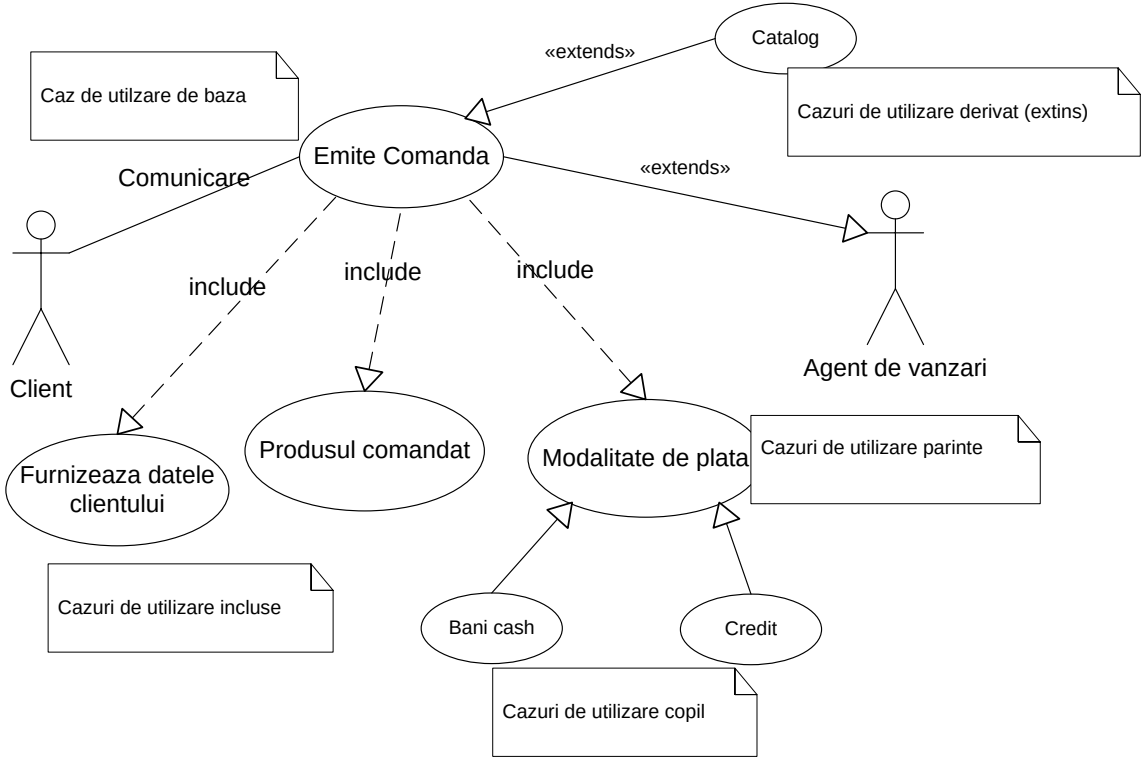


Fig. 4.6. Diagrama cazurilor de utilizare pentru comerţ on-line

Un alt exemplu de diagramă a cazurilor de utilizare este prezentat în figura 4.7. şi se referă la un sistem de evidenţă a studenţilor.

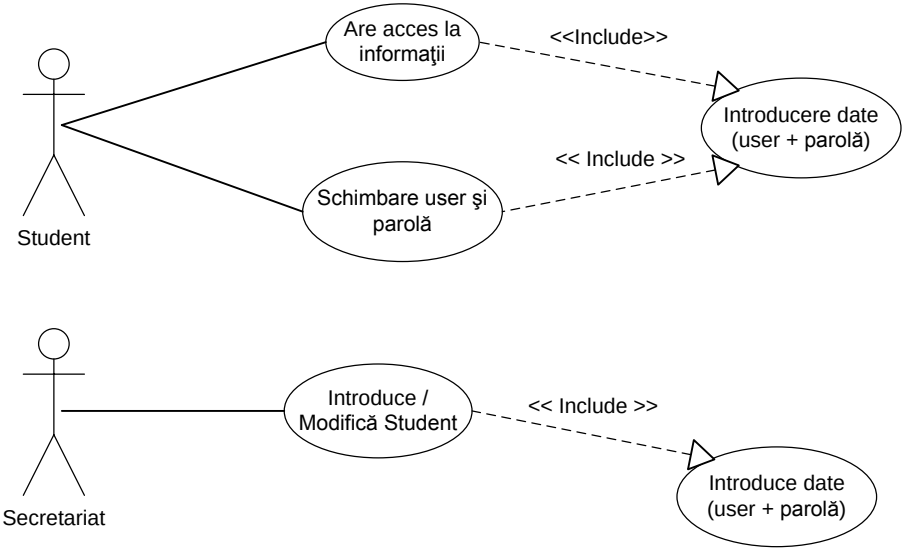


Fig. 4.7. Diagrama cazurilor de utilizare pentru un sistem de evidenţă a studenţilor

Un alt exemplu de diagramă este ilustrat în figura 4.8.

Este prezentat un caz de utilizare al unui sistem destinat urmării activităţii unei agenţii de turism.

Utilizatorul (actor în sistem) se autentifică şi poate deveni client al agenţii.

De asemenea un ofertant de servicii turistice se poate adresa agenţiei şi devine utilizator, dar de alt tip.

Operatorul (angajat al agenţii de turism), actor şi el în sistem, poate vizualiza clienţii existenţi, ofertele şi poate face rezervări.

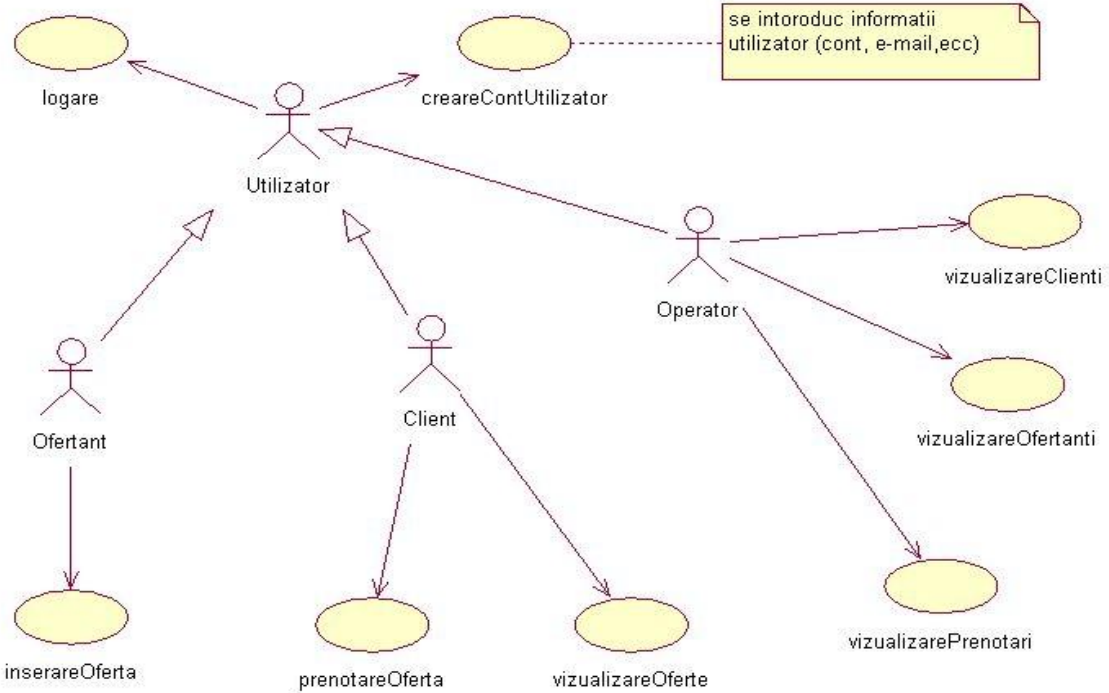


Fig. 4.8. Diagrama cazurilor de utilizare pentru o agenţie de turism

4.2.3. Diagrame de comportament

4.2.3.1. Diagrama de stare

Diagrama de stare modelează comportamentul unui singur obiect (instanţă a unei clase), a unui caz de utilizare, a unui actor sau a întregului sistem arătând de fapt comportamentul orientat - eveniment al obiectului.

O diagramă de stare UML este un graf care poate conţine următoarele elemente: stări, maşini de stări, tranziţii, evenimente, acţiuni şi bare de sincronizare.

O *maşină de stări* este o succesiune de stări prin care trece un obiect, pe durata sa de viaţă ca răspuns la evenimente.

O maşină de stare poate fi reprezentată prin intermediul următoarelor diagrame :

- diagrama de stare, caz în care accentul este pus pe comportamentul ordonat-eveniment al obiectului;
- diagrama de activitate, caz în care accentul este pus pe activităţile ce au loc în obiect.

O *stare* reprezintă o situaţie din viaţa unui obiect, putând satisface anumite condiţii, realizând activităţi sau aşteptând producerea unor evenimente.

O stare poate fi:

- Iniţială, sau de început – arată momentul de iniţiere a maşinii de stare sau al unei substări şi se reprezintă în diagramă printr-un cerc înnegrit;
- Intermediară – o stare prin care trece maşina de stare;
- Finală – maşina de stare a fost executată. Se reprezintă prin două cercuri concentrice, cercul din interior fiind înnegrit.
- Compusă – are în componenţă mai multe substări disjuncte. Se reprezintă printr-un dreptunghi împărţit pe orizontală în două zone.
- Concurenţială. Se reprezintă printr-un dreptunghi împărţit pe orizontală în trei zone.

Într-o diagramă, stările se reprezintă prin dreptunghiuri cu colţurile rotunjite.

Tranziţia reprezintă trecerea de la o stare la alta dintr-o diagramaă de stare. Se reprezintă printr-o săgeată.

Un exemplu de diagramă de stare, cu toate elementele descrise mai sus este prezentată în fig. 4.9 şi reprezintă maşina de stare pentru un obiect *Produs.* Diagrama de stare compusă pentru acelaşi obiect este ilustrată în fig.4.10.

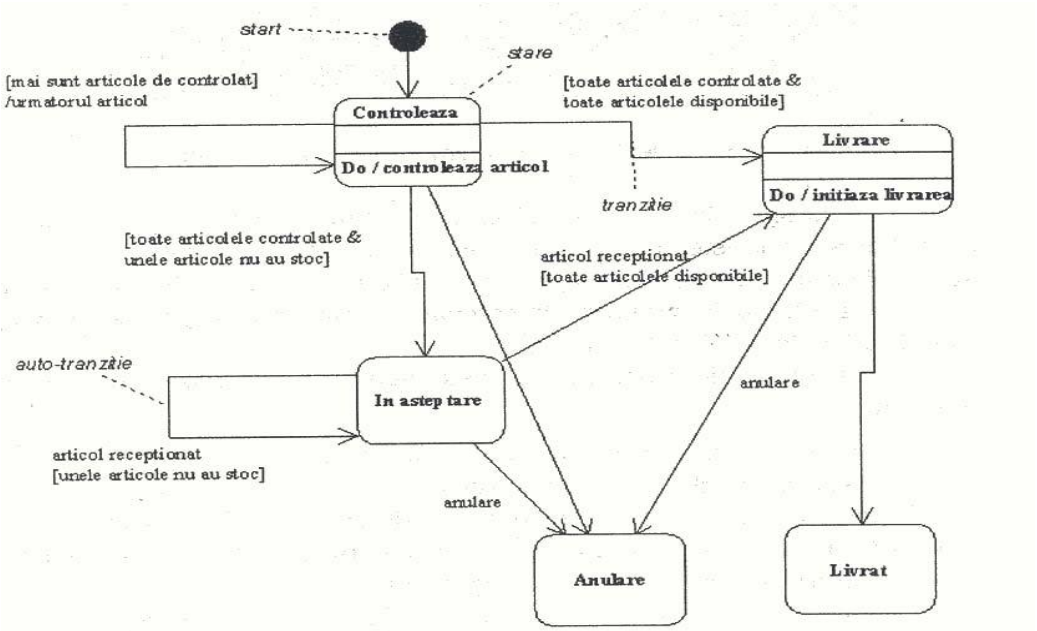


Fig. 4.9. Diagramă de stare

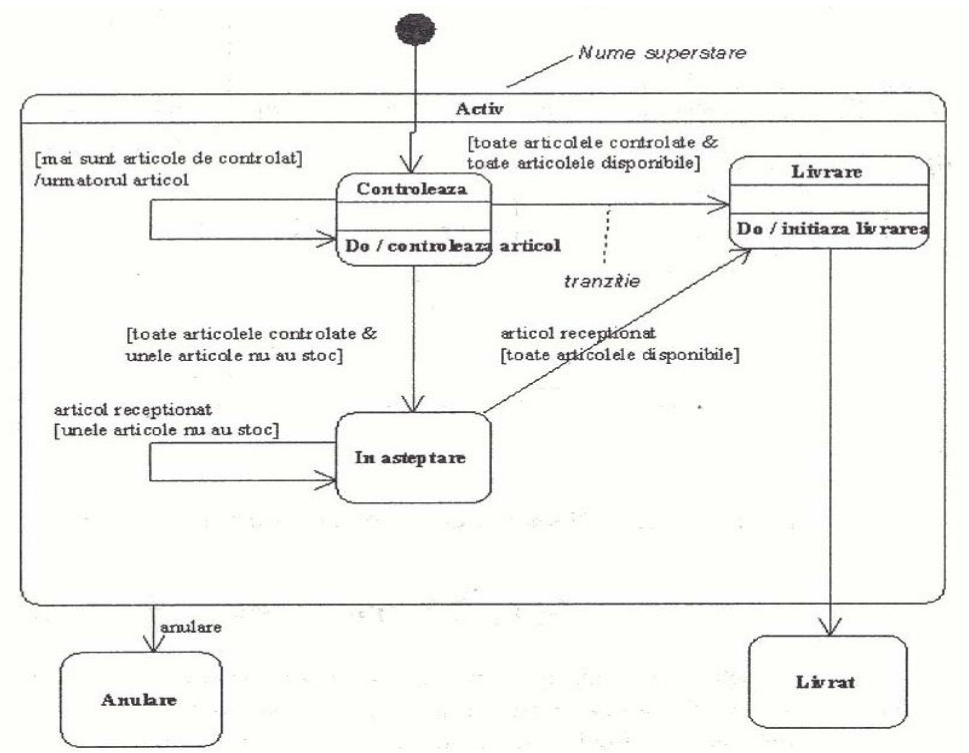


Fig.4.10. Diagrama de stare compusă

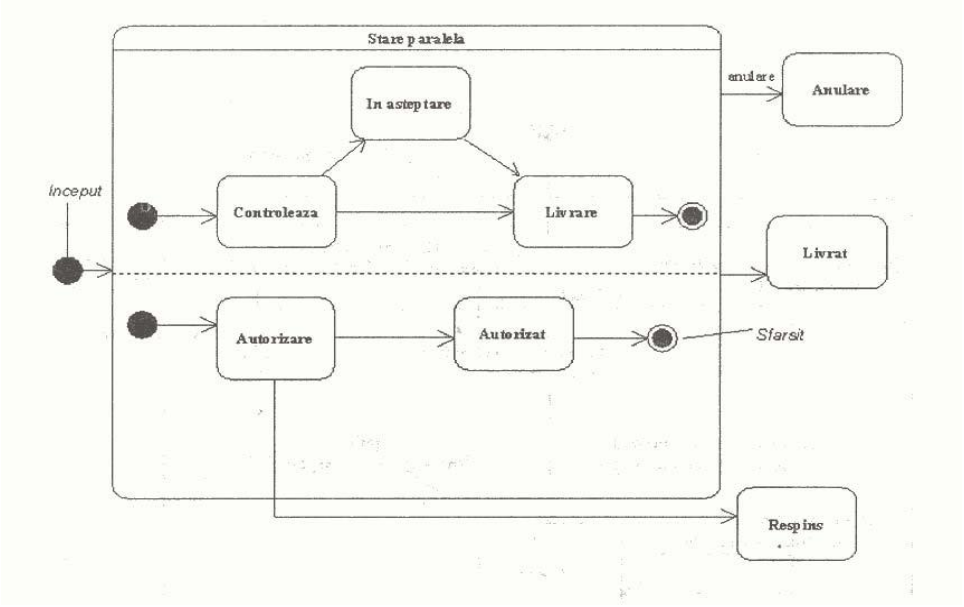


Fig. 4.11. Diagrama de stare concurenţială

Cazul exemplului discutat, agenţia de turism diagrama de stare compusă este ilustrată în figura 4.12.

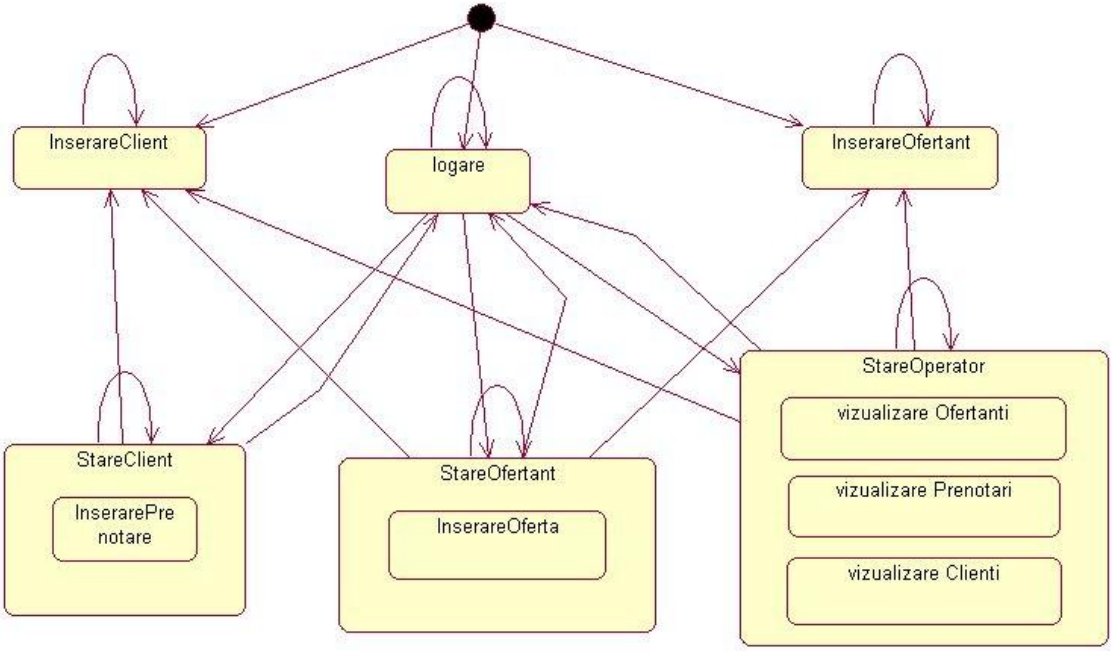


Fig. 4.12. Diagrama de stare compusă pentru agenţia de turism

4.2.3.2. Diagrama de activitate

O diagramă de activitate este un caz particular al diagramelor de stare UML, care defineşte un proces ce evoluează de-a lungul acţiunilor sale.

Această diagramă nu extinde semantica diagramelor UML dar defineşte forme prescurtate care se aplică modelării proceselor.

Este utilizată mai ales atunci când este necesară evidenţierea legăturii fiecărui serviciu sau a mai multor procese paralele.

Diagrama evidenţiază fluzul de control de la o activitate la alta.

O activitate rezultă dintr-o anumită acţiune (apelul unei operaţii, trimiterea unui semnal, crearea sau distrugerea unui obiect), sau evaluarea unei expresii care schimbă starea sistemului sau întoarce o valoare.

Diagrama de activitate este o variantă a unei maşini de stări, în care stările reprezintă performanţa activităţilor sau subactivităţilor.

O stare de activităţi reprezintă o subactivitate care are o anumită durată şi este constituită dintr-un set de acţiuni.

Activităţile reprezintă task-uri ce trebuie realizate de către calculator sau de o persoană, şi vor fi traduse în cadrul modelului prin intermediul unor metode specifice ataşate claselor care vor îngloba comportament de control.

Diagrama de activitate conţine următoarele elemente UML: *stări* (de activitate, de acţiune, de început, de sfârşit), *tranziţii*, *obiecte*, *decizii*, *semnale* (recepţionate sau transmise), *bare de sincronizare*, *culoare* (*swimlane*). Simbolurile lor grafice sunt:


Reprezintă starea iniţială, începutul procesului;


Reprezintă starea finală sau sfârşitul procesului (nu este absolut necesară figurarea stării finale, dacă aceasta reiese clar din contextul activităţilor reprezentate).


Starea de acţiune – stările prin care este posibil să treacă programul în funcţie de ceea ce are de executat.


Bloc de condiţionare ce împarte secvenţa în mai multe alternative


Bloc de bifurcare - este similar conectorului logic "**şi**". Firele de execuţie care pleacă din acest element se pot desfăşura paralel sau pe rând.


Bloc de reunire – element prin care se sincronizează firele de execuţie.

Bloc de sincronizare. Este folosit în cazul subsecvenţelor concurente pentru a sincroniza relaţia “producator-consumator” astfel încât consumatorul să utilizeze resursele disponibile la un moment dat.

Tranziţie –menţine stările active şi elementele modelului împreună.

Reprezintă dependenţe. Se pot trasa între oricare dintre elementele modelului.

Pentru o mai bună înţelegere a diagramelor de activităţi vom considera următorul exemplu:

Se consideră sistemul de gestiune al unei biblioteci. O persoană poate împrumuta sau restitui o carte, nu poate împrumuta nici o carte dacă are datorii către bibliotecă sau dacă are deja cinci cărţi împrumutate. Pentru acest caz diagrama de activitate este prezentată în figura 4.13.

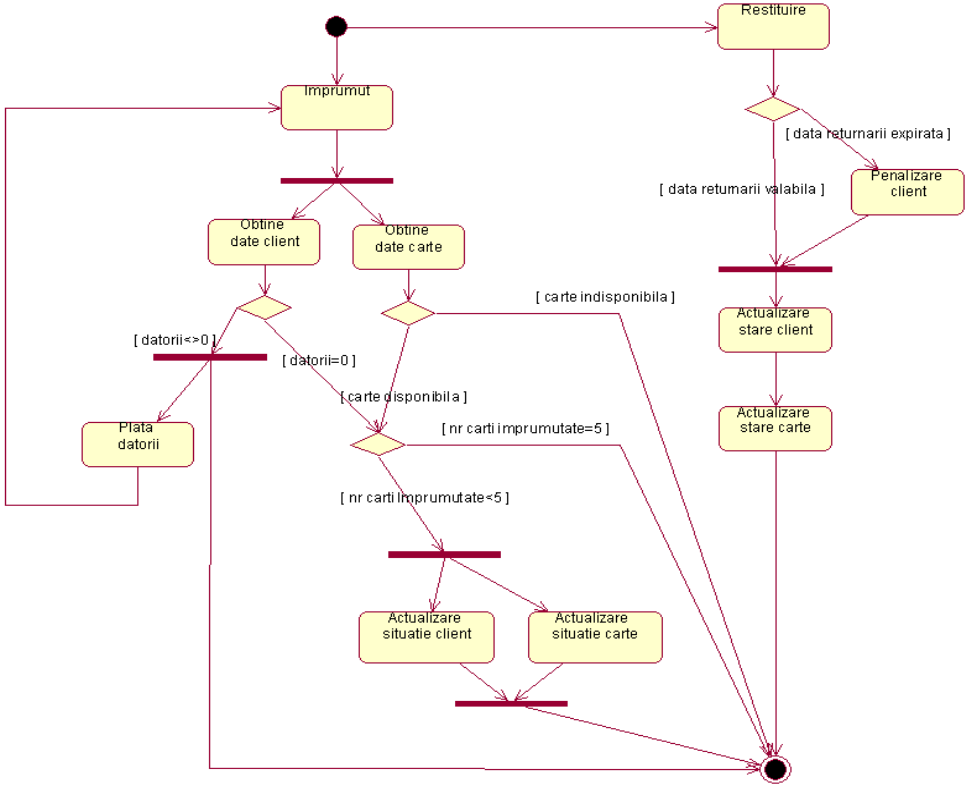


Fig. 4.13. Diagrama de activitate pentru sistemul “Biblioteca”

Pentru sistemul Agenţia de turism pe care l-am mai exemplificat anterior, diagrama de activitate este prezentată în figura 4.14.

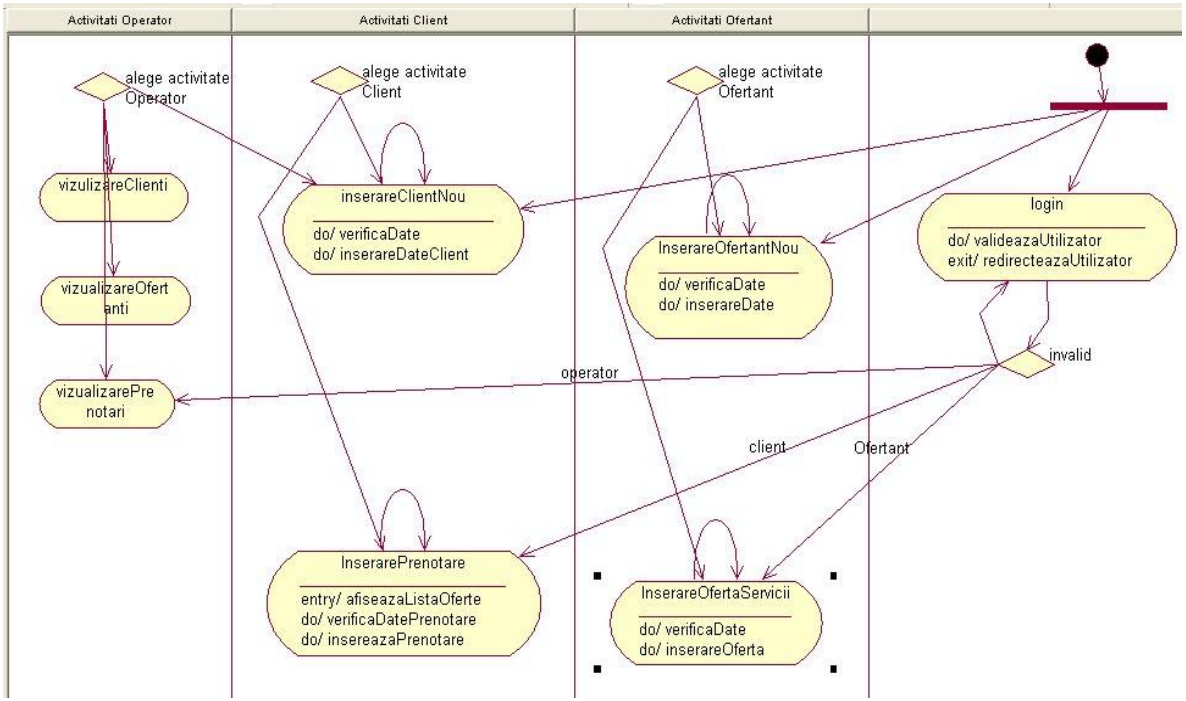


Fig. 4.14. Diagrama de activităţi pentru sistemul Agenţia de turism

Diagrama de activitate reflectă comportamentul mai multor obiecte din cadrul unui caz de utilizare. În figura 4.14 este prezentată de exemplu, activitatea de inserare de client nou şi de logare a acestuia în calitate de client.

Diagrama de activitate poate fi utilizată pentru:

1. Modelarea fluxului de activitate, activităţi aşa cum sunt văzute de actorii din sistem. Aceasta presupune:
   - selectarea obiectelor care au responsabilităţi de nivel înalt pentru fluxul de activitate;
   - identificarea precondiţiilor stărilor iniţiale şi postcondiţiilor stărilor finale;
   - specificarea activităţilor şi acţiunilor începând cu starea iniţială;
   - evidenţierea tranziţiilor care conectează aceste activităţi şi acţiuni;
   - precizarea obiectelor importante implicate în fluxul de activitate, cu evidenţierea schimbării valorilor.
2. Modelarea operaţiilor:
   - colectarea abstracţiilor implicate în operaţii (parametri, atributeale claselor);
   - identificarea precondiţiilor stărilor iniţiale şi postcondiţiilor stărilor finale;
   - specificarea activităţilor şi acţiunilor începând cu starea iniţială;

- folosirea ramificărilor, dacă este necesar;
- folosirea bifurcării şi reunirii pentru specificarea fluxurilor de control paralele.

4.2.3.3. Diagrame de interacţiuni

Diagramele de interacţiuni sunt modele care descriu modul în care grupuri de obiecte colaborează în acelaşi comportament.

De obicei o diagramă de interacţiuni capturează comportamentul unui singur caz de utilizare.

Diagrama prezintă un obiecte şi mesaje care se schimbă între acele obiecte în cazul de utilizare respectiv.

Există două tipuri de diagrame de interacţiune: diagramele secvenţiale şi diagramele de colaborare.

4.2.3.4. Diagramele secvenţiale

Diagramele secvenţiale sunt reprezentări alternative pentru interacţiuni între obiecte.

Ele reprezintă interacţiunile între obiecte din punct de vedere temporal, contextul obiectelor nefiind prezentat în mod explicit (ca în diagramele de colaborare), accentul concentrându-se pe exprimarea interacţiunilor.

Într-o diagramă secvenţială, un obiect este desenat ca un dreptunghi în capătul unei linii verticale întrerupte care reprezintă linia de viaţă a obiectului.

Diagrama de secvenţe, alături de diagrama de colaborare, surprinde colaborările între obiecte în cadrul unui anumit scenariu.

Obiectivul principal al acestei diagrame este acela de a exprima *fluxul* mesajelor între obiecte, în timp secvenţial.

Diagrama de secvenţe arată ordonarea în timp secvenţial a interacţiunilor între obiecte, iar în particular, aceasta arată obiectele care participă la o interacţiune şi succesiunea mesajelor care sunt schimbate.

Modelarea fluxului de control prin ordonarea în timp a mesajelor presupune:

- Stabilirea contextului interacţiunii (sistem, subsistem, operaţie, clasă, un scenariu al unui caz de utilizare sau colaborare);
- Identificarea obiectelor care joacă rol în acţiune;
- Stabilirea pentru fiecare obiect a duratei de viaţă în timpul interacţiunii. Pentru obiectele create şi distruse în timpul interacţiunii trebuie să se indice explicit, prin mesaj, acest lucru;

- Pentru fiecare mesaj începând cu primul (care iniţiază acţiunea) şi continuând cu celelalte în ordinea succesiunii, se prezintă proprietăţile (parametrii);
- Timpul şi spaţiul cerut pentru fiecare mesaj;
- Precondiţii sau postcondiţii pentru fiecare mesaj.

Pentru un flux complet de control se pot utiliza mai multe diagrame.

Diagrama secvenţială are două dimensiuni: una **verticală** care reprezintă timpul, şi una **orizontală** care reprezintă diferite obiecte.

Firele verticale întrerupte reprezintă durata de viaţă a unui obiect.

Mesajele indicate pe săgeţile ce intră într-un anumit fir nu sunt altceva decât metode ale clasei obiectului respectiv, care au fost apelate în cadrul obiectului cu rol de control.

Aşa cum am precizat deja în interiorul unei diagrame secvenţiale, obiectele sunt plasate la începutul diagramei. Dedesubtul fiecărui obiect se află o linie întreruptă, care este numită *linia de viaţă a obiectului*, şi care reprezintă durata de viaţă a obiectului în timpul interacţiunii.

Fiecare mesaj este reprezentat de o săgeată plasată între liniile de viaţă ale celor două obiecte care interacţionează.

Ordinea în care aceste mesaje sunt transmise este de la începutul către sfârşitul diagramei.

Fiecare mesaj are o etichetă cu numele mesajului; de asemenea se pot include argumente şi unele informaţii de control şi se pot folosi auto-delegaţiile.

Auto-delegaţia este un mesaj pe care un obiect şi-l transmite singur, reprezentat de o buclă întoarsă către linia de viaţă a obiectului.

Un element nou care apare în diagrama secvenţială este ***activarea***.

Activarea se petrece atunci când o metodă este activată, deoarece ea poate să fie în execuţie sau în aşteptare.

Un alt element care apare este *mesajul asincron*. Acest mesaj asincron poate executa unul din următorii paşi:

- creează un nou fir;
- creează un obiect nou;
- comunică cu un fir care este deja în execuţie.

În fig. 4.15 este prezentată diagrama secvenţială pentru acelaşi sistem exemplificat şi în diagrama cazurilor de utilizare şi destinat unei agenţii de turism.

Diagrama a fost realizată în mediul Visual Paradigm.

Dreptunghiurile verticale situate pe liniile de viaţă ale obiectelor reprezintă activarea metodelor.

Simbolurile X prezente la sfârşitul liniilor de viaţă indică ştergerea obiectului.

Săgeţile întoarse reprezintă auto-delegaţia.

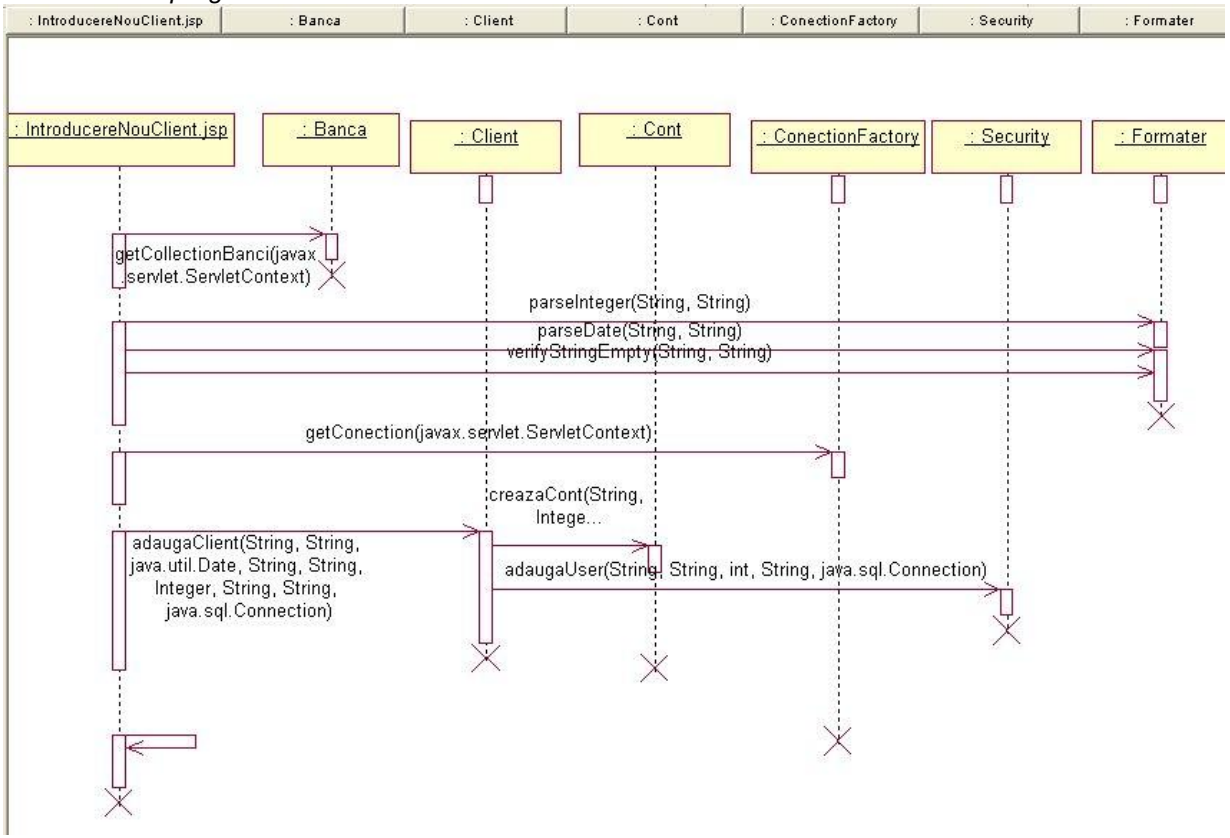


Fig. 4.15. Diagrama secvenţială pentru o agenţie de turism

În concluzie *diagrama secvenţială* trebuie să analizeze detaliat o anumită secvenţă liniară a fluxului de control din cadrul unui caz de utilizare, urmărind un anumit fir de activităţi din diagrama de activităţi, după ce clasele au fost modelate în detaliu .

4.2.3.5. Diagrame de colaborare

Diagrama de colaborare este un tip de diagramă de interacţiune, înrudită cu diagrama secvenţială, cu diferenţa că, în acest caz, accentul cade pe interacţiunea (comunicarea prin schimb de mesaje) între diferitele obiecte implicate într-un caz de utilizare şi nu pe succesiunea în timp a mesajelor .

Secvenţialitatea acestora poate fi totuşi modelată prin numerotare (nu prin dispunerea de-a lungul axei care simboliza durata de viata a unui obiect) .

Fiecare diagramă de colaborare realizează o vedere de ansamblu a legăturilor sau a relaţiilor structurale ce se stabilesc între obiectele şi entităţile obiectelor din modelul curent.

Se pot crea una sau mai multe diagrame de colaborare pentru fiecare pachet logic din model.

Elementele de bază ale acestui tip de diagramă sunt obiectele (instanţe ale claselor), legăturile (instanţe ale asocierilor definite între clase în diagrama claselor), şi mesajele care pot fi asociate bidirecţional legăturilor.

Un obiect are: stare, comportament şi identitate.

Fiecare obiect din diagramă indică o instanţă a clasei.

Simbolul de obiect este similar cu cel al clasei exceptând faptul că numele este subliniat.

Dacă se utilizează acelaşi nume pentru mai multe obiecte utilizate în aceeaşi diagramă, ele se presupun a reprezenta acelaşi obiect; pe de alta parte, fiecare simbol de obiect reprezintă în mod distinct un obiect.

Dacă există mai multe instanţe de obiecte ale aceleaşi clase, se poate modifica simbolul de obiect de exemplu, cu un click pe opţiunea Multiple Instances din Object Specification, al meniului mediului UML.

Mesajele dintre obiecte reprezintă comunicarea dintre acestea şi indică acţiunea în desfăşurare. Mesajul se scrie orizontal pe o săgeată ce face legătura dintre două obiecte.

Un mesaj se poate reprezenta în trei moduri: mesajul singur, mesajul însoţit de numărul secvenţei sau mesajul cu numărul secvenţei şi o etichetă.

Să luăm ca exemplu un sistem de gestionare a unei biblioteci departamentale şi să întocmim diagrama de colaborare pentru : scenariul de împrumut a unei cărţi şi scenariul de restituire a cărţii. Fig. 4.16. reprezintă diagrama de colaborare pentru scenariul de împrumut al unei cărţi, iar figura 4.17 prezintă diagrama de colaborare pentru scenariul de restituire a cărţii.

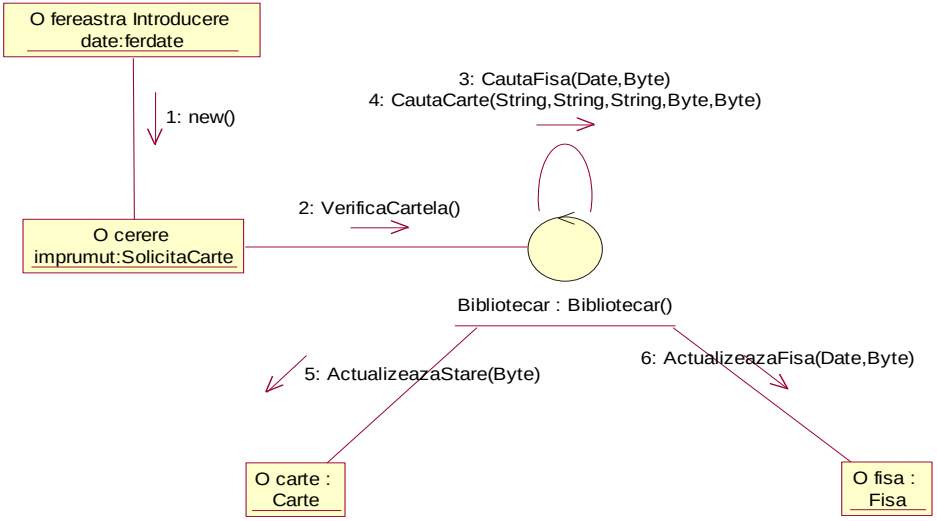


Fig. 4.16. Diagrama de colaborare pentru împrumut de carte dintr-o bibliotecă

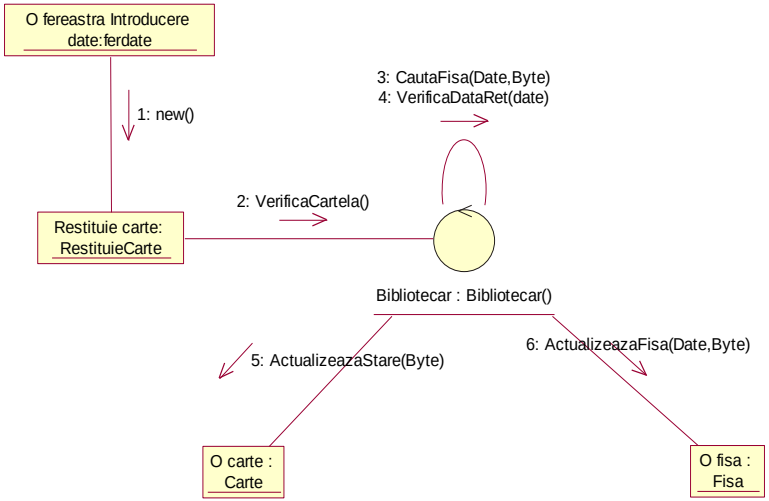


Fig. 4.17. Diagrama de colaborare pentru scenariu de restituire a unei cărţi

Pentru o mai bună înţelegere şi eventual o comparaţie între cele două tipuri de diagrame de interacţiuni (diagrama secvenţială şi diagrama de colaborare) figura 4.18. reprezintă diagrama de colaborare pentru sistemul destinat agenţiei de turism discutat anterior.

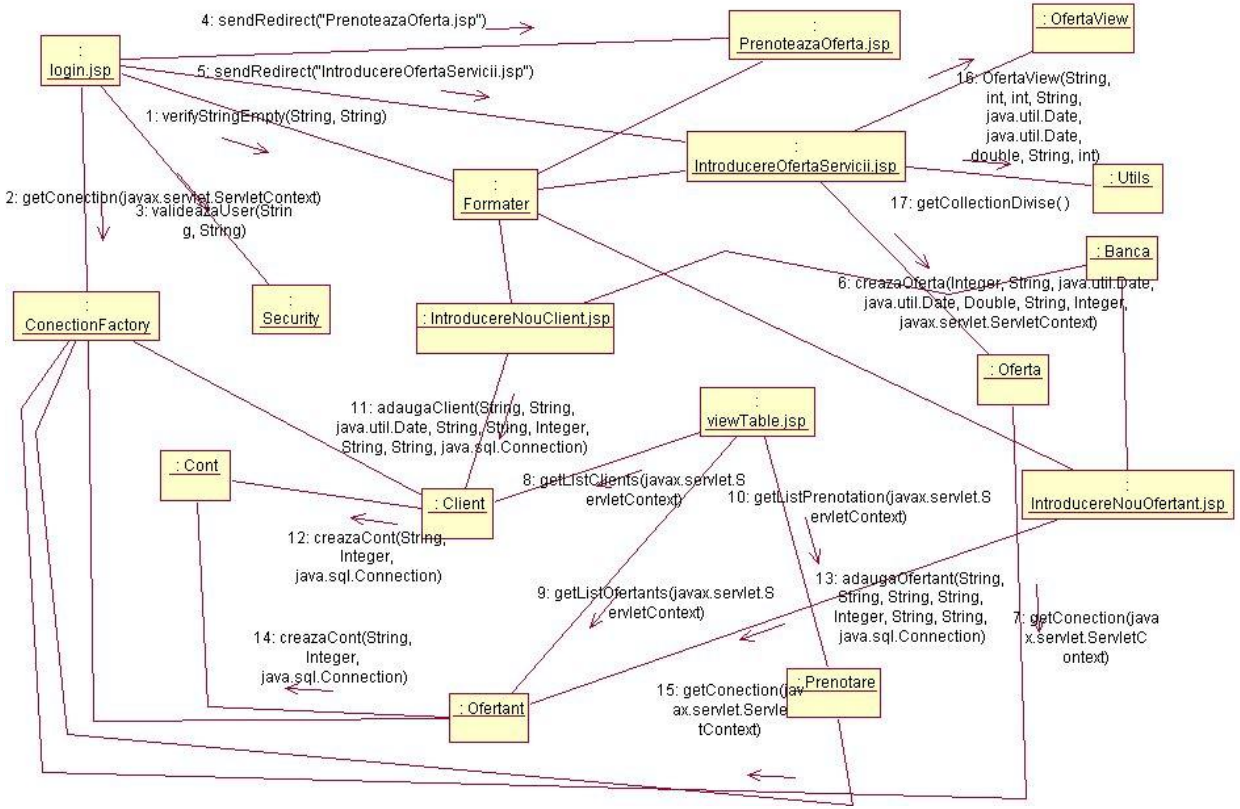


Fig. 4.18. Diagrama de colaborare pentru agenţia de turism

4.2.4. Diagrame de implementare

Diagramele de implementare, aşa cum le arată şi numele, relevă aspecte ale implementării, incluzând cod sursă şi execuţia.

Există două tipuri de astfel de diagrame şi anume: diagrame de componente şi diagrame de aplicaţie.

4.2.4.1. Diagrama de componente

Diagrama de componente este utilizată în modelarea aspectelor fizice ale sistemelor prezentând modul de organizare şi relaţiile de dependenţă între componente şi obiecte.

O astfel de diagramă are în componenţă următoarele elemente UML: componente, obiecte, interfeţe şi relaţii de dependenţă.

Componenta se va utiliza pentru a reprezenta software-ul utilizat de sistem (cod sursă, cod binar sau executabil) sau alte documente existente în sistem.

O instanţă a unei componente reprezintă o implementare run-time şi poate fi utilizată pentru a arăta implementarea unit-urilor care au identitate în momentul execuţiei aplicaţiei.

Componentele unui sistem, incluzând programe, DLL, etc., pot fi amplasate în noduri.

Pentru a figura dependenţele dintre diferite componente se utilizează o linie întreruptă de la o componentă la alta sau, de la o componentă la interfaţa altei componente.

Alte elemente care pot fi incluse sunt: note, legături, note ataşate.

Diagrama se utilizează în unul dintre următoarele scopuri:

- Modelarea codului sursă;
- Modelarea codurilor executabile;
- Modelarea bazelor de date fizice;
- Modelarea sistemelor adaptabile.

Fig. 4.19. prezintă o diagramă cu trei componente: *Clienţi*, care este o bază de date, *Comenzi*- bază de date, *Evidenţă comenzi*- aplicaţie. De asemenea în diagramă există şi un obiect, *Popescu Ion*, care este instanţă a clasei *Client*. Relaţiile dintre aceste elemente sunt relaţii de dependenţă.

Fig.4.20. prezintă o diagramă cu patru componente: *Comenzi*, *Terţi* şi *Produse*, care sunt fişiere şi *Evidenţă terţi*, care este aplicaţie.

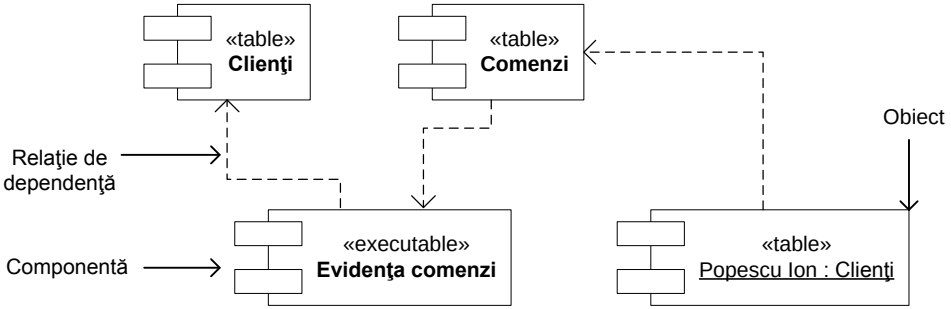


Fig. 4.19. Diagrama cu trei componente

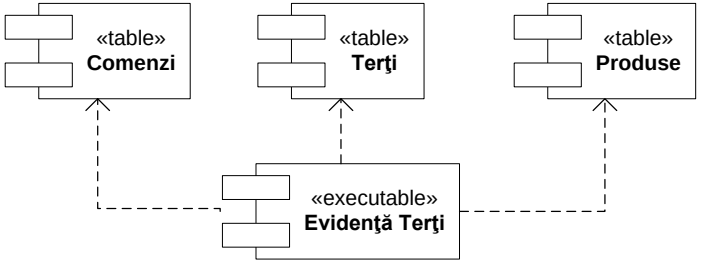


Fig. 4.20. Diagrama cu patru componente

4.2.4.2. Diagrama de aplicaţie (deployment diagram)

Diagramele de aplicaţie sau de desfăşurare arată structura fizică a sistemului hardware şi software, mai precis configurarea elementelor de procese run-time, a componentelor software, şi a proceselor şi obiectelor.

Au în componenţă următoarele elemente UML: noduri, componente, obiecte, relaţii de dependenţă, relaţii de asociere (comunicaţie), precum şi elemente ajutătoare: note, legături, etc.

Componentele au acelaşi rol şi funcţii ca în diagrama componentelor.

Nodurile sunt obiecte fizice care există în timpul execuţiei aplicaţiei şi reprezintă de obicei resurse de prelucrare.

Ele includ componente ale calculatoarelor, resurse umane sau resurse de procesare mecanică.

Diagramele de aplicaţie arată configuraţia elementelor de procesare în timpule execuţiei aplicaţiei, precum şi componente software, procese şi obiecte.

Ele sunt utilizate pentru a modela viziunea asupra desfăşurării statice a sistemului.

Componentele care nu există ca entităţi la execuţie (de exemplu un program care a fost compilat) nu apar în aceste diagrame, ci numai în diagrama componentelor.

O diagramă este reprezentată ca un graf în care nodurile sunt conectate prin relaţii de comunicare.

Nodurile pot conţine şi instanţe ale componentelor.

Acest lucru indică faptul că acele componente există sau rulează în acele noduri.

La rândul lor componentele pot conţine obiecte.

Componentele sunt conectate cu alte componente prin intermediul relaţiilor de dependenţă (linii întrerupte în diagramă), sau prin interfeţe.

Aceste interfeţe indică faptul că o componentă utilizează serviciile unei alte componente.

Componentele pot să migreze de la un nod la altul, iar acest lucru se reprezintă în diagramă cu ajutorul stereotipului << becomes>> pentru relaţii de dependenţă.

O diagramă de aplicaţie, pentru o aplicaţia cu clienţi şi furnizori este prezentată în fig. 4.21.

Diagramele de aplicaţie se folosesc în următoarele cazuri:

- Modelarea sistemelor cu software implantat hardware. Diagramele se utilizează pentru a modela dispozitivele şi procesele care compun sistemul.

- Modelarea sistemelor client-server. Un astfel de sistem este o arhitectură focalizată pe realizarea unei separări nete între interfaţa sistemului cu utilizatorul (de la client) şi datele permanente ale sistemului (de pe server).
- Modelarea sistemelor complet distribuite.

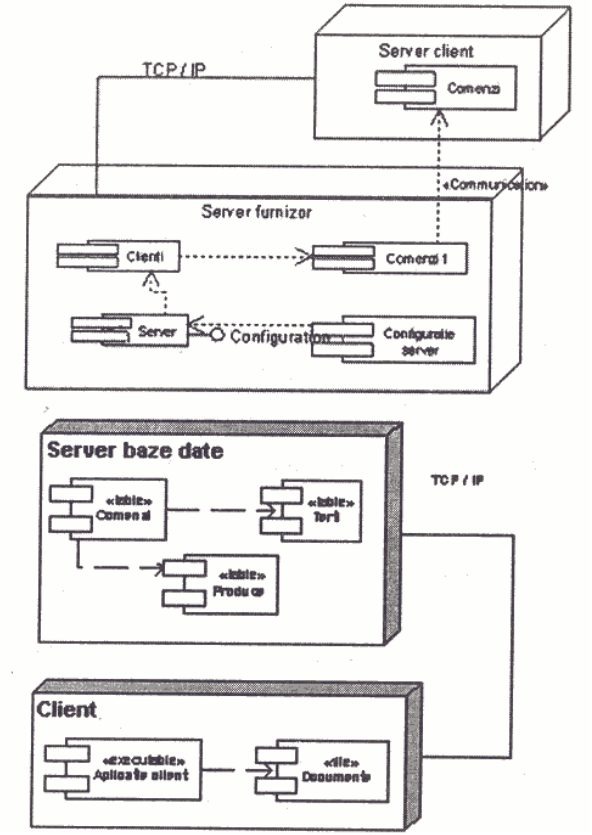


Fig. 4.21. Diagrama de aplicaţie

Principii de proiectare orientate pe obiect

Proiectarea orientată pe obiecte (OOD : Object Oriented Design) este o metodă de descompunere a arhitecturii unui sistem software cu scopul obţinerii modularizării acestuia.

OOD se bazează pe programarea orientată pe obiect, implicit pe obiectele pe care orice sistem sau subsistem le manipulează.

Se poate spune că OOD este relativ independent faţă de limbajul de programare folosit (Java, C++).

Pentru construirea unui design bun al sistemelor software, trebuie respectate mai multe principii de baza ale OOD, respectarea acestora putând fi privită ca o modalitate de rezolvare a acestor probleme.

5.1. Principiul Deschis – Închis - OCP (Open Close Principle)

Principiul Deschis – Închis, a fost formulat de către Bertrand Meyer, astfel:

"*Orice entitate software (clase, module, funcţii, etc) ar trebui sa fie **deschisă** pentru **extindere**, şi **închisă** pentru **modificare**."*

Prin utilizarea principiului deschis – închis se evită fragilitatea, rigiditatea si imobilitatea piesei de soft, proiectându-se module care "să nu se modifice niciodată".

În proiectarea sistemului de foloseşte **abstractizarea şi polimorfismul**.

Când specificaţiile sistemului se modifică, comportamentul modulelor se extinde prin adăugarea de cod nou, fără a interveni cu modificări în codul deja existent, care funcţionează.

Un modul software care respectă principiul deschis – închis are două caracteristici principale:

1. "**deschis pentru extindere**": comportamentul modulului poate fi extins. Se poate obţine astfel un comportament nou în conformitate cu noile cerinţe ale aplicaţiei.
2. **"închis pentru modificări"**: codul sursă al modulului este "inviolabil", nimănui nu i se permite să facă schimbări în cod.

Deşi la o prima vedere cele două cerinţe par contradictorii, totuşi ele pot fi simultan satisfăcute prin utilizarea **abstractizării**.

În limbajele de programare orientate pe obiecte, se pot crea abstractizări care sunt fixe din punct de vedere al codului, dar care totuşi reprezintă un grup nelimitat de posibile comportamente.

Abstractizările sunt clasele de bază abstracte, iar grupul nelimitat de comportamente este dat de toate clasele ce se pot deriva din acestea.

Deci, prin utilizarea abstractizării se îndeplineşte restricţia impusă de OCP: modulele sunt scrise astfel încât să poată fi extinse fără a fi modificate.

În **Fig. 5.1** se prezintă un exemplu foarte utilizat de proiectare a unei aplicaţii simple, în care atât clasa *Client* cât şi clasa *Server* sunt două clase concrete.

Clasa *Client* utilizează clasa *Server*.

Dacă se doreşte ca un obiect *Client* să utilizeze un alt obiect *Server*, atunci clasa *Client* trebuie modificată pentru a indica numele noii clase server, deci aceasta proiectare **nu respectă OCP**.

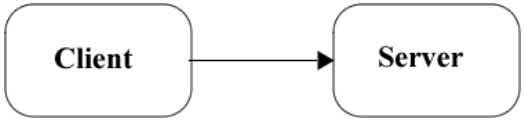


Fig. 5.1 Proiectare greşită Client-Server

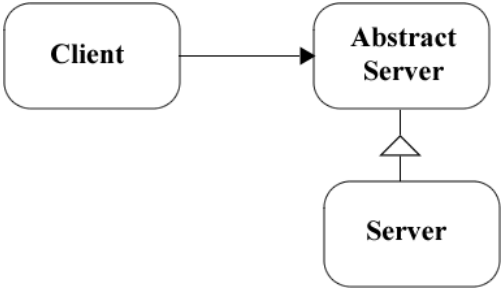


Fig. 5.2. Proiectare corectă Client-Server

Fig. 5.2 prezintă design-ul pentru aplicaţia **Server – Client** astfel încât să respecte principiul deschis – închis.

În acest caz, se utilizează o clasă abstractă *AbstractServer* , din care se derivează clasa concreta *Server* şi clasa *Client* depinde de această clasa abstractă.

Obiectele *Client* vor folosi obiecte ale unei clasei derivate *ServerA*.

Dacă se doreşte ca obiectele *Client* să folosească obiecte ale unei alte clase *Server*, atunci se creează, prin derivare, din clasa abstractă o nouă clasă concretă *ServerB*.

În acest mod, codul clasei *Client* rămâne nemodificat.

Pentru obţinerea unor noi comportamente necesare extinderii aplicaţiei, trebuie doar să se deriveze din clasa abstractă, clase concrete *Server* care implementează noi comportamente.

Închidere strategică

“Nici un program nu poate fi închis 100%.” .

Nici un program nu poate fi închis pentru modificări în mod total, deoarece, oricât de închis ar fi un modul, întotdeauna pot apărea situaţii pentru care nu a fost închis.

Având în vedere că, închiderea unui modul faţă de modificări nu poate fi completă, ea trebuie să fie **strategică**.

Închiderea strategică presupune ca designer-ul arhitecturii sistemului să abstractizeze partea care este cea mai susceptibilă a fi extinsă.

Închiderea explicită a sistemului la modificări se obţine utilizând **abstractizarea**.

Clasa abstractă furnizând metode care pot fi invocate dinamic şi în cadrul cărora se stabilesc politicile de decizie la nivel general.

O altă metodă care ajută la închiderea sistemului se bazează pe abordare “data-driven”, care presupune plasarea codului care se referă la deciziile de politică volatilă într-o locaţie separată, fie într-un alt fişier, fie într-un alt obiect, astfel că pe viitor modificările se vor face într-un număr minim de locaţii.

Reguli de proiectare folosite în OOD

Principiul deschis – închis este sursa unor reguli de proiectare folosite în OOD. Acestea se referă la variabile private şi variabilele globale folosite în program:

Toate variabilele să fie private

Aceasta este dintre cele mai întâlnite convenţii în OOD.

Variabilele unei clase trebuie să fie cunoscute doar în metodele definite în clasa respectivă.

Aceste variabile nu trebuie cunoscute de către alte clase, nici măcar de clasele derivate.

De aceea, este recomandat să fie declarate *private*, şi nu *public* sau *protected*.

- când variabilele dintr-o clasă se modifică, fiecare dintre clasele care depind de acestea ar trebui modificate.

Deci, nici o funcţie care depinde de o variabilă, nu poate fi închisă faţă de aceasta.

Datele constante (declarate *const* în C++ sau *final* în Java) pot fi publice sau protejate, fără a încălca principiul închiderii.

În OOD, închiderea celorlalte clase, inclusiv a claselor derivate, faţă de modificarea variabilelor dintr-o clasă, poartă numele de **încapsulare datelor.**

Avantaje ale încapsulării datelor:

- **securitatea datelor**: acestea nu pot fi modificate din exteriorul clasei în care sunt vizibile;
- **consistenţă**: prin modificarea variabilelor din alte clase, de către diferiţi utilizatori pot apărea situaţii de inconsistenţă, datorate unor modificări care nu sunt atomice.

Fără variabile globale

Argumentele împotriva variabilelor globale sunt similare celor împotriva variabilelor publice.

Nici un modul care utilizează o variabilă globală nu poate fi închis faţă de celelalte module care modifică respectiva variabilă.

Este posibil ca un modul să utilizeze variabila globală într-un mod neaşteptat pentru celelalte module care o mai folosesc.

Designerul trebuie să evalueze cât din închiderea aplicaţie este "sacrificată" în favoarea variabilelor globale şi să determine dacă avantajele oferite de utilizarea variabilelor globale compensează dezavantajele date de utilizarea lor.

Concluzii

În multe privinţe acest principiu este considerat esenţa proiectării orientate pe obiecte.

Avantajele care se obţin de pe urma folosirii acestuia sunt reutilizarea codului şi mentenabilitatea software-ului.

Un design bine gândit poate fi extins fără modificări, noile caracteristici ale sistemului adăugându-se prin completarea de cod nou, şi nu prin modificarea celui existent.

Mecanismele pe care se sprijină acest principiu sunt ***abstractizarea şi polimorfismul***.

Faptul ca se programează într-un limbaj orientat pe obiecte nu duce automat la concordanţă / conformitate cu OCP, ci este necesar ca designerul să aplice abstractizarea pe acele părţi ale programului care sunt susceptibile de a fi modificate.

Crearea abstractizărilor şi apoi derivarea claselor concrete din aceste abstractizări, duc la obţinerea unor module deschise pentru extindere şi închise pentru modificare.

Mecanismul care asigura crearea acestor clase derivate este moştenirea, iar principiul substituţiei al lui Liskov este cel care ghidează designul acestor ierarhii de clase.

5.2. Principiul substituţiei Liskov

Principiul substituţiei al lui Liskov (Liskov Substitution Principle, LSP) a fost enunţat în literatura de specialitate astfel:

Funcţiile care utilizează pointeri sau referinţe către clasele de bază, trebuie să poată utiliza obiecte ale claselor derivate din clasa de bază în mod transparent [Riel 1996].

Doar atunci când obiectele claselor derivate pot înlocui complet obiecte ale claselor de bază, codul poate fi reutilizat, atingându-se astfel scopul OCP (obţinerea unui cod reutilizabil, prin extindere

şi nu prin modificare); deci Principiul substituţiei al lui Liskov poate fi interpretat ca un mijloc de verificare a codului, din punctul de vedere al obţinerii unui design în acord cu OCP.

Acest principiu creează ierarhii de clase care se conformează OCP.

Să presupunem că există o metodă care nu se conformează LSP, atunci acea metodă utilizează un pointer sau o referinţă către clasa de bază, şi în acelaşi timp "ştie", conform presupunerii de mai sus, despre toate clasele derivate din clasa de bază.

O astfel de metodă încalcă OCP deoarece trebuie modificată ori de cate ori o nouă clasă derivată din clasa de bază este creată.

Exemple pentru ilustrarea LSP: Fie o clasă *Dreptunghi*:

```
public class Dreptunghi {
private double  width;
private double  height;
public double getHeight() {
  return height;
}
public void setHeight(double height) {
  this.height = height;
}
```

```
public double getWidth() {
  return width;}
public void setWidth(double width) {
  this.width = width;}}
```

Atât în C++ cât şi în Java, moştenirea se bazează pe modelul relaţional.

Modelul relaţional ISA descrie o relaţie strictă între clase, în care membrii unei clase formează o submulţime a unei alte clase.

Modelul ISA (“is a”) pune în evidenţă că un obiect (*RombA* din Fig. 5.3.) care aparţine unei subclase (clasa *Romb* din Fig. 5.3), aparţine simultan tuturor super-claselor (deci *RombA* este o instanţă a lui *Romb*, dar este în acelaşi timp instanţă şi a claselor *Patrulater* si *Poligon*)

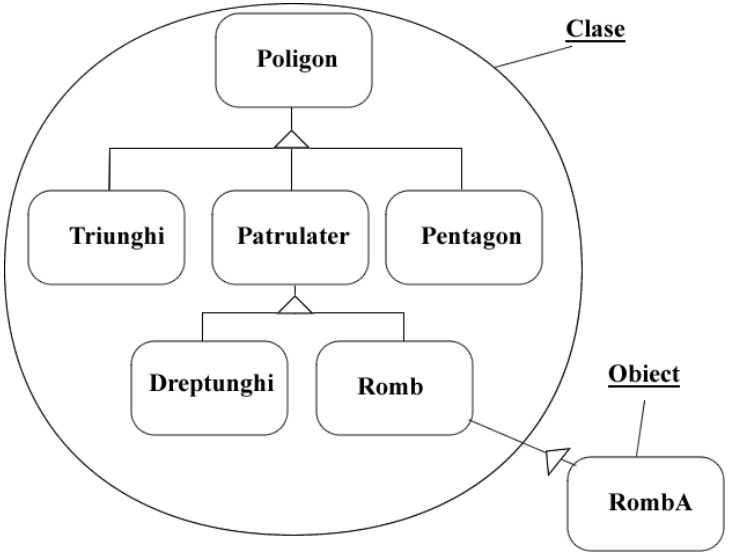


Fig. 5.3. Modelul relaţional ISA

Utilizarea modelului ISA este considerată ca fiind o tehnică de bază în Analiza Orientată pe Obiecte (Object Oriented Analysis, OOA).

În continuarea exemplului de mai sus, să presupunem că la un moment dat, în decursul dezvoltării unor noi aplicaţii, proiectantul are nevoie să utilizeze şi pătrate.

Deoarece un pătrat este un dreptunghi cu toate laturile egale, putem modela clasa *Pătrat* prin derivare din clasa *Dreptunghi*, în conformitate cu modelul ISA relationship.

Dar acest mod de a gândi, poate duce la anumite probleme pe care le vom întâmpina când trecem efectiv la scrierea codului.

Prima problemă, şi cea mai importantă, este legată de variabilele *heigth* şi *width*; pentru a defini un pătrat nu avem nevoie şi de lăţime si de înălţime (deci de amândouă), dar clasa *Pătrat* le va moşteni pe amândouă.

O problemă secundară este legată de folosirea eficientă a memoriei, mai ales dacă se utilizează sute de obiecte ale clasei *Pătrat*.

Dar trecând peste acest aspect al eficienţei memoriei utilizate, o problemă importantă este cea generată de existenţă celor două metode *setWidth* şi *setHeight*.

În primul rând, aceste metode sunt inadecvate pentru clasa *Pătrat*, deoarece dimensiunile unui pătrat sunt egale.

Iată deci o problemă de design, care totuşi poate fi depăşită dacă modificăm codul celor două metode astfel: când se setează "lăţimea" unui obiect *Pătrat*, "înălţimea" va fi ajustată în mod corespunzător, şi reciproc.

În acest mod un obiect *Pătrat* îşi păstrează proprietăţile matematice.

```
public class Pătrat extends Dreptunghi{
 public void setWidth(double width) {
      super.setWidth(width);
      super.setHeight(width);
 }
 public void setHeight(double height) {
      super.setHeight(height);
      super.setWidth(height);
 }     }
```

Dar dacă pentru a deriva, trebuie modificată clasa de bază, înseamnă ca aceasta are o problemă de proiectare.

Mai mult, acest lucru reprezintă o încălcare a OCP, deoarece programul ar trebui sa fie închis pentru modificări.

Consistenţa unui model

În acest moment, există două clase, care în aparenţă funcţionează corespunzător: *Pătrat* şi *Dreptunghi,* fiecare modelează corespunzător obiectele matematice ale căror nume le poartă (pătrat şi dreptunghi).

În aparenţă, se poate spune că sistemul are un comportament consistent.

Se poate trage concluzia că modelul este auto-consistent şi corect.

Dar această concluzie ar fi o eroare **deoarece un model care este auto-consistent nu este în mod necesar consistent şi cu toţi utilizatorii săi.**

Să considerăm următoarea funcţie:

```
void g(Dreptunghi r)
 {
 r.setWidth(5);
 r.setHeight(4);
 assert ((r.getWidth() * r.getHeight()) == 20) ;
 }
```

În mod evident aceasta metodă funcţionează corespunzător pentru *Dreptunghi*, dar pentru *Pătrat* generează o aserţiune falsă.

Aceasta este o problemă gravă.

Desigur, programatorul care a scris această metodă a făcut o presupunere îndreptăţită şi anume că modificarea lăţimii unui dreptunghi lasă înălţimea acestuia neschimbată.

Funcţia *g(Dreptunghi)* este un exemplu de funcţie care acceptă referinţe către *Dreptunghi* dar care nu funcţionează corespunzător pentru obiectele din clasa *Pătrat*.

Acest tip de funcţii încalcă principiul substituţiei al lui Liskov.

Validitatea nu este intrinsecă

Cazul expus mai sus impune o concluzie importantă: *validitatea unui model nu poate fi considerată semnificativă decât într-un anumit* ***context***.

Un model izolat nu poate fi apreciat din punct de vedere al validităţii lui.

Validitatea unui model se exprimă în termenii clienţilor săi.

Spre exemplu, în cazul de mai sus când s-a examinat versiunea finală a claselor *Pătrat* şi *Dreptunghi*, în mod izolat, s-a ajuns la concluzia că sunt consistente şi valide.

Totuşi, examinându-le din punctul de vedere al unui programator care a făcut respectiva presupunere în legătură cu clasa de bază, s-a ajuns la concluzia că modelul este greşit.

Deci, pentru aprecierea unui proiect, analiza trebuie să fie făcută într-un context şi nu în mod izolat.

Design-ul trebuie văzut din perspectiva utilizatorilor acestuia; ei lucrează pe baza unor presupuneri rezonabile asupra modului de funcţionare a modelului.

Trebuie analizat de ce modelul claselor *Pătrat* şi *Dreptunghi,* care părea corect, nu a funcţionat.

Nu este *pătratul* un *dreptunghi*? Nu a funcţionat relaţia modelului ISA, orice obiect al clasei *Pătrat* aparţine sau nu şi clasei *Dreptunghi*?

Un pătrat poate fi un dreptunghi, dar din punct de vedere strict matematic.

Un pătrat ca obiect al clasei *Pătrat* nu este un obiect *Dreptunghi*, deoarece comportamentul unui obiect *Pătrat* nu este consistent cu comportamentul unui obiect *Dreptunghi*, iar clasele tocmai acest aspect trebuie sa-l modeleze: **comportamentul**.

Prin comportament se înţelege **comportamentul public**, extrinsec, cel pe care se bazează clienţii, şi nu comportamentul privat, intrinsec al obiectului.

Spre exemplu, autorul funcţiei *g(),* de mai sus, s-a bazat pe faptul că pentru un obiect al clasei *Dreptunghi*, lăţimea şi înălţimea variază independent. Această independenţă a celor două variabile este un comportament public extrinsec pe care, probabil, contează şi alţi programatori.

toate clasele derivate trebuie să respecte comportamentul pe care clienţii îl aşteaptă de la clasa de bază pe care o utilizează.

Design prin contract

Există o strânsă relaţie între Principiul substituţiei al lui Liskov şi conceptul de “Design by contract”, aşa cum a fost definit de Bertrand Meyer.

Utilizând această schemă, metodele claselor declară precondiţii şi postcondiţii.

O metodă se execută dacă precondiţiile sunt adevărate.

La terminarea metodei, aceasta garantează că postcondiţiile vor fi adevărate.

Aplicând pe exemplul de mai sus, putem spune că postcondiţia metodei s*etWidth(double w)* în clasa *Dreptunghi* este:

```
assert((width == w) && (height == old.height))
```

Regula care se aplică precondiţiilor şi postcondiţiilor la derivare, aşa cum a formulat-o B. Meyer este următoarea:

Când se redefineşte o metodă în clasa derivată, precondiţia se înlocuieşte prin una mai "slabă", iar postcondiţia prin una mai "puternică".

Cu alte cuvinte, utilizarea unui obiect se face prin intermediul interfeţei clasei de bază şi utilizatorul cunoaşte doar precondiţiile şi postcondiţiile acestei clase.

Deci, obiectele claselor derivate nu trebuie să impună precondiţii mai puternice decât ale clasei de bază, ele trebuie să accepte orice precondiţie pe care clasa de bază o acceptă.

De asemenea, clasele derivate trebuie să se conformeze tuturor postcondiţiilor clasei de bază, comportamentul şi ieşirile lor nu trebuie să încalce nici una din constrângerile impuse de superclasă.

Postcondiţia din metoda *setWidth(double w)* din clasa *Pătrat* este mai puţin restrictivă decât cea din metoda s*etWidth(double w)* din clasa *Dreptunghi*, deoarece nu se conformează celei din superclasa, mai exact nu respectă şi condiţia: (height == old.height*)*.

Deci, metoda *setWidth(double w)* din clasa *Pătrat* încalcă contractul clasei de bază.

Concluzii

Respectarea OCP are ca efect obţinerea unor aplicaţii mai robuste, cu o mai bună mentenabilitate şi reutilizabilitate a codului.

Principiul substituţiei este o caracteristică importantă a acelor programe/aplicaţii care respectă OCP, deoarece tipurile derivate pot înlocui tipurile de bază astfel încât codul claselor de bază poate fi oricând reutilizat şi codul claselor derivate oricând modificat.

Substituţia claselor de bază cu obiecte ale claselor derivate este posibilă, deoarece clasele derivate respectă comportamentul extrinsec al superclaselor.

Obiectele subclaselor, conform LSP, se comportă într-o manieră consistentă cu “promisiunile” făcute de superclasă în API.

Astfel, claselor client li se furnizează un comportament stabil, pe care se pot baza.

5.3. Principiul Inversării Dependenţelor - DIP

Structura unui program, rezultată în urma respectării OCP şi LSP, este generalizată în principiul inversării dependenţelor (The Dependency Inversion Principle, DIP), formulat astfel:

A. Modulele de pe nivelele superioare nu trebuie să depindă de modulele de pe nivelele inferioare. Modulele de pe cele două nivele ar trebuie să depindă de nişte abstractizări.

B. Abstractizările nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstractizări.

Acest principiu este mecanismul de realizare a Principiului Deschis-Închis (OCP), deoarece prin aplicarea DIP se creează o arhitectură a sistemului închisă pentru modificări (abstractizările de pe nivelul superior) şi deschisă pentru extindere (clase concrete de pe nivelul inferior), deci modulele sunt reutilizabile şi stabile.

Metodele tradiţionale de dezvoltare a software-ului (programarea structurală) creează structuri în care modulele de pe nivelele superioare depind de cele de pe nivelele inferioare şi abstractizările depind de detalii de implementare.

Normal este ca nivelul superior să impună schimbări în nivelul inferior, modulele de pe nivelul superior să fie independente faţă de cele de pe nivelul inferior.

Astfel, **DIP impune** ca, pe de o parte, **într-o ierarhie de clase, clasa din care se derivează să nu cunoască nici una dintre subclasele sale**, iar pe de alta parte, **modulele care implementează detaliile depind de abstractizări, şi nu invers**.

Fie exemplul din **Fig. 5.4** al unui program alcătuit din trei module.

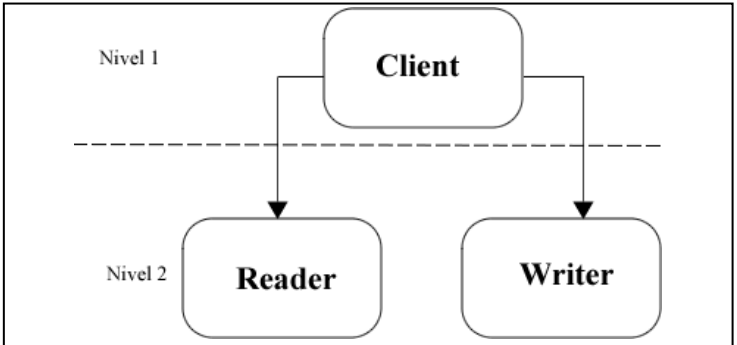


Fig. 5.4. Structura unui program alcătuit din trei module

Modul de pe nivelul 1, *Client*, implementează logica programului, iar modulele de pe nivelul 2 realizează anumite operaţi, conform deciziilor luate de modulul *Client*.

Aşa cum se observă şi din figură, modulul *Client* este dependent de modulele de pe nivelul 2.

Dacă modulele de pe nivelul 2 mai pot fi reutilizate în aplicaţii în care să îndeplinească aceleaşi funcţii, modulul de pe nivelul 1 este nereutilizabil el fiind strict dependent de modulele de pe nivelul inferior, cel mult poate fi reutilizat într-un context care să implice celelalte două module.

Dacă modulul *Client* poate fi modificat astfel încât să devină independent de celelalte module, atunci el poate fi reutilizat fără probleme, în alte programe de aceeaşi natură.

OOD ne pune la dispoziţie un mecanism pentru a face acest lucru: **inversarea dependenţelor**.

Dacă sistemului din Fig. 5.4 i se aplică principiul inversării dependenţelor, conform căruia modulele de pe nivele superioare nu depind de modulele de pe nivelele inferioare ci de nişte abstractizări, iar abstractizările nu depind de “detalii”, se obţine diagrama de clase din Figura 5.5.

Astfel, în această proiectare apar două clase abstracte "AbstractReader" şi "AbstractWriter". Modulul *Client* depinde de aceste două abstractizări. În acest fel a fost înlăturată dependenţa clasei Client faţă de clasele Reader şi Writer.

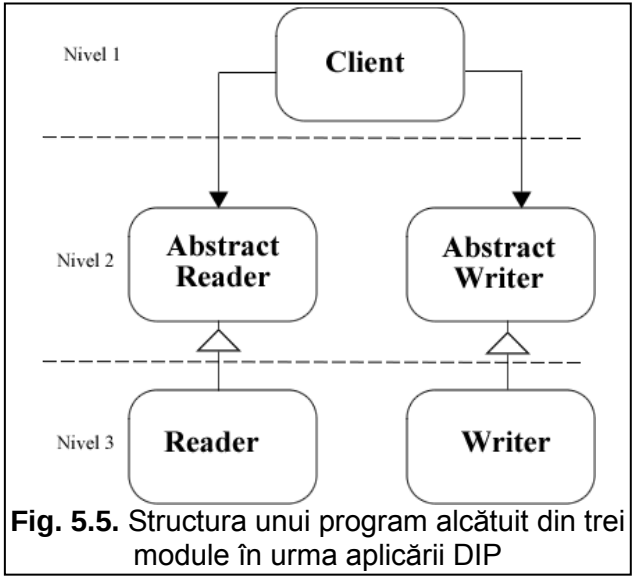


Fig. 5.5. Structura unui program alcătuit din trei module în urma aplicării DIP

Dependenţa a fost inversată, în sensul că atât clasa Client cât şi clasele Reader şi Writer depind de cele două clase abstracte.

Cu acest nou design, clasa Client poate fi reutilizată şi în alte contexte, independent de clasele concrete Reader şi Writer.

Se pot adăuga noi clase derivându-se din clasele abstracte AbstractReader, respectiv AbstractWriter; clasa Client nu va depinde de nici una din clasele nou create prin derivare.

În acest fel clasa Client a devenit **mobilă**.

În general, prin utilizarea principiul inversării dependenţelor se obţine o structură organizată pe nivele, cu modulele reutilizabile pe cele două nivele.

Modulele de pe nivelul inferior sunt reutilizate în forma librăriilor.

Organizarea pe nivele

Conform lui Booch [Booch, 1996]:

"[…] Toate arhitecturile orientate pe obiect, bine structurate au nivele clar definite, fiecare nivel oferind un set de servicii prin intermediul interfeţei sale."

O interpretare simplistă a acestei afirmaţii ar putea duce la realizarea unei arhitecturi în care dependenţele ar fi “pasate” de la un nivel la altul, având în vedere că **dependenţa este tranzitivă**.(**Fig. 5.6**).

Nivelul Deciziilor depinde de **Nivelul Mecanismelor**, care depinde de **Nivelul Serviciilor**, deci **Nivelul Deciziilor** depinde de **Nivelul Serviciilor** (**tranzitivitatea dependenţelor**).

O structură corectă este cea în care nivelele inferioare oferă servicii prin intermediul interfeţelor abstracte. (**Figura 5.7**).

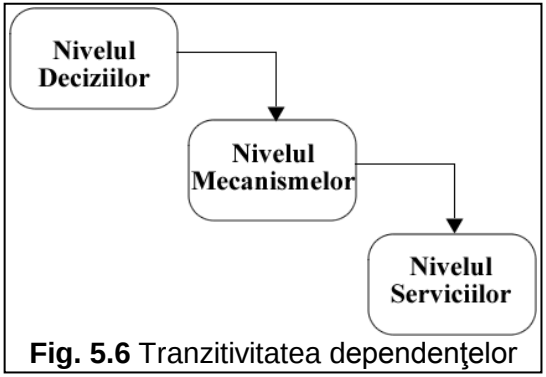


Fig. 5.6 Tranzitivitatea dependenţelor

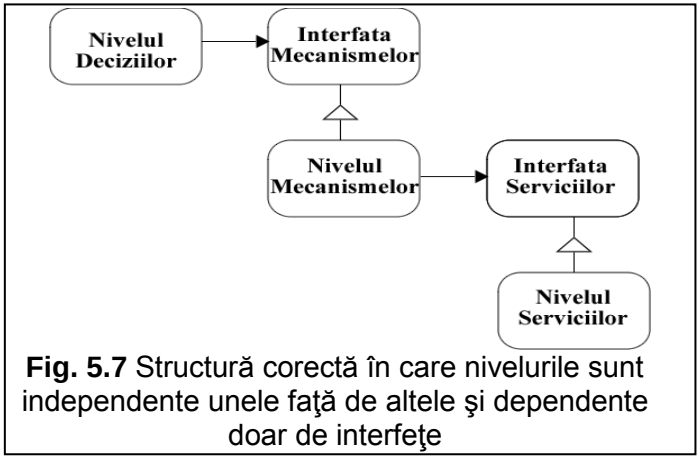


Fig. 5.7 Structură corectă în care nivelurile sunt independente unele faţă de altele şi dependente doar de interfeţe

Interfaţa reprezintă *totalitatea serviciilor pe care nivelul inferior le oferă nivelului superior.*

Deci, toate nivelele sunt independente unele faţă altele, şi dependente doar de clasele abstracte.

Cu alte cuvinte, conform acestui model, **Nivelul Deciziilor** este complet independent de nivelele inferioare, putând fi reutilizat în orice alt context în care se defineşte un nivel inferior bazat pe interfaţa **Nivelului Mecanismelor**.

Deci, inversând dependenţele, se creează o structură care are toate calităţile unui bun design: flexibilitate, durabilitate şi mobilitate.

Comparaţie între o arhitectura procedurală şi una OO

Comparaţia între o arhitectură procedurală şi una orientată-obiect are ca scop să pună în evidenţă avantajele şi dezavantajele fiecăreia.

În **Fig. 5.8** este schiţată arhitectura unui sistem procedural, în **Fig. 5.9** este schiţată arhitectura unui sistem orientat pe obiecte.

Dezavantajele unei arhitecturi procedurale sunt:

- nu există o separare clară a nivelului decizional al aplicaţiei de nivelul mecanismelor de implementare şi de cel al detaliilor;
- orice modificare în modulele de pe nivelul inferior duce la modificări în modulele de pe nivelul superior.

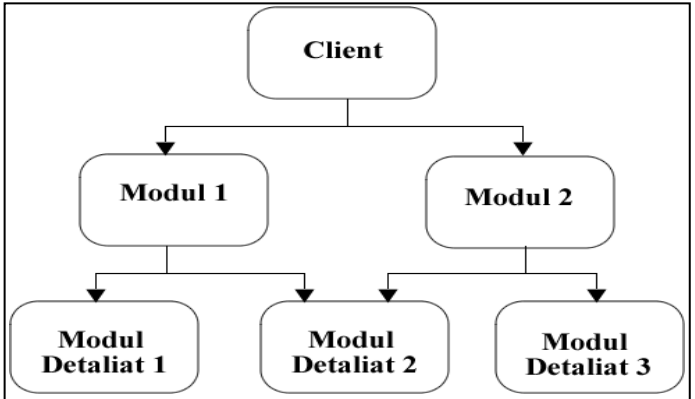


Fig. 5.8. O arhitectură a unui sistem procedural

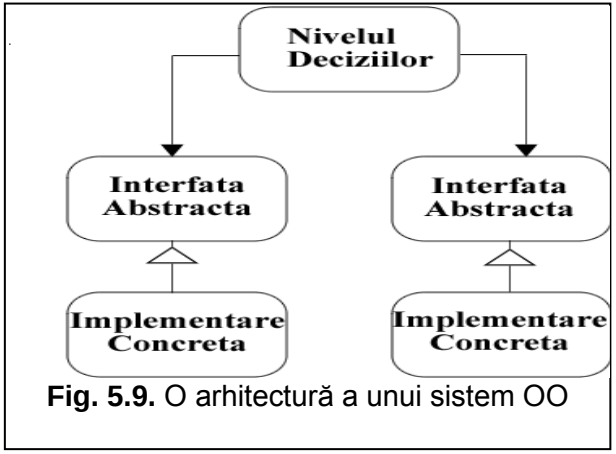


Fig. 5.9. O arhitectură a unui sistem OO

Aceste dezavantaje sunt înlăturate în cadrul unei arhitecturi OO, care respectă principiile OCP, LSP şi DIP:

- prin organizarea pe nivele se separă clar nivelul decizional de cel al implementării detaliilor,
- prin inversarea dependenţelor, obţinându-se o arhitectură alcătuită din piese de software flexibile, mobile şi uşor de reutilizat, deci toate atributele unui bun design.

După cum se observă, într-o arhitectură OO, modulele care au detalii de implementare nu depind de un alt modul, ci numai de abstractizări.

Reguli de proiectare folosite în OOD

Prin utilizarea Principiului Inversării Dependenţelor, s-a ajuns la câteva rezultate practice.

Utilizarea interfeţelor (claselor abstracte) pentru a evita legăturile directe între clasele concrete ***(Fig.5.10)***

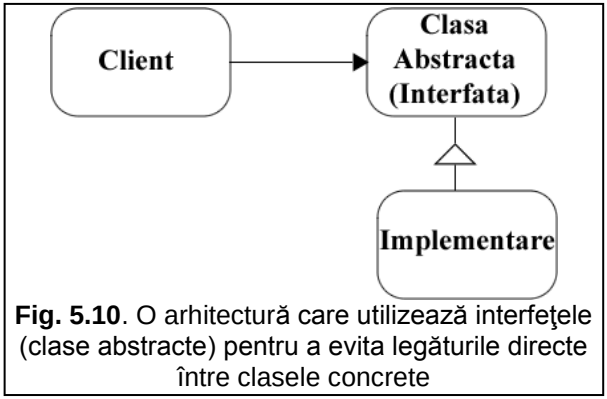


Fig. 5.10. O arhitectură care utilizează interfeţele (clase abstracte) pentru a evita legăturile directe între clasele concrete

Utilizarea interfeţelor contribuie la obţinerea unui **cod stabil** pentru clasa concretă Client.

O clasa abstractă este mai puţin probabil să fie modificată, pe de altă parte o clasă abstractă este mai uşor de extins/modificat.

Totuşi, trebuie avut în vedere că a modifica o abstractizare contravine OCP, care se bazează pe stabilitatea abstracţiunii.

Regulă de bază - Evitarea dependenţelor tranzitive prin utilizarea interfeţelor!

Concluzii

Principiului Inversării Dependenţelor este sursa multor beneficii ale tehnologiei orientate pe obiect; prin aplicarea lui se obţin module reutilizabile.

Este foarte important pentru construcţia codului, ca acesta să fie rezistent, robust şi elastic la modificări.

Deoarece abstractizările şi detaliile aplicaţiei sunt izolate unele de altele, codul este mult mai uşor de întreţinut (mentenabilitate crescută).

Cheia din principiul inversării dependenţelor este utilizarea abstractizării.

Detaliile de implementare sunt izolate unele de altele, între ele fiind interpuse abstractizările (care sunt clase stabile), astfel o modificare efectuată într-un modul ce implementează detalii nu se propagă în întreg sistemul.

Izolarea claselor care implementează detalii şi dependenţa acestora doar de abstractizări contribuie la obţinerea unor clase care pot fi utilizate, cu uşurinţă, în alte aplicaţii.

Privit din perspectiva principiilor prezentate în secţiunile anterioare, se poate spune că OCP este **scopul**, DIP (Principiului Inversării Dependenţelor) asigură **mecanismul** de îndeplinire a scopului, iar LSP (Principiul substituţiei al lui Liskov) este modalitatea de **verificare** a mecanismului.

Care este scopul unui design conform OCP ?

Obţinerea unor entităţi software care să fie deschise pentru extindere dar închise la modificări, în acest timp păstrându-se calităţile unui bun design: flexibilitate, mobilitate şi reutilizabilitate.

DIP specifică modalitatea de atingerea acestui scop:

- într-o ierarhie de clase, clasa din care se derivează nu cunoaşte nici una dintre subclasele sale
- modulele care implementează detaliile depind de abstractizări, şi nu invers.

* Iar prin LSP se asigură o măsura a calităţii moştenirii, verificându-se ca prin moştenire să se obţină subclase care au aceleaşi proprietăţi ca şi superclasa.

Aceste trei principii sunt strâns legate între ele: dacă este încălcat unul dintre principiile LSP sau DIP, atunci implicit este încălcat OCP.

5.4. Stabilitate. Principiul dependenţelor stabile

Aşa cum s-a arătat în cadrul subcapitolului *1.2. Probleme ale software-ului*, **interdependenţa** reprezintă una dintre cauzele pentru care un design este rigid, imobil şi dificil de reutilizat.

Totuşi, interdependenţa este necesară dacă modulele implicate în design "colaborează".

Ca urmare există tipuri de **dependenţă utile** şi tipuri de **dependenţă indezirabile**.

În acest paragraf se propune un model de design în care dependenţele sunt toate utile şi se descrie un set de indicatori pentru măsurarea conformităţii designului cu modelul propus.

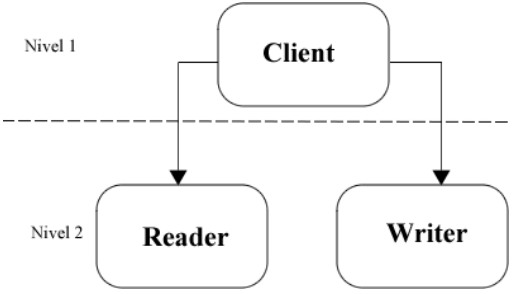
Aceşti indicatori măsoară stabilitatea software-ului.

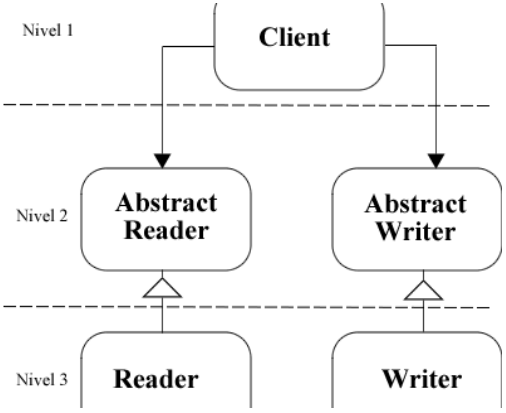
Stabilitatea este însăşi esenţa designului software.

Stabilitate şi Dependenţă. Dependenţe utile

Se consideră exemplul din *subcapitolul 5.3 (Principiul inversării dependenţelor),* cel al unui sistem alcătuit din 3 module: *Client*, *Reader*, *Writer* (Fig. 5.4).

Din analiza detaliată a acestei structuri, a rezultat că acest design este rigid, fragil şi dificil de reutilizat, din cauza faptului că modulul care implementează logica programului (modulul *Client*), este dependent de modulele de pe nivelul inferior. Pentru înlăturarea acestei dependenţe, se aplică DIP şi se obţine o structură (Fig. 5.5) în care modulul de pe nivelul superior depinde de nişte abstractizări. Designul astfel obţinut, este **robust, mentenabil şi uşor de reutilizat**.





Dependenţe utile

Totuşi, nu toate dependenţele au fost îndepărtate, ci doar cele care afectează calităţile programului.

-Dependenţele rămase sunt **nevolatile**, pentru că obiectul/modulul/clasa faţă de care se manifestă dependenţa este puţin probabil să se modifice.

- Probabilitatea ca aceste clase abstracte, AbstractReader şi AbstractWriter, să se modifice este foarte mică, spunem despre ele că au o **volatilitate scăzută**.

-Deoarece clasa Client depinde de module nevolatile, este puţin predispusă la schimbări.

- Această situaţie ilustrează foarte bine principiul deschis – închis: clasa *Client* este **deschisă la extinderi**, deoarece putem crea noi versiuni de clase concrete derivate din AbstractReader şi AbstractWriter pe care Client să le acţioneze, şi este **închisă pentru modificări** deoarece nu trebuie modificată pentru a face aceste extinderi.

Putem afirma în consecinţă, că o **dependenţă utilă** este o dependenţă faţă de un modul/clasă cu o volatilitate scăzută.

Cu cât volatilitatea este mai scăzută, cu atât dependenţa este "mai bună".

O dependenţă este indezirabilă când se manifestă faţă de un modul volatil.

Cu cât modulul/clasa faţă de care se manifestă dependenţa este mai volatil, cu atât dependenţa este "mai nedorită".

Stabilitatea

Volatilitatea unui modul depinde de mai multe tipuri de factori.

Spre exemplu, există programe care sunt publicate cu numărul de versiune; module care conţin numărul versiunii sunt volatile, deoarece sunt modificate ori de câte ori o nouă versiune este lansată.

Pe de altă parte, alte module sunt modificate mult mai rar.

Volatilitatea depinde şi de presiunea pieţei şi cererile clienţilor.

Un modul este sau nu posibil să fie modificat, dacă conţine sau nu ceva ce clientul doreşte să schimbe. Acest tip de factori sunt greu de apreciat.

Există, totuşi, un factor care influenţează volatilitatea şi care poate fi măsurat: **stabilitatea**. Stabilitatea, în sensul general acceptat, se defineşte ca fiind "**greu de modificat**".

Stabilitatea *nu este o măsura a probabilităţii de a modifica un modul, ci a dificultăţii de a-l modifica*.

Deci, modulele care sunt mai greu de modificat, sunt mai puţin volatile.

În exemplul amintit mai sus, clasele abstracte AbstractrReader şi AbstractWriter sunt stabile. Caracteristicile care fac aceste clase stabile, sunt:

- sunt **independente,** nu depind de ”nimic” şi ”nimic” nu poate forţa o schimbare a lor.

  *Se numesc clase **independente** acele clase care nu depind de nimic altceva.*

- de aceste clase depind alte clase (Client, Reader şi Writer depind de clasele abstracte din exemplu).

- Clasele derivate sunt clase dependente.
- Cu cât vor exista mai multe clase derivate, cu atât mai greu va fi să modificăm clasele de bază.
- Dacă vom dori să modificăm cele două clase abstracte, va trebui să modificăm şi clasele derivate.
- Deci, există motive serioase pentru care nu vom modifica cele două clase, îmbunătăţindu-le astfel stabilitatea.

*Clasele de care depind multe alte clase, se numesc **responsabile**.*

Clasele responsabile tind să fie stabile deoarece orice modificare a lor are un impact puternic asupra claselor dependente.

Cele mai stabile clase sunt cele care sunt independente şi responsabile, deoarece asemenea clase nu au nici un motiv să se modifice, şi au multe motive să nu se modifice.

Principiul dependenţelor stabile

Într-un proiect (design), dependenţa dintre pachete ar trebui să fie în sensul creşterii stabilităţii pachetelor. Un pachet ar trebui să depindă de pachete mai stabile decât el.

Un design nu poate fi complet static, un anume grad de volatilitate este necesar dacă se doreşte realizarea mentenanţei.

Acest lucru se obţine dacă designul se conformează *principiului închiderii generale* (Common Closure Principle, CCP).

Prin utilizarea acestui principiu, se creează pachete care sunt sensibile doar la anumite tipuri de modificări.

Aceste pachete *sunt proiectate să fie volatile*.

Modificările în aceste pachete sunt aşteptate şi prevăzute.

Nici un pachet, despre care ştim că va fi volatil, nu ar trebui să aibă între dependenţi pachete greu de modificat. În caz contrar, şi pachetul volatil va fi greu de modificat.

În conformitate cu principiul dependenţelor stabile pachetele care sunt proiectate să fie instabile (uşor de modificat) nu au ca dependenţi pachete care au o stabilitate mai mare decât a lor (mai greu de modificat).

Indicatori ai stabilităţii

O modalitate de a măsura stabilitatea unui pachet este numărarea dependenţelor care "intră" şi "ies" din pachet. Aceasta ne va ajuta la calcularea **stabilităţii poziţionale** a pachetului.

Se definesc următorii indicatori:

- *Ca* : **dependenţe aferente**: Numărul claselor din afara pachetului care depind de clasele din acest pachet;
- *Ce* : **dependenţe eferente**: Numărul de clase din cadrul pachetului care depind de clase din afara pachetului;
- *I* : **Instabilitatea**:

$$I = \frac{Ce}{Ca + Ce}$$

Acest indicator are valori în domeniul [0,1].

I = 0 indică un **grad maxim de stabilitate** a pachetului;

I = 1 indică un **grad maxim de instabilitate** a pachetului.

Indicatorii *Ca* şi *Ce* sunt calculaţi prin numărarea **claselor** din exteriorul pachetului în cauză care au relaţii de dependenţă cu clasele din interiorul pachetului.

Să considerăm următorul exemplul din **Fig. 5.11.,** în care săgeţile punctate reprezintă dependenţele între pachete.

Relaţiile dintre clasele pachetelor arată care este natura dependenţei, modul cum este implementată aceasta (moştenire, agregare, asociere).

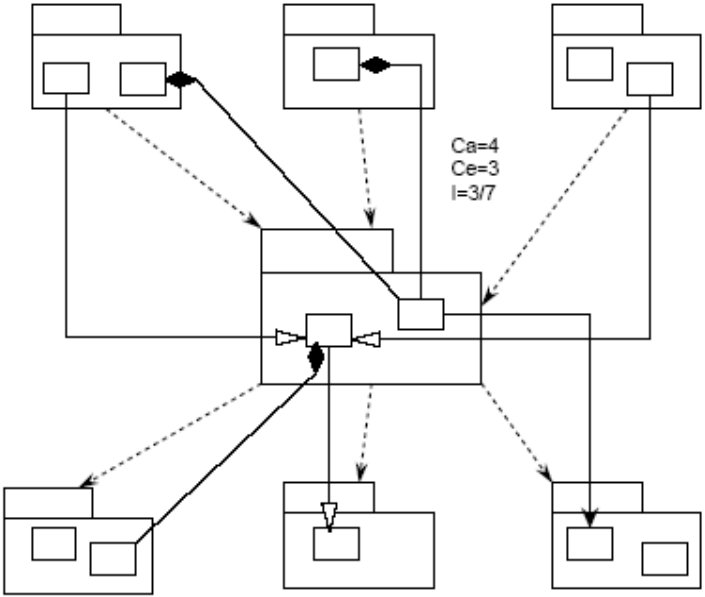


Fig. 5.11. Exemplu de calculare a gradului de stabilitate al unui pachet

Calcularea stabilităţii pachetului din centrul diagramei.

Observăm ca există 4 clase exterioare pachetului care sunt în relaţii cu clasele interioare pachetului, deci *Ca=4*. Mai mult, există 3 clase exterioare pachetului central de care depind clasele interioare, deci *Ce=3*.

$$I = \frac{Ce}{Ca + ce} = \frac{3}{7}$$

Când ***I=1***, nici un alt pachet nu depinde de pachetul curent, dar el depinde de alte pachete. Acesta este gradul maxim de instabilitate al unui pachet; pachetul este *iresponsabil* şi *dependent.*

Când ***I=0***, pachetul curent nu depinde de nici un alt pachet, dar de el depind alte pachete.

Este *responsabil* şi *independent*.

Un astfel de pachet are un grad maxim de stabilitate.

Din cauza pachetelor dependente, pachetul curent este dificil de modificat, şi nu depinde de alte pachete care ar putea forţa o modificare a acestuia.

Principiul dependenţelor stabile afirmă că indicatorul I al unui pachet ar trebui să fie mai mare decât indicatorul I al pachetelor dependente de el; I ar trebuie să descrească în sensul dependenţei.

Nu toate pachetele ar trebui să fie stabile. Dacă toate pachetele din sistem sunt stabile, atunci sistemul nu ar mai putea fi modificat. Sistemul trebuie proiectat astfel încât unele pachete să fie stabile iar altele instabile. **Fig. 5.12** arată situaţia ideala pentru un sistem de trei pachete.

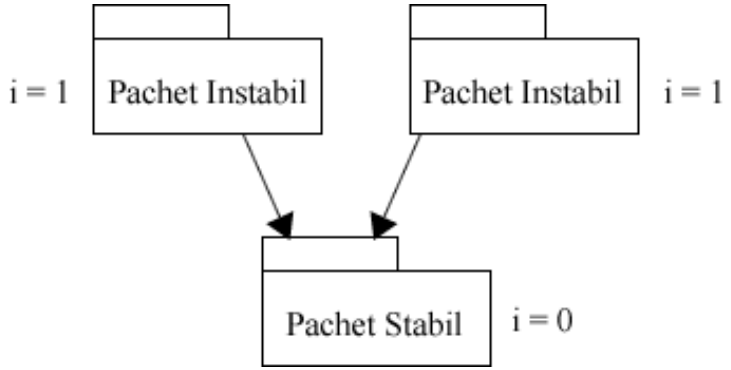


Fig. 5.12. Situaţia ideală din punct de vedere a stabilităţii pentru un sistem cu trei pachete

Pachetele care prezintă o probabilitate mai mare de a fi modificate sunt aşezate pe un nivel superior, iar cele care sunt stabile sunt aşezate la bază.

Pentru acest mod de aranjare a pachetelor, orice săgeată direcţionată în sus înseamnă o încălcare a principiului dependenţelor stabile.

O parte din software-ul unei aplicaţii nu trebuie să se modifice foarte des, şi anume logica de nivel înalt a aplicaţiei, deciziile de design.

Nu este de dorit ca aceste decizii arhitecturale să fie volatile, deci software-ul care încapsulează deciziile de nivel înalt, ar trebui plasat în cadrul unor pachete stabile.

Pachetele instabile ar trebui să conţină doar acele părţi de software, despre care se ştie că este probabil să fie modificate.

Dacă logica aplicaţiei este plasată în pachete stabile, atunci codul sursă al acestor pachete va fi dificil de modificat.

Acest fapt, ar putea face desing-ul inflexibil. OCP oferă soluţia pentru a face un pachet să fie flexibil şi totuşi să aibă un grad maxim de stabilitate (*I=0*),.

Conform acestui principiu este posibil şi este oportună crearea unor clase care să fie suficient de flexibile pentru a fi extinse fără modificări.

Acest lucru se realizează prin utilizarea claselor **abstracte**.

Principiul abstractizărilor stabile

Pachetele care au grad maxim de stabilitate ar trebui să fie maxim abstracte.

Pachetele instabile ar trebui să fie concrete.

Abstractizarea unui pachet ar trebui să fie direct proporţională cu stabilitatea sa.

Principiul abstractizărilor stabile (The Stable Abstractions Principle, SAP) stabileşte o **relaţie între stabilitate şi abstractizare** afirmând că un pachet stabil ar trebui să fie şi abstract astfel încât stabilitatea sa să nu-l împiedice să fie extins.

Pe de altă parte, un pachet instabil ar trebui să fie concret deoarece instabilitatea permite codului să fie cu uşurinţă modificat.

Dacă un pachet se doreşte a fi stabil, ar trebui să fie alcătuit din clase abstracte astfel încât să poată fi extins.

Pachetele stabile care pot fi extinse sunt şi flexibile şi nu constrâng design-ul.

Spre deosebire de **principiul inversări dependinţelor** care este un principiu pentru **clase**, **principiile dependenţelor stabile şi al abstractizărilor stabile** sunt principii care se aplică **pachetelor**.

Măsurarea gradului de abstractizare a unui pachet se face utilizând indicatorul *A,* care se calculează ca fiind raportul dintre numărul de clase abstracte dintr-un pachet şi numărul total de clase din pachet.

$$A = \frac{NumarClaseAbstracte}{NumarTotalClase}$$

Valorile indicatorului A sunt din intervalul [0,1].

Dacă ***A=0*** : înseamnă că un **pachet nu are clase abstracte**.

Dacă ***A=1*** : înseamnă ca **pachetul în cauză conţine numai clase abstracte**.

Relaţia dintre stabilitate, I, şi gradul de abstractizare, A (Fig. 5.13)

Pentru stabilirea relaţiei dintre stabilitate, măsurată de indicatorul *I*, şi gradul de abstractizare, măsurată cu indicatorul *A*, se realizează un grafic în care *A* se pune pe axa verticală, iar *I* pe axa orizontală.

Pachetele care au o stabilitate şi o abstractizare maximă se găsesc în punctul (0,1). Pachetele cu gradul de instabilitate cel mai mare şi concrete se găsesc în punctul (1,0).

Nu se poate impune tuturor pachetelor să se găsească fie la (0,1) fie la (1,0) deoarece pachetele au grade diferite de stabilitate şi abstractizare , aşa încât se admite că există un loc geometric al punctelor care definesc o poziţie rezonabilă pentru pachete în planul A/I.

Se deduce care este acest loc geometric, găsind ariile în care pachetele nu ar trebui să se găsească, adică ***zonele de excluziune***.

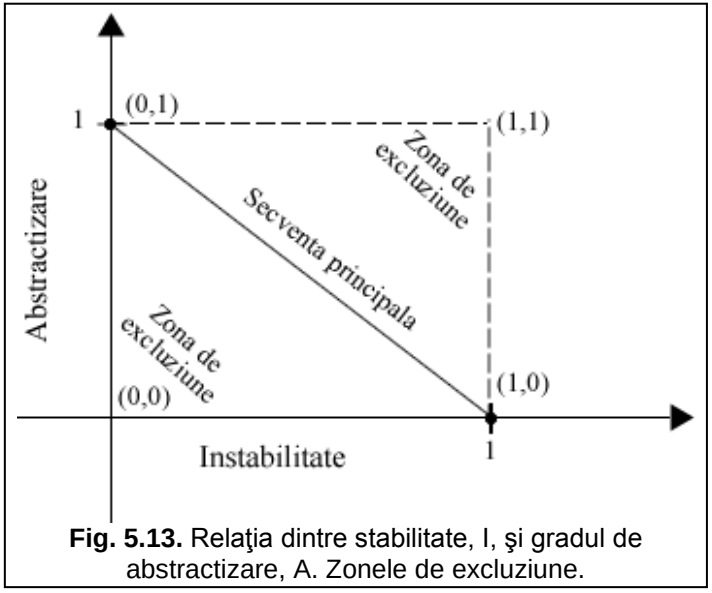


Fig. 5.13. Relaţia dintre stabilitate, I, şi gradul de abstractizare, A. Zonele de excluziune.

Se consideră un pachet în zona determinată de **A=0 şi I=0**, acesta va fi un **pachet stabil şi concret**.

Un astfel de pachet nu este de dorit deoarece este **rigid**, nu poate fi extins pentru că nu este abstract şi este foarte dificil de modificat pentru că este stabil.

Deci, nu este de dorit ca toate pachetele să se găsească în zona punctului (0,0). *Zona de vecinătate a punctului (0,0) este o zonă de excluziune.*

Se consideră un pachet în zona determinată de **A=1 şi I=1**.

Un astfel de tip de pachet este de asemenea indezirabil (poate chiar imposibil) deoarece are grad maxim de abstractizare şi totuşi nici un alt pachet nu depinde de el.

Şi acest tip de pachet este rigid, pentru că abstractizările sunt imposibil de extins.

Deci, un pachet in această zonă este fără sens.

Zona din jurul punctului (1,1) este o zonă de excluziune.

Fie pachetul **din A=0.5 şi I=0.5.** Acest pachet este parţial extensibil pentru că este parţial abstract; este parţial stabil deci extinderile nu duc la instabilitate maximă.

Un astfel de pachet pare **echilibrat**, deoarece *stabilitatea şi abstractizarea se compensează reciproc*.

Deci, **zona în care A=I nu este o zonă de excluziune.** Pe linia dintre (0,1) şi (1,0) se găsesc pachetele a căror abstractizare este compensată de către stabilitate.

Un pachet de pe această dreaptă nu este "prea abstract" pentru stabilitatea sa şi nici "prea instabil" pentru abstractizarea sa; are un număr "potrivit" de clase concrete şi abstracte, proporţional cu dependenţele aferente şi eferente.

Totuşi cel mai favorabil punct în care se poate găsi un pachet este la unul din cele două capete ale acestei drepte.

În practică, nu există un astfel de caz ideal.

Un pachet are caracteristici bune, dacă este plasat pe această dreaptă sau cât mai aproape de ea.

Distanţa faţă de Secvenţa Principală

Considerând că pe dreapta numită Secvenţa Principală, se găsesc pachetele care au caracteristicile ideale, putem crea un indicator care măsoară distanţa pachetului curent faţa de cazul ideal.

$$D = \left| \frac{A + I - 1}{\sqrt{2}} \right|$$

unde :

- D = Distanţa dintre punctul în care se găseşte pachetul faţă de dreapta numită secvenţa principală;
- A = gradul de abstractizare;
- I = gradul de stabilitate.

Acest indicator are valori cuprinse în intervalul [0, ~0.707]. (Putem normaliza acest indicator să aibă valori cuprinse în intervalul [0,1].)

Fiind dat acest indicator, un design poate fi analizat din punctul de vedere al conformării lui la **dreapta principală** . Pentru un pachet dat, D poate fi calculat. Orice pachet pentru care indicatorul D nu are o valoare apropiată de zero trebuie reexaminat şi restructurat.

Concluzii

Indicatorii descrişi mai sus măsoară conformitatea unui design faţă de un model de dependenţe şi abstractizări.

Aceşti indicatori încearcă să măsoare calitatea unui design din anumite puncte de vedere, fără a avea pretenţia de adevăr absolut.

În funcţie de dorinţe de la un design, se pot defini şi folosi diverşi indicatori.

Motto: "Şabloanele de proiectare software te ajuta să
înveţi mai mult din succesele altora, decât din propriile eşecuri"
[Mark Johnson]

Şabloane software de proiectare

6.1. Elementele unui şablon de proiectare software

În ingineria software, prin şablon de proiectare (în engleză "design pattern") se înţelege o soluţie, un tipar care se aplică la un anumit tip de probleme.

Un design pattern este o descriere sau un model pentru o soluţie a problemei

Un şablon nu poate fi transformat direct în cod sursă.

Şabloanele de proiectare în OOD (Object Oriented Design) arată relaţiile şi interacţiunile dintre clase sau obiecte, fără a detalia clasele şi obiectele care sunt implicate.

Scopul proiectării cu şabloane este acela de a face un bun design orientat pe obiecte, şi mai mult de atât, un **design reutilizabil** (ceea ce este mai important decât reutilizarea codului, dar adesea reutilizarea designului implică şi reutilizarea codului).

Calitatea unui design software este direct proporţională cu experienţa şi cunoştinţele anterioare ale celui care îl realizează.

Un expert reutilizează soluţii pe care le-a găsit în trecut şi care deja şi-au dovedit eficienţa în probleme asemănătoare.

Când un expert găseşte o soluţie pe care o consideră bună, o refoloseşte.

Şabloanele de proiectare rezolvă probleme specifice de design, făcând proiectul OO mai flexibil, elegant şi reutilizabil. „Design patterns” ajută designerii de software propunând nişte şabloane bazate pe experienţa anterioară.

Un designer care este familiarizat cu DP, le poate aplica imediat, pentru rezolvarea unor probleme specifice, fără a fi nevoit să le redescopere.

Şabloanele software sunt structuri care respectă principiilor de proiectare, deoarece sunt obţinute ca urmare a aplicării acestor principii (şabloanele sunt rezultate directe ale principiilor).

Cu alte cuvinte, prin utilizarea şabloanelor software în cursul proiectării unei arhitecturi OO, se obţine garanţia că sistemul respectă principiile prezentate în *cursul 5* , având şi calităţile unui ”bun design”: reutilizabil, flexibil şi robust.

Lucrarea de referinţă pentru şabloanele de proiectare este ”*Design Patterns: Elements of Reusable Object-Oriented Software*” [Gamma, 1997], adesea se fac referiri la ea sub denumirea de GoF sau Gang-Of-Four.

Autorii propun soluţii recurente la probleme comune în designul software.

În cartea GoF sunt propuse 23 de şabloane de proiectare într-o formă uşor de reutilizat; aceste şabloane încorporează experienţa autorilor în rezolvarea problemelor de OOD.

Şabloanele din aceasta carte au scopul de a descrie obiectele, clasele şi modul în care acestea comunică, scopul urmărit fiind rezolvarea unei probleme generală de design într-un context particular.

Un şablon are patru elemente esenţiale:

1. **Nume**: se doreşte ca numele să descrie în câteva cuvinte, o problemă de design pattern, soluţia şi efectele ei.
2. **Problema**: descrie situaţiile în care se aplică şablonul. Explică atât problema cât şi contextul acesteia.
3. **Soluţia**: descrie elementele care fac parte din design, relaţiile dintre ele, responsabilităţile fiecărui element şi colaborările dintre acestea. Soluţia nu descrie un proiect concret specific

unei situaţii sau o implementare concretă, deoarece şablonul are un caracter general, referindu-se la o diversitate de contexte. DP furnizează o descriere abstractă a unei probleme şi un aranjament general al elementelor componente (clase şi obiecte) care rezolvă problema descrisă.

4. **Efecte**: se referă la rezultatele aplicării şablonului. Acestea sunt foarte importante atunci când evaluăm alternativele şi pentru a estima costurile şi beneficiile aplicării şablonului. Efectele unui şablon se resimt şi în flexibilitatea, extensibilitatea şi portabilitatea unui sistem.

Şablonul identifică clasele şi instanţele participante, rolul acestora, colaborarea şi distribuirea responsabilităţilor între ele.

Tabel 6.1. Şabloanele de proiectare GoF

		SCOP (Ce face şablonul)		
		Creaţional	**Structural**	**Comportamental**
Cui se aplică şablonul ?	**Clasa**	Factory Method	Adapter	Interpreter Template Method
	Obiect	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsability Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

După scop autorii GoF au propus trei mari categorii de şabloane :

Şabloanele creaţionale

Abstract Factory – furnizează o interfaţă pentru crearea familiilor de obiecte dependente sau relaţionate, fără a specifica clasele lor concrete.

Builder – separă construcţia unui obiect complex de reprezentările sale astfel încât procese de cosctrucţie similare pot crea reprezentări diferite.

Factory Method – defineşte o interfaţă pentru crearea unui obiect, dar lasă subclasele să decidă care clasă va fi instanţiată.

Prototype – specifică tipurile de obiecte create folosind o interfaţă prototipizată şi crează noi obiecte copiind acel prototip.

Singleton – asigură că o clasă are numai o instanţă, dar furnizează un punct global pentru a o accesa.

Şabloanele structurale

Adapter – face conversia interfeţei unei clase la interfaţa dorită de client. *Adapter* permite claselor să lucreze împreună, ceea ce altfel n-ar fi posibil din cauza incompatibilităţii intefeţelor.

Bridge – decupleză o formă abstractă de implementarea sa asftel încât cele două pot fi schimbate independent.

Composite – compune obiecte din arborele de structură care reprezintă parţi sau întreaga ierarhie. *Composite* lasă clienţii să trateze fiecare în parte şi uniform obiectele şi compunerea lor.

Decorator – ataşează în mod dinamic responsabilităţi adiţionale obiectelor. Furnizează subclaselor o alternativă flexibilă pentru extinderea funcţionalităţilor.

Facade – furnizează interfaţă unificată cu un set de interfeţe într-un subsistem. Defineşte o interfaţă de nivel înalt care determină ca subsistemul să fie mai uşor de utilizat.

Flyweight – foloseşte tehnica ”sharing” (divizare în mod egal) pentru a lucra cu un număr mare de obiecte mici în mod efficient.

Proxy – furnizeză un înlocuitor pentru alt obiect în scopul de a controla accesul la el.

Şabloane de comportament

Chain of Responsability- permite cuplarea expeditorului unei cereri cu destinatarul ei dând mai mult decât unui sisngur obiect şansa de a utiliza acea cerere. Înlănţuie obiecte destinatare şi lasă cererea să treacă de-a lungul şirului atât timp cât un obiect o utilizează.

Command – încapsulează cererea ca un obiect lasând prin aceasta clienţi parametrizaţi cu diferite cereri, coadă sau tabel de cereri şi suportă operaţii care se pot detaşa.

Interpreter – dându-se un limbaj, definind o reprezentare pentru gramatica sa, interpretează propoziţii din acel limbaj.

Iterator – furnizează o cale de acces la elementele unui agregat de obiecte secvenţiale fără a expune reprezentarea sa internă.

Mediator – defineşte un obiect care încapsulează toată mulţimea de obiecte cu care interacţionează. *Mediator* promovează cuplajul slab păstrând obiectele pentru a putea fi referite în mod explicit şi lasă să poată fi schimbată independent interacţiunea lor.

Memento – fără a fi violată încapsularea, capturează şi externează starea internă a unui obiect, astfel încât, obiectul poate fi restaurat mai târziu în starea sa.

Observer - defineşte o dependenţă „ unu-multi" între obiecte, astfel încât când un obiect îşi schimbă starea toate obiectele din dependenţa sa sunt actualizate în mod automat.

State – permite unui obiect să-şi modifice comportamentul atunci când se schimbă starea sa internă. Obiectul va părea că-şi schimbă clasa.

Strategy – defineşte o familie de algoritmi, încapsulaţi fiecare şi-i face interşanjabili. *Strategy* lasă algoritmul să poată fi schimbat în mod independent de clienţii care îl utilizează.

Template Method – lasă subclasele să-şi redefinească câţiva paşi din algoritm fară a modifica structura algoritmului.

Visitor – permite să defineşti o nouă operaţie fără schimbarea claselor de elemente cu care aceasta operează.

Acestea sunt cele mai cunoscute DP, dar nu sunt singurele.

Lucrarea amintită mai sus a fost publicată pentru prima dată în 1995, de atunci au apărut şi alte şabloane mai mult sau mai puţin cunoscute, unele noi (tratând noi tipuri de probleme, apărute ca urmare a dezvoltării limbajelor OO), altele sunt variaţii ale şabloanelor amintite mai sus (aplicaţii ale şabloanelor GoF pe cazuri particulare).

6.2. Cum rezolvă şabloanele problemele de proiectare

Şabloanele de proiectare rezolvă o serie întregă de probleme curente cu care se confruntă proiectanţii de sisteme orientate pe obiecte, în diferite moduri.

Programele orientate pe obiecte sunt constituie, evident din obiecte.

Un obiect „împachetează”, asamblează la un loc, atât datele cât şi procedurile care operează cu acele date.

Procedurile se numesc *metode* sau *operaţii*.

Un obiect execută o operaţie în momentul în care primeşte o *cerere* (sau *un mesaj*) de la un client.

Cererile constituie singura modalitate de a determina un obiect sa execute o operaţie.

Operaţiile constituie singura cale de a schimba datele interne ale obiectului.

Din cauza acestor restricţii se spune că starea internă a obiectului este *încapsulată;*ea nu poate fi accesată direct, iar reprezentarea sa este invizibilă în afara obiectului.

Partea cea mai dificilă a proiectării orientată-obiect este descompunerea proiectului în obiecte.

Sarcina este dificilă deoarece trebuie luaţi în consideraţie mulţi factori caum ar fi: încapsulare, granularitate, dependenţă, flexibilitate, performanţă, evoluţie, reutilizare, şi mulţi alţii.

Ei tot influenţează descompunerea şi câteodată sunt în contradicţie.

Multe obiecte provin din modelul de analiză, dar adesea proiectele OO au clase care nu au nici o legătură cu lumea reală.

Unele dintre acestea sunt clase de nivel inferior, cum ar fi tabelele, altele, sunt de nivel mai ridicat, cum ar fi *Compositor*, şablonul care introduce o abstractizare pentru tratarea uniformă a obiectelor care fizic nu sunt la fel.

Modelarea strictă a lumii reale conduce către un sistem care reflectă astăzi lumea reală dar nu în mod necesar şi mâine!

Abstractizările care decurg din proiectare sunt cheia realizării unui proiect flexibil.

Şabloanele de proiectare ajută la identificarea celor mai puţin evidente abstractizări şi obiecte care le capturează.

De exemplu, obiectele care reprezintă un proces sau un algoritm nu apar în natură, dar sunt parte crucială pentru proiecte flexibile.

Şablonul *Strategy* ( ) descrie cum se implementeză familii interşanjabile de algoritmi.

Şablonul *State* ( ) reprezintă fiecare stare a unei entităţi sau obiect.

Aceste obiecte sunt rareori găsite în timpul analizei sau în stadii timpurii ale proiectării.

Ele sunt descoperite mai târziu în cursul derulării proiectării când se încercă realizarea unui proiect mai flexibil sau reutilizabil.

Obiectele pot varia drastic ca număr şi dimensiune.

Se poate reprezenta orice ca obiect, plecând de la hardware şi până la întrega aplicaţie.

Cum decidem care poate fi un obiect?

Şabloanele se adresează exact acestei cerinţe.

Şablonul *Facade* descrie cum putem reprezenta subsisteme complete ca obiecte, *Flyweight* descrie cum vor fi suportate un număr uriaş de obiecte de o granularitate mai fină.

Alte şabloane descriu căi specifice de descompunere a unui obiect în altele mai mici.

6.3. Cum se selectează un şablon

Atunci când sunt mai multe şabloane la dispoziţie este relativ dificil să se selecteze exact acelea care se adresează unei probleme particulare, mai ales atunci când proiectantul nu este familiarizat cu ele.

Există câteva criterii de selecţie a celui mai potrivit şablon pentru o anumită problemă.

- Se trec în revistă şabloanele de proiectare care sunt disponibile, se studiază ce oferă fiecare şi se aleg acelea care se potrivesc cel mai bine problemei care trebuie proiectată.
- Se studiază relaţiile dintre şabloane (fig. 6.1) şi se alege grupul cel mai potrivit.
- Se studiază scopurile, asemanările şi deosebirile dintre şabloane.

- Se parcurge lista (Tabel 6.2.) cu acele aspecte ale şabloanelor care pot fi modificate în mod independent, modificări care corespund scopului proiectului propus.

Dacă proiectantul nu are experienţă în proiectarea orientată pe obiect se poate începe cu cele mai simple şi comune şabloane de proiectare: Abstract Factory, Factory Method, Adapter, Observer, Composite, Strategy, Decorator, Template Method.

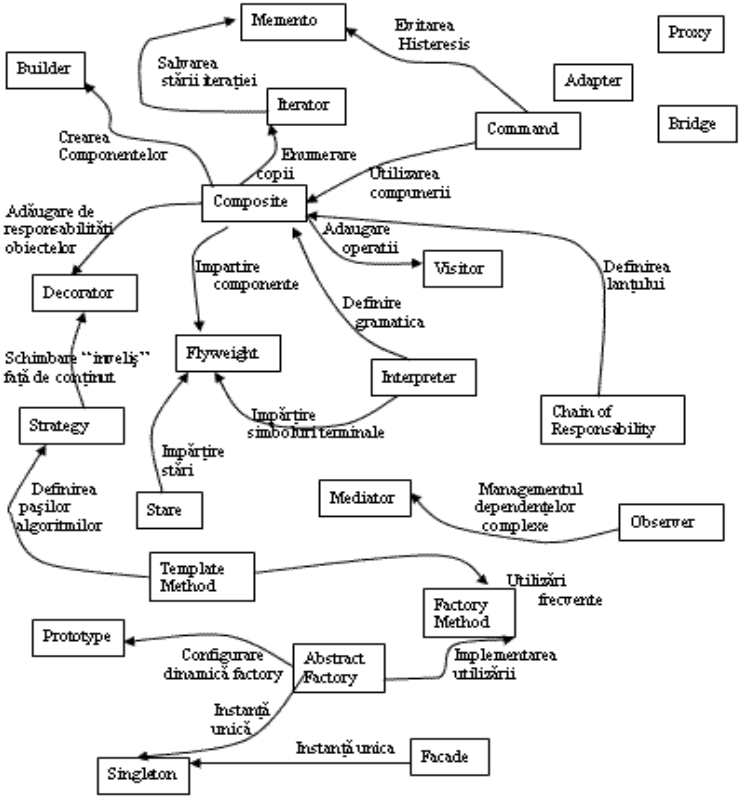


Fig. 6.1. Relaţiile între şabloanele de proiectare

6.4. Cum se foloseşte un şablon de proiectare

O abordare pas cu pas a proiectării unui sistem software folosind şabloane ar putea să decurgă în felul următor:

1. *Se citeşte descrierea şablonului pentru a avea o imagine generală*. Se acordă o atenţie deosebită secţiunilor de aplicabilitate şi consecinţe pentru a avea garanţia că şablonul de proiectare corespunde problemei;
2. *Se revine şi se studiază secţiunile de structură, participanţi şi colaborări*. Trebuie să se înţeleagă exact clasele şi obiectele din şablon şi modul în care aceastea relaţionează unele cu altele.
3. *Se examinează apoi secţiunea de cod care prezintă un exemplu concret*. Studiind codul se poate învăţa cum se implementează şablonul.
4. *Se aleg numele participanţiilor din şablon*, nume care trebuie să aibă o semnificaţie în contextul aplicaţiei. Numele participanţilor din design patterns sunt de obicei prea abstarcte pentru a apărea direct în aplicaţie şi de aceea este util să se încorporeze numele participanţilor în numele care apar în aplicaţie pentru a fi mai explicite în implementare. De

exemplu, dacă se utilizează *Strategy* pattern pentru un text care compune algoritmul, atunci putem avea clasa numită *SimpleLayoutStrategy* sau *TextLayoutStrategy.*

5. *Se definesc clasele*. Se declară interfeţele, se stabilesc relaţiile de moştenire şi se definesc variabilele care reprezintă datele şi obiectele referite. Se identifică clasele existente din aplicaţie pe care şablonul le va afecta şi se pun de acord.
6. *Se definesc numele specifice aplicaţiei pentru operaţiile din şablon*. Încă o dată, numele depind în general de aplicaţie. Se folosesc responsabilităţile şi colaborările asociate cu fiecare operaţie ca un ghid. De asemenea, este bine să existe consecvenţă în convenţiile de notare. De exemplu, se foloseşte prefixul “Creează-„ de fiecare dată când desemnăm o metodă Factory.
7. *Se implementează operaţiile având grijă de responsabilităţile şi colaborările din şablon*. Secţiunea de implemantare oferă sugestii pentru implementare.
8. *Se definesc clasele*. Se declară interfeţele, se stabilesc relaţiile de moştenire şi se definesc variabilele care reprezintă datele şi obiectele referite. Se identifică clasele existente din aplicaţie pe care şablonul le va afecta şi se pun de acord.

9. *Se definesc clasele*. Se declară interfeţele, se stabilesc relaţile de moştenire şi se definesc variabilele care reprezintă datele şi obiectele referite. Se identifică clasele existente din aplicaţie pe care şablonul le va afecta şi se pun de acord.
10. *Se definesc numele specifice aplicaţiei pentru operaţiile din şablon*. Încă o dată, numele depind în general de aplicaţie. Se folosesc responsabilităţile şi colaborările asociate cu fiecare operaţie ca un ghid. De asemenea, este bine să existe consecvenţă în convenţiile de notare. De exemplu, se foloseşte prefixul “Creează-„ de fiecare dată când desemnăm o metodă Factory.
11. *Se implementează operaţiile având grijă de responsabilităţile şi colaborările din şablon*. Secţiunea de implemantare oferă sugestii pentru implementare.

Tabelul 6.2. Aspectele de proiectare care pot fi modificate în design patterns

Scop	Şablon de proiectare	Aspecte care pot fi modificate
Creaţional	Abstract Factory	Familiile de obiecte produs
	Builder	Modalitatea în care pot fi create obiectele compozite
	Factory Method	Subclasa obiectelor care sunt instanţiate
	Prototype	Clasa obiectelor care este instanţiată
	Singleton	O singură instanţă a unei clase
Structural	Adapter	Interfaţa cu un obiect
	Bridge	Implementarea unui obiect
	Composite	Structura şi compoziţia unui obiect
	Decorator	Responsabilităţile proprii ale unui obiect fără subclase
	Facade	Interfaţa cu un subsistem
	Flyweight	Memorarea costurilor obiectelor
	Proxy	Modul în care este accesat obiectul- locaţia sa
Comportamental	Chain of Responsability	Obiectul care poate îndeplini o solicitare
	Command	Când şi cum poate fi indeplinită o solicitare
	Interpreter	Gramatica şi interpretarea unui limbaj
	Iterator	Cum sunt accesate şi parcurse elementele unei mulţimi
	Mediator	Cum şi care obiecte interacţionează unele cu altele
	Memento	Ce informaţie privată este memorată în afara obiectului şi când.
	Observer	Numărul obiectelor care depind de alt obiect; Cum poate fi

		actualizat obiectul dependent.
	State	Stările unui obiect
	Strategy	Un algoritm
	Template Method	Paşii unui algoritm
	Visitor	Operaţiile care pot fi aplicate obiectului (obiectelor) fără schimbarea clasei (claselor).

Proiectarea sistemelor software

7.1. Procesul de proiectare

Proiectarea software-ului implică următoarele stadii:

(1*) Studierea şi înţelegerea problemei*.

Problema trebuie examinată din unghiuri diferite, astfel încât să permită o privire din interior a cerinţelor de proiectare.

(2) *Identificarea caracteristicilor principale şi, în cele din urmă, o posibilă soluţie.*

- Adesea este util să se identifice mai multe soluţii şi să se evalueze fiecare dintre ele.
- Alegerea soluţiei depinde de experienţa proiectantului, de componentele reutilizabile pe care le are la dispoziţie, de simplitatea soluţiilor derivate.

(3) *Descrierea fiecărei abstractizări utilizate în soluţie.*

- Înainte de a crea documentaţia formală, totuşi proiectantul trebuie să stabilescă dacă este necesar să construiască o descriere informală a proiectului.
- Erorile şi omisiunile din nivelurile înalte ale proiectării, care sunt descoperite de-a lungul proiectării la nivelurile scăzute, pot fi corectate înainte ca proiectul să fie documentat.

Un model general de proiectare a software-ului este un graf orientat.

- Nodurile grafului reprezintă entităţile de proiectare, cum ar fi procesele, funcţiile sau tipurile, iar legăturile reprezintă relaţiile existente între entităţi.
- Scopul procesului de proiectare este de a crea un asemenea graf fără inconsistenţe şi în care toate relaţiile dintre entităţi sunt legale (posibile).

Proiectantul începe cu o imagine foarte neformală a proiectului şi o rafinează, adăugându-i informaţie pentru a realiza un proiect mai bine formalizat (Fig.7.1).

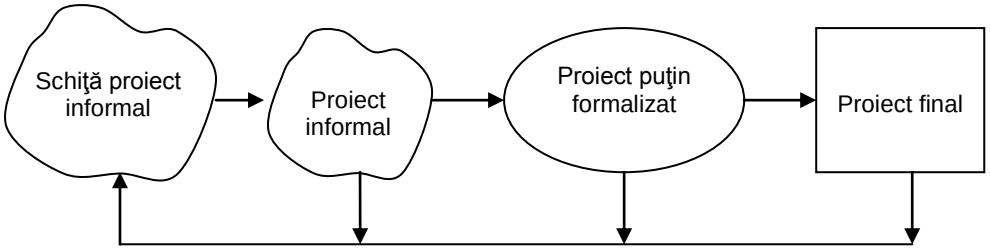


Fig.7.1. Procesul de proiectare

Procesul de proiectare implică descrierea sistemului într-un număr diferit de niveluri de abstractizare, permiţând astfel descoperirea mai devreme a omisiunilor sau a erorilor.

Această reacţie creează posibilitatea ca diferitele stadii de proiectare să fie rafinate.

Fig.7.2. prezintă stadiile procesului de proiectare şi al descrierilor de proiectare produse ca rezultat al acestor activităţi.

Ieşirea fiecărei activităţi de proiectare este o specificaţie.

Astfel, procesului de proiectare continuă, adăugându-se din ce în ce mai multe detalii la specificare.

Ultimile ieşiri sunt specificaţiile algoritmilor şi ale structurilor de date care vor constitui baza pentru implementarea sistemului.

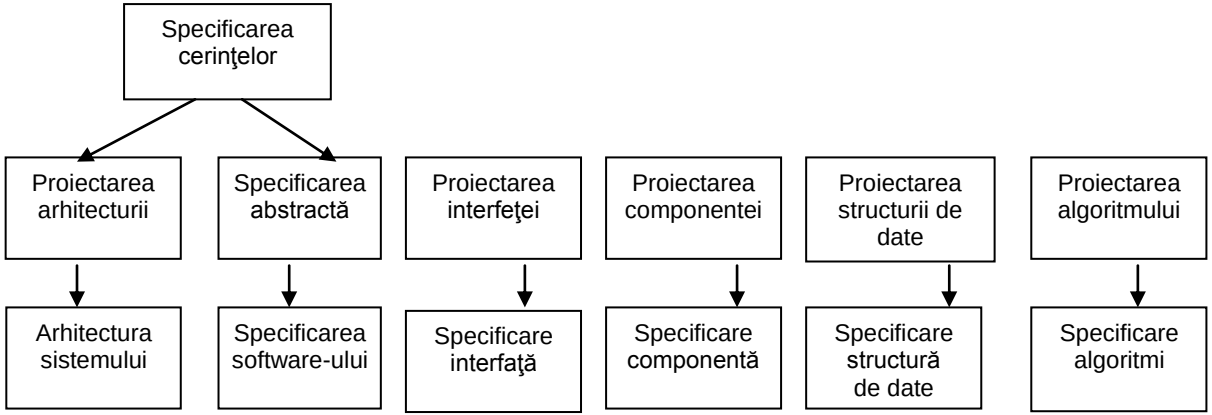


Fig.7.2. Activităţi de proiectare şi produse proiectate

Activităţile prezentate în fig. 7.2 sunt esenţiale în proiectarea sistemelor software mari şi cuprind următoarele:

(1). Proiectarea arhitecturii. Se realizează subsistemele întregului sistem identificându-se şi documentându-se relaţiile dintre ele.

(2). Specificarea abstractă. Se generează specificări abstracte ale serviciilor pentru fiecare subsistem în parte şi se stabilesc restricţiile sub care va opera sistemul.

(4). Proiectarea componentei. Serviciile produse de subsistem sunt partiţionate în funcţie de componentele din acel subsistem.

(5). Proiectarea structurilor de date. Structurile de date folosite în implementarea sistemului vor fi proiectate în detaliu şi specificate.

(6). Proiectarea algoritmului. Algoritmii utilizaţi pentru a furniza servicii vor fi proiectaţi în detaliu şi specificaţi.

Procesul este repetat pentru fiecare subsistem, până când componentele identificate pot fi transpuse direct în componente ale unui limbaj de programare cum ar fi package, proceduri sau funcţii.

O abordare recomandată pentru proiectarea pe scară largă este proiectarea top-down în care problema este recursiv împărţită în subprobleme, până când sunt identificate toate subproblemele elementare.

Forma generală a proiectului care de obicei iese la iveală dintr-o astfel de abordare este aproximativ ierarhică (fig.4.3), legăturile de încrucişare observându-se în graf pe nivelurile scăzute ale arborelui de proiectare, astfel încât proiectanţii pot identifica posibilităţile de reutilizare.

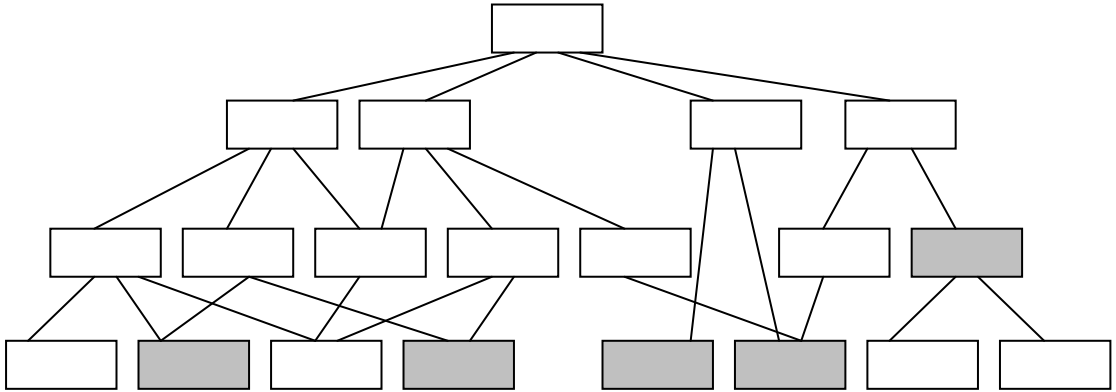


Fig.7. 3. Structura proiectării software

De fapt nu este recomandabil să se proiecteze sistemele într-o manieră care este strict top-down.

Proiectanţii îşi utilizează adesea experienţa anterioară de proiectare şi nu au nevoie întotdeauna să descompună toate abstractizările, ştiind exact care parte a sistemului poate fi construită.

Proiectarea top-down a fost propusă în conjuncţie cu descompunerea funcţională şi este o abordare validă în care componentele proiectate sunt strâns legate.

Totuşi când este adoptată o proiectare orientată pe obiect şi multe obiecte existente urmează să fie reutilizate, proiectarea top-down nu este cea mai indicată.

Proiectantul utilizează obiectele existente într-o reţea de proiectare şi construieşte proiectul cu ele.

Nu există nici un concept de ierarhizare a tuturor obiectelor existente într-un singur obiect ierarhic.

7.2. Proiectarea arhitecturală

Este etapa în care se construieşte **soluţia problemei.** Rezultatul este **un model fizic** al viitorului sistem, care este descris în **Documentul de Proiectare Arhitecturală.**

Cerinţele din documentul Cerinţelor Software sunt alocate unor componente ale viitorului sistem. Rezulta un set de subsisteme/componente interconectate. Arhitectura software rezulta printr-un proces iterativ de descompunere a cerinţelor. Documentul de proiectare arhitecturala trebuie sa specifice rolul fiecarei componente, cerinţele care i-au fost alocate, interfaţa de comunicare cu celelalte componente ale sistemului. De asemenea, in etapa de proiectare arhitecturala se întocmeşte **Planul Testelor de Integrare.**

7.2.1. Procesul: obţinerea modelului de proiectare arhitecturala

1. Abordare descendenta

- Se pleaca de la specificatia cerintelor software: S
- Se descompune setul cerintelor, S, in subseturi relativ independente, S1, S2,

- Fiecare subset de cerinte, Si, este alocat unui subsistem al arhitecturii
- software: SA1, SA2...
- Fiecare subset de cerinte Si este descompus in subseturi mai simple, Si1, Si2, .. care sunt alocate unor subsisteme ale subsistemului SAi: SAi1, SAi2,..
- .....

➢ **Descompunerea modelului de proiectare arhitecturala se opreşte atunci cand nivelul sau de detaliu permite**:

- continuarea dezvoltarii in paralel de catre mai multi membrii ai echipei de dezvoltare,
- planificarea activitatilor urmatoare ale procesului de dezvoltare (pana la livrare),
- estimarea resurselor umane necesare si a costurilor.

Rezultatul este o structura ierarhica alcatuita din subsisteme interconectate, fiecare subsistem fiind descompus pana la nivel de module.

➢ **Un modul de proiectare poate fi**: modul functional (functie, procedura), clasa, componenta, in functie de metoda de proiectare folosita: functionala sau orientat obiect.

- Fiecarui modul de proiectare i s-a alocat un set de cerinte pe care trebuie sa le implementeze.

- Se incearca gasirea unor module existente care ar putea fi folosite in implementarea modulelor definite in procesul de proiectare.

2. Abordare ascendentă

- Centrata pe reutilizare
- Se pleaca de la un set de module existente : module functionale sau clase
- Se cauta o descompunere a cerintelor in subseturi care pot fi implementate folosind componentele existente

3. Abordare hibrida

Se pleaca de la setul de cerinte software, care se descompune iterativ, dar in procesul de descompunere se urmareste definirea de module care pot fi implementate folosind module existente.

7.2.2. Modelul de proiectare athitecturală

- Este o structură ierarhică realizată din subsisteme interconectate, fiecare subsistem fiind alcatuit dintr-un set de module interconectate sau din alte subsisteme, s.a.m.d.
- Modelul poate fi reprezentat prin :

- diagrame de componente si diagrame de distributie combinate cu diagrame de componente
- diagrame de clase, in care relaţiile ierarhice se bazează pe generalizare si specializare
- diagrame de structura (Fig.7.4), in cazul unei descompuneri funcţionale : descompunerea iterativa a funcţiilor pe care trebuie sa le implementeze sistemul

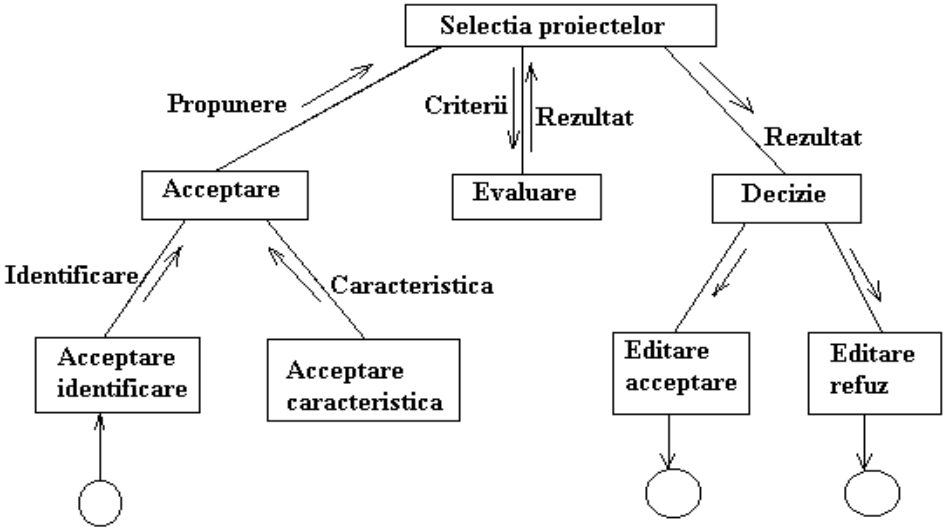


Fig.7.4. Diagrama de structura

- Nodurile arborelui sunt module functionale.
- Arcele reprezinta relatiile de apel intre module.
- Sagetile indica fluxul datelor

Pentru fiecare modul al arhitecturii software se scrie o specificatie:

Fiecare modul este specificat prin:

- Identificatorul(unic) si tipul sau (clasa, functie, fisier, program)
- Scopul sau – cerintele software pe care le implementeaza
- Componenetele subordonate: modulele apelate/ obiectele utilizate/ fisierele unei baze de date
- Interfata componentei: fluxul datelor si al controlului
- Dependentele sale: conditii care trebuie sa fie satisfacute inainte sau dupa executia sa, operatii interzise in timpul executiei componentei
- Prelucrarea interna a componentei, la nivelul cel mai coborat in limbaj natural
- Structurile de date interne

Planul testelor de integrare precizeaza ordinea in care vor fi integrate modulele in subsisteme si apoi subsistemele pana la nivel de sistem precum si testele care vor fi efectuate la integrare.

7.3. Proiectarea calităţii

Nu există nici în momentul de faţă o modalitate absolut garantată care să ateste că un proiect este fără nici o eroare. În funcţie de domeniul de aplicaţie şi de cerinţele proiectului, un proiect bun ar putea fi proiectul care permite producerea unui cod eficient, sau ar putea fi proiectul cu dimensiunea cea mai mică posibilă, pentru care implementarea să fie cât mai compactă, sau proiectul care asigură mentenanţa cea mai bună.

Un proiect mentenabil poate fi cu rapiditate adaptat pentru a i se adauga noi functionalităţi sau pentru a le modifica pe cele existente. Proiectul trebuie să fie inteligibil, iar schimbările să aibă un efect local. Componentele proiectului trebuie să aibă coeziune, ceea ce înseamnă că toate părţile componente să fie legate logic între ele. Ele trebuie să poată fi cu uşurinţă decuplate, cuplajul fiind o măsură a independenţei componentelor.

Multe preocupări şi cercetări ştiinţifice au avut în vedere stabilirea unor metrici de proiectare a calităţii, care să ateste în final că un proiect este bun. Cele mai multe asemenea metrici au fost dezvoltate în conjuncţie cu metodele structurate ale lui Yourdan. Caracteristicile de calitate sunt în mod egal aplicabile atât proiectării orientate pe obiect cât şi proiectării oriantate funcţional, şi sunt : coeziunea, cuplajul, inteligibilitatea, adaptabilitatea.

7.3.1. Coeziunea

Coeziunea unei componente este o măsură a cât de bine se potriveşte ea logic cu celelalte. O componentă poate implementa o singură funcţie logică sau o singură entitate logică şi toate părţile componentei trebuie să contribue la această implementare. Dacă componenta include părţi care nu sunt direct legate de funcţia ei logică (de exemplu, dacă există un grup de operaţii care se execută în acelaşi timp şi nu au legătură cu funcţia), aceasta înseamnă că gradul de coeziune este scazut.

Constantine şi Yourdan (1979) au identificat şapte niveluri de coeziune în ordine crescătoare a gradului, şi anume:

1). *Coeziune de coincidenţă*. Părţile componente nu sunt într-o relaţie, ci pur şi simplu sunt asamblate într-o singură componentă.

2). *Asociere logică.* Componentele care realizează funcţii similare, cum ar fi operaţii de intrare, tratarea erorilor,etc., sunt asamblate împreună într-o singură componentă.

3). *Coeziune temporală*. Toate componentele care sunt active la un moment de timp, cum ar fi cele necesare la pornire sau la oprire, trebuie să fie la un loc.

4). *Coeziunea procedurală*. Elementele dintr-o componentă trebuie să aibă o singura secvenţă de control.

5). *Coeziune de comunicare.* Toate elementele unei componente operează pe aceleaşi date de intrare sau produc aceleaşi date de ieşire.

6). *Coeziune secvenţială*. Ieşirea dintr-un element al componentei serveşte ca intrare pentru alt element.

7). *Coeziune funcţională.* Fiecare parte a componentei este necesară pentru execuţia unei singure funcţii.

Aceste clase de coeziune nu sunt în mod strict definite, iar Constantine şi Yourdan le-au ilustrat prin exemple. Nu este întotdeauna uşor să se facă o clasificare a unui sistem într-o anumită categorie de coeziune.

Metoda lui Yourdan este aplicabilă şi este evident că cea mai bună formă de coeziune a unei unităţi este funcţia.Totuşi, cel mai mare grad de coeziune îl au sistemele orientate pe obiecte şi acest lucru reprezintă de fapt o caracteristică a acestora. Într-adevar, unul dintre principalele avantaje ale acestei modalităţi de proiectare este acela că obiectele creează sistemului o coeziune naturală.

Un obiect coerent (care are coeziune) este acela în care este reprezentată o singură entitate şi în care sunt incluse toate operaţiile pentru acea entitate. În acest context se poate defini următoarea clasa de coeziune:

-*Coeziune de obiect*. Fiecare operaţie are ca rezultat acea funcţionalitate care să permită obiectului să poată fi modificat, inspectat sau utilizat ca sursă de servicii.

Coeziunea este o caracteristică de dorit deoarece aceasta înseamnă că un modul reprezintă o singură parte din soluţia problemei. Dacă este necesar să se modifice sistemul, acestă parte există într-un singur loc şi orice trebuie făcut cu ea se află încapsulat într-o singură unitate.

Dacă funcţionalitatea este furnizată de un sistem-obiect care utilizează moştenirea din super-clase, coeziunea obiectului care moşteneşte atribute şi operaţii este redusă. Nu este posibil mult timp să se considere obiectul ca o unitate distinctă. Toate super-clasele vor putea de asemenea să fie inspectate dacă funcţionalitatea unui obiect nu este în totalitate înţeleasă.

7.3.2. Cuplajul

Cuplajul este legat de coeziune şi este un indicator al tăriei intercorelării dintre unităţile programului.

Sistemele cu cuplaj ridicat au interconectări puternice cu unităţile din program care depind unele de altele. Sistemele cu cuplaj slab sunt construite din unităţi independente sau aproape independente.

Ca reguli generale: toate modulele au cuplaj ridicat dacă ele utilizează variabile în comun sau dacă ele îşi schimbă informaţia de control. Constantine si Yourdan au numit această *cuplare comună* sau *cuplare prin control*. Cuplajul slab este obţinut prin asigurarea ca, atât timp cât este posibil, informaţia reprezentativă este păstrată în interiorul unei componente, iar interfaţa de date cu alte unităţi se realizează numai prin lista de parametri. Dacă este nevoie să fie partajate datele, această partajare poate fi controlată, cum este în ADA în cazul “package”. Aceasta se numeşte cuplare prin date. **Fig. 7.5 si 7.6** ilustrează cuplajul puternic, respectiv cuplajul slab al modulelor.

O problema a cuplajului provine din faptul că se pot cupla diverse module prin valori de variabile. Orice schimbare a acestor valori presupune o schimbare în program.

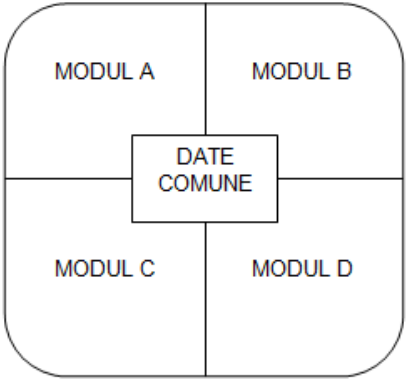


Fig.7.5. Cuplaj puternic

Se pare ca principalul avantaj al proiectării orientate pe obiect rezidă din faptul că proiectele rezultate manifestă un cuplaj slab.

Principala caracteristică a acestei abordari orientate pe obiect este tocmai aceea că “ascunde” ceea ce este în interiorul obiectului astfel încât sistemul nu trebuie să aibă o stare partiţionată cu nici un alt obiect, iar un obiect poate fi la rândul lui înlocuit de altul, cu aceeaşi interfaţă.

Moştenirea în sistemele orientate pe obiect are o formă definită de cuplaj.

Obiectele care moştenesc atribute şi operaţii sunt cuplate cu super-clasele lor, iar schimbările făcute în superclase trebuie operate cu grijă, deoarece acestea se propagă în toate clasele care moştenesc acea super-clasă.

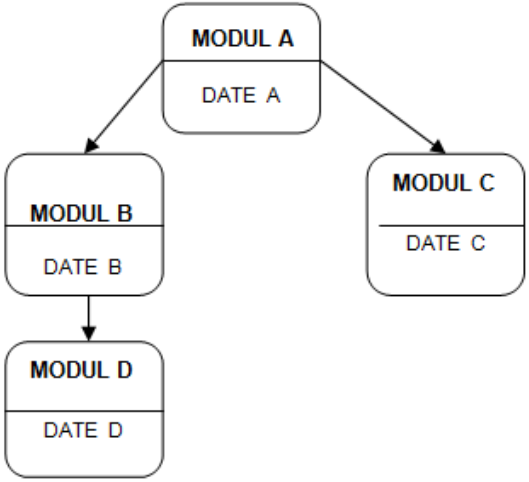


Fig.7.6 Cuplaj slab

7.3.3. Inteligibilitatea

Schimbarea componentei unui proiect are drept implicaţie ca persoana responsabilă de acea schimbare să înţeleagă operaţia facută. Acest lucru se referă la câteva caracteristici ale componentei şi anume:

(1) Coeziunea. Poate acea componenta să fie înţeleasă fără alte referiri la celelalte componente?

(2) Numele. Sunt numele utilizate în înţelesul componentei?

(3) Documentare. Este componenta documentată astfel încât să reflecte o legatură clară între entităţile din lumea reala şi componentă?

(4) Complexitate. Cât de mare este complexitatea algoritmilor utilizaţi în implementarea componentei? În acest context se utilizează complexitatea într-o manieră neformală.

Complexitatea înaltă implică multe relaţii între diferite componente ale proiectului şi o structură logică complexă, care poate implica secvenţe “if-then-else” imbricate.

Complexitatea componentelor este greu de înţeles şi de aceea proiectantul trebuie să se străduiască să conceapă componente pe cât posibil mai simple.

Cel mai mare efort în stabilirea de metrici pentru calitatea proiectului este concentrat pe încercarea de a asigura complexitatea componentei, obţinându-se astfel o măsură a inteligibilităţii componentei. Complexitatea afectează înţelegerea, dar sunt şi alţi factori care o afectează, cum ar fi organizarea datelor şi stilul în care proiectul este descris.

Măsurile complexităţii pot numai să furnizeze un indicator al inteligibilităţii componentei. Moştenirea în proiectele orientate pe obiect afectează înţelegerea.

Moştenirea este folosită pentru a ascunde detalii de proiectare, iar proiectul este uşor de înţeles.

Pe de altă parte, utilizarea moştenirii solicită celui care citeşte proiectul să privească la multe clase diferite de obiecte din ierarhia de moştenire, iar înţelegerea proiectului este redusă.

7.3.4. Adaptabilitatea

Pentru ca un proiect să fie bine întreţinut, el trebuie să poată fi cu uşurinţă adaptat diverselor cerinţe survenite ulterior. Aceasta implică, subînţeles, ca toate componentele să fie cuplate slab. Mai mult decât atât, adaptabilitatea înseamnă ca proiectul să fie bine documentat, documentaţia

componentelor să poată fi înţeleasă cu uşurinţă în concordanţă cu implementarea, iar implementarea să fie scrisă într-o formă accesibilă.

Un proiect adaptabil trebuie să aibă un nivel înalt de vizibilitate şi trebuie să existe o relaţie clară între diferitele niveluri ale proiectului. Trebuie să fie posibil pentru cititorul proiectului să găsească relaţia dintre reprezentarea ca o diagramă de structuri şi reprezentarea transformării datelor (Fig.7.7).

De asemenea, trebuie să fie uşor să se încorporeze schimbările făcute din proiect în toate documentele proiectului. Pentru o adaptabilitate optimă componenta trebuie să se conţină numai pe sine.

Adaptabilitatea unei componente poate să implice schimbări numai ale unei părţi a componentei care se referă la funcţiile externe, astfel ca specificarea acestor funcţii externe să fie de asemenea luată în consideraţie de către cel care face modificarea.

Unul dintre principalele avantaje ale moştenirii sistemelor orientate pe obiect este acela că în acest caz componentele pot fi adaptate cu uşurinţă.

Mecanismul de adaptare nu se referă la modificarea componentei, ci la crearea unei noi componente care moşteneşte atributele şi operaţiile componentei originale.

Numai atributele şi operaţiile care trebuie schimbate sunt modificate. Această adaptabilitate simplă este motivul pentru care limbajele orientate pe obiect sunt atât de eficiente pentru prototipizarea rapidă.

Totuşi, pentru sistemele cu viaţă scurtă, trebuie avut în vedere că moştenirea devine o problemă atunci când sunt operate din ce în ce mai multe schimbări şi reţeaua de moştenire devine din ce în ce mai complexă.

Funcţionalitatea este adesea distribuită în diverse puncte ale reţelei, iar componentele sunt în acest caz greu de înţeles.

Experienţa în programarea orientată pe obiect a arătat că reţeaua de moştenire trebuie periodic revizuită şi restructurată pentru a se reduce complexitatea şi duplicarea funcţionalităţii. În mod clar, aceasta face să crească costul schimbării sistemului.

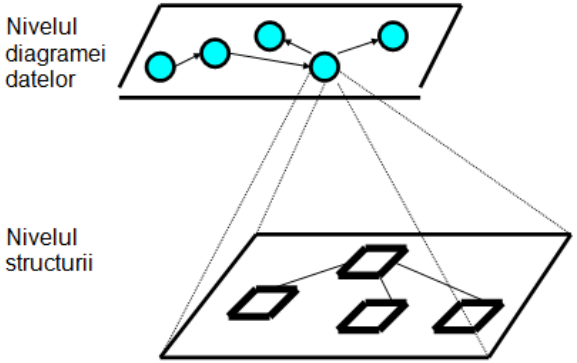


Fig.7.7 Diagrama de structură şi reprezentare a datelor

7.4. Modularizarea proiectului

7.4.1. Principii de modularizare:

- Modulele trebuie sa fie simple si cat mai independente unul de altul:
    - O modificare a unui modul are influenta minima asupra altor componente
    - O schimbare mica a cerintelor nu conduce la modificari majore ale arhitecturii software (gruparea cerintelor corelate in acelasi modul)
    - Efectul unei conditii de eroare este izolat in modulul are a generat-o
    - Un modul poate fi inteles ca o entitate de sine-statatoare
- Modulele trebuie sa “ascunda” modul de implementare a functiilor descrise de interfata lor, de ex. cum sunt memorate datele cu care lucreaza (tablou, lista, arbore, in memorie sau intr-un fisier)

7.4.2. Reguli de proiectare a modulelor:

- **Minimizarea cuplarii** intre module:
    - Minimizarea numarului de elemente prin care comunica modulele;
    - Evitarea cuplarii prin structuri de date
    - Evitarea cuplarii prin variabile “steag” (cuplarea prin control)
    - Evitarea cuplarii prin date globale
- **Maximizarea coeziunii interne** a fiecarei componente: elementele grupate intr-o componenta trebuie sa fie corelate, de ex. sa contribuie la aceeasi prelucrare
- **Restrangerea numarului de module apelate (fan-out)** de un modul:
- **Maximizarea numarului de module care utilizeaza un modul (fan-in)** – incurajaza reutilizarea (Fig. 7.8)

  “Fan-in” mare: un numar mare de module depind de el

„Fan-out“ mare: modulul depinde de multe module

- **Factorizare**: functionalitatile comune sunt definite in module reutlizabile.

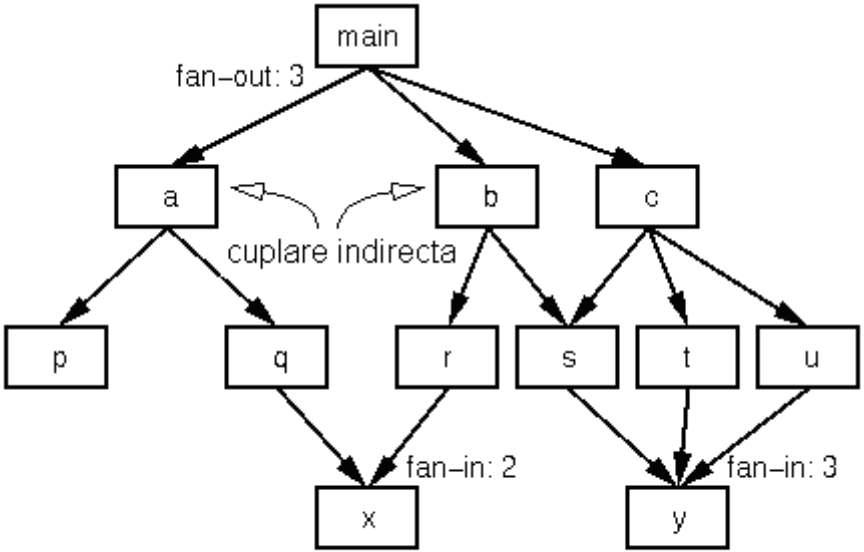


Fig.7.8. Exemplu de diagramă de structură

Documentul de proiectare arhitecturala (ADD)

Şablonul documentului in standardele ESA:

a.	Abstract	
b.	Table of Contents	
c.	Document Status Sheet	Status sheet for configuration control.
d.	Document Change Records since previous issue	A list of document changes.
1. Introduction		
1.1	Purpose	The purpose of this particular ADD and its intended readership.
1.2	Scope	Scope of the software. Identifies the product by name, explains what the software will do.
1.3	List of definitions	The definitions of all used terms, acronyms and abbreviations.
1.4	List of references	All applicable documents.
1.5	Overview	Short description of the rest of the ADD and how it is organized.
2. System overview		Short introduction to system context and design. Background of the project.
3. System context (for each external interface ...)		
3.n	External interface definition	The relationship with external system n.
4. System design		

4.1	Design method	Name and reference of the method used.
4.2	Decomposition description	Overview of components: decomposition, dependency or interface view.
5. Component descriptions (for each component ...)		
5.n	Component identifier	A unique identifier.
5.n.1	Type	Task, procedure, package, program, file, ...
5.n.2	Purpose	Software requirements implemented.
5.n.3	Function	What the component does.
5.n.4	Subordinates	Child components (modules called, files composed of, classes used).
5.n.5	Dependencies	Components to be executed before/after, excluded operations during execution.
5.n.6	Interfaces	Data and control flow in and out.
5.n.7	Resources	Needed to perform the function.
5.n.8	References	To other documents.
5.n.9	Processing	Internal control and data flow.
5.n.10	Data	Internal data.
6. Feasibility and resource estimates		A summary of computer resources needed to build, operate and maintain the software.
7. Requirements traceability matrix		A table showing how each software requirement of the SRD is linked to components in the ADD.

7.4.3. Proiectarea de detaliu

- Se efectuează la nivelul modulelor definite in proiectarea arhitecturala.
- Poate avea loc in paralel, pentru diferite module.
- Detaliază modelul de proiectare arhitecturala:
  - pot fi definite module de nivel mai coborât
  - se detaliază componenta claselor: atributele si funcţiile membre
  - se aleg biblioteci utilizate in implementare
  - se încurajează reutilizarea
  - sunt descrişi algoritmii

Testarea sistemelor software

8.1. Intoducere

Un test constă în execuţia programului pentru un set de date de intrare convenabil ales, pentru a verifica dacă rezultatul obţinut este corect.

Un caz de test este un set de date de intrare împreună cu datele de ieşire pe care programul ar trebui să le producă.

Cazurile de test se aleg astfel încât să fie puse în evidenţă, dacă este posibil, situaţiile de funcţionare necorespunzătoare.

Testarea este activitatea de concepţie a cazurilor de test, de execuţie a testelor şi de evaluare a rezultatelor testelor, în diferite etape ale ciclului de viaţă al programelor.

Tehnicile de testare sunt mult mai costisitoare decât metodele statice (inspectările şi demonstrările de corectitudine), de aceea în prezent se apreciază că tehnicile statice pot fi folosite pentru a le completa pe cele dinamice, obţinîndu-se astfel o reducere a costului total al procesului de verificare şi validare.

Metodele traditionale de verificare/validare presupun realizarea verificării/validării prin inspectări şi teste.

Aceste activităţi ocupă circa 30÷50% din efortul total de dezvoltare, în funcţie de natura aplicaţiei.

Prin testare nu se poate demonstra corectitudinea unui program.

Aceasta deoarece în cele mai multe cazuri este practic imposibil să se testeze programul pentru toate seturile de date de intrare care pot conduce la execuţii diferite ale programului.

Testarea poate doar să demonstreze prezenţa erorilor într-un program.

Într-un sens, testarea este un proces distructiv, deoarece urmăreşte să determine o comportare a programului neintenţionată de proiectanţi sau implementatori.

Din acest punct de vedere testarea nu trebuie să fie făcută de persoanele care au contribuit la dezvoltarea programului.

Pe de altă parte conoaşterea structurii programului si a codului poate fi foarte utilă pentru alegerea unor date de test relevante.

Testarea în vederea verificării programului foloseşte date de test alese de participanţii la procesul de dezvoltare a programului.

Ea este efectuată la mai multe nivele: la nivel de unitate funcţională, de modul, de subsistem, de sistem.

Testarea în vederea validării programului, numită şi ***testare de acceptare,*** are drept scop stabilirea faptului că programul satisface cerinţele viitorilor utilizatori.

Ea se efectuează în mediul în care urmează să funcţioneze programul, deci folosindu-se *date reale*.

Prin testarea de acceptare pot fi descoperite şi erori, deci se efectuează şi o verificare a programului.

8.2. Testarea pe parcursul ciclului de viaţă al unui program.

8.2.1. Testele "unitare"

Testarea unui modul (o funcţie, o clasă, unitate) este realizată de programatorul care implementează modulul.

Toate celelalte teste sunt efectuate, în general, de persoane care nu au participat la dezvoltarea programului.

Scopul testarii unui modul este de a se stabili că modulul este o implementare corectă a specificaţiei sale (conforma cu specificatia sa). Specificatia poate fi neformala sau formala.

De exemplu:

- o specificaţie de pre şi post condiţii pentru o funcţie sau procedură;
- un invariant al clasei, care specifică mulţimea stărilor posibile ale fiecărui obiect din clasa respectivă, împreună cu specificaţii de pre şi post condiţii la nivelul funcţiilor membru;
- o specificaţie algebrică a clasei;
- o specificaţie Z/Z++ etc.

În cursul testarii unui modul, modulul este tratat ca o entitate independentă, care nu necesită prezenţa altor componente ale programului.

Testarea izolată a unui modul ridică două probleme:

- simularea modulelor apelate de cel testat;
- simularea modulelor apelante.

Modulele prin care se simulează modulele apelate de modulul testat se numesc module "ciot" (în engleză, "stub") . Un modul "ciot" are aceeaşi interfaţă cu modulul testat şi realizează în mod simplificat funcţia sa. De exemplu, dacă modulul testat apelează o funcţie de sortare a unui vector, cu antetul:

```
void sortare(int n, int *lista);
```

se poate folosi următoarea funcţie "ciot":

```
void sortare(int n, int *lista)
{ int i;
  printf(" \n Lista de sortat este:");
  for(i=0; i<n; i++)  printf("%d", lista[i]);
  // se citeste lista sortata, furnizata de testor
   for(i=0; i<n; i++)  scanf("%d", lista[i]);
 }
```

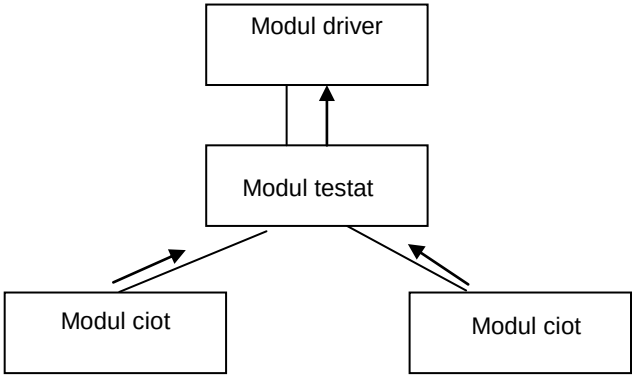


Fig.8.1. Structura programului executabil pentru testarea izolată a unui modul

Un modul "ciot" se poate reduce eventual la o tabelă de perechi de forma: "valori ale parametrilor de intrare la un apel - rezultatul prevăzut".

Cazurile de apel ale modulului testat de către celelalte module ale programului sunt simulate în cadrul unui "modul driver".

Modulul driver apelează modulul testat furnizndu-i ca date de intrare datele de test ale modulului.

Datele de test pot fi generate de modulul driver, pot fi preluate dintr-un fişier sau furnizate de testor într-o manieră interactivă.

8.2.2. Testele de integrare

Sunt dedicate verificării interacţiunilor dintre module, grupuri de module, subsisteme, până la nivel de sistem. Există mai multe metode de realizare a testelor de integrare.

Testele de integrare presupun, ca şi testele unitare, realizarea de module "ciot" şi module "driver".

Numărul de module "driver" şi de module "ciot" necesare în testele de integrare depinde de ordinea în care sunt testate modulele (metoda de integrare).

Testele de integrare necesită de asemenea instrumente de gestiune a versiunilor şi a configuraţiilor.

8.2.2.1. Metoda "big-bang"

Sunt integrate într-un program executabil toate modulele existente la un moment dat.

Modulele "driver" şi "ciot" necesare sunt de asemenea integrate.

Metoda este periculoasă căci toate erorile apar în acelaşi timp şi localizarea lor este dificilă.

8.2.2.2. Integrare progresiva

Metodele de *integrare progresivă* sunt mult mai eficiente.

În fiecare moment se adaugă ansamblului de module integrate numai un singur modul.

Astfel, erorile care apar la un test provin din modulul care a fost ultimul integrat.

8.2.2.2.1. Integrarea ascendentă (bottom-up)

Se începe prin testarea modulelor care nu apelează alte module, apoi se adaugă progresiv module care apelează numai modulele deja testate, până când este asamblat întregul sistem.

Metoda necesită implementarea câte unui modul "driver" pentru fiecare modul al programului (şi nici un modul "ciot").

Avantajele testării de jos în sus

Nu sunt necesare module "ciot". Modulele "driver" se implementează mult mai uşor decât modulele "ciot". Există chiar instrumente care produc automat module "driver".

Dezavantajele testării de jos în sus

1. Programul pe baza căruia se efectuează validarea cerinţelor este disponibil numai după testarea ultimului modul.

2. Corectarea erorilor descoperite pe parcursul integrării necesită repetarea procesului de proiectare, codificare şi testare a modulelor.

Principalele erori de proiectare sunt descoperite de abia la sfârşit, când sunt testate modulele principale ale programului. ceea ce, în general, conduce la reproiectare şi reimplementare.

8.2.2.2.2. Integrarea descendentă (top-down)

Se începe prin testarea modulului principal, apoi se testează programul obţinut prin integrarea modulului principal şi a modulelor direct apelate de el, şi aşa mai departe.

Metoda presupune implementarea unui singur modul "driver" (pentru modulul principal) şi a câte unui modul "ciot" pentru fiecare alt modul al programului.

Integrarea descendentă poate avea loc pe parcursul unei implementări descendente a programului.

Modulul principal este testat imediat ce a fost implementat, moment în care nu au fost încă implementate modulele apelate de el.

De aceea, pentru testare este necesar să se implementeze module "ciot".

În continuare, pe măsură ce se implementează modulele de pe nivelul ierarhic inferior, se trece la testarea lor folosind alte module “ciot”, ş.a.m.d.

În fiecare pas este înlocuit un singur modul "ciot" cu cel real.

Avantajele testării de sus în jos

1. Erorile de proiectare sunt descoperite timpuriu, la inceputul procesului de integrare, atunci când sunt testate modulele principale ale programului.

Aceste erori fiind corectate la început, se evită reproiectarea şi reimplementarea majorităţii componentelor de nivel mai coborât, aşa cum se întâmplă când erorile respective sunt descoperite la sfârşitul procesului de integrare.

2. Programul obtinut este mai fiabil căci principalele module sunt cel mai mult testate.

3. Prin testarea modulelor de nivel superior se poate considera că sistemul în ansamblul său există dintr-o fază timpurie a dezvoltării şi deci se poate exersa cu el în vederea validării cerinţelor; acest aspect este de asemenea, foarte important în privinţa costului dezvoltării sistemului.

Dezavantajele testării de sus în jos

1. Este necesar să se implementeze câte un modul "ciot" pentru fiecare modul al programului, cu excepţia modulului principal.

2. Este dificil de simulat prin module "ciot" componente complexe şi componente care conţin în interfaţă structuri de date.

3. În testarea componentelor de pe primele nivele, care de regulă nu afişează rezultate, este necesar să se introducă instrucţiuni de afişare, care apoi sunt extrase, ceea ce presupune o nouă testare a modulelor.

Aceste dezavantaje pot fi reduse aplicand tehnici hibride, de exemplu, folosind în locul unor module "ciot", direct modulele reale testate.

Integrarea nu trebuie sa fie strict descendenta.

De exemplu, experienta arata ca este foarte util sa se înceapă prin integrarea modulelor de interfaţă utilizator.

Aceasta permite continuarea integrării în condiţii mai bune de observare a comportării programului.

8.3. Testele de sistem

Acestea sunt teste ale sistemului de programe şi echipamente complet.

Sistemul este instalat şi apoi testat în mediul său real de funcţionare.

Sunt teste de conformitate cu specificaţia cerintelor de sistem (software) :

- *teste functionale*, prin care se verifica satisfacerea cerintelor functionale
- *teste prin care se verifica satisfacerea cerintelor ne-functionale :*
  - de performanţă,
  - de fiabilitate,
  - de securitate, etc.

Adesea, testele de sistem ocupă cel mai mult timp din întreaga perioadă de testare.

8.3.1. Testele de acceptare (validare)

Sunt teste de conformitate cu produsul solicitat, conform contractului cu clientul (->Specificatia cerintelor utilizatorilor).

Aceste teste sunt uneori conduse de client.

Pentru unele produse software, testarea de acceptare are loc în două etape:

1. **Testarea alfa**: se efectuează folosindu-se specificaţia cerinţelor utilizatorilor, până când cele două părţi cad de acord că programul este o reprezentare satisfăcătoare a cerinţelor.
2. **Testarea beta**: programul este distribuit unor utilizatori selecţionaţi, realizându-se astfel testarea lui în condiţii reale de utilizare.

8.3.2. Testele regresive

Se numesc astfel testele executate după corectarea erorilor, pentru a se verifica dacă în cursul corectării nu au fost introduse alte erori.

Aceste teste sunt efectuate de regulă în timpul mentenanţei sistemului.

Pentru uşurarea lor este necesar să se arhiveze toate testele efectuate în timpul dezvoltării programului, ceea ce permite, în plus, verificarea automată a rezultatelor testelor regresive

8.4. Determinarea cazurilor de test

Testarea unui modul, a unui subsistem sau chiar a întregului program presupune stabilirea unui set de cazuri de test.

Un *caz de test* cuprinde:

- un set de date de intrare;
- funcţia / funcţiile exersate prin datele respective;
- rezultatele (datele de ieşire) aşteptate;

În principiu (teoretic) testarea ar trebui să fie exhaustivă, adică să asigure exersarea tuturor căilor posibile din program.

Adesea o astfel de testare este imposibilă, de aceea trebuie să se opteze pentru anumite cazuri de test.

Prin acestea **trebuie să se verifice răspunsul programului atât la intrări valide cât şi la intrări nevalide.**

Sunt două metode de generare a cazurilor de test, care nu se exclud, de multe ori fiind folosite împreună.

Ambele metode pot sta la baza unor instrumente de generare automată a datelor (cazurilor) de test:

1. Cazurile de test se determină pe baza specificaţiei componentei testate, fără cunoaşterea realizării ei; acest tip de testare se numeşte **testare "cutie neagra"** ("black box").
2. Cazurile de test se determină prin analiza codului componentei testate. Acest tip de testare se mai numeşte şi testare "cutie transparentă", sau **testare structurală.**

8.4.1. Testarea "cutie neagră".

Cazurile de test trebuie să asigure următoarele verificări:

- reacţia componentei testate la intrări valide;
- reacţia componentei la intrări nevalide;
- existenţa efectelor laterale la execuţia componentei, adică a unor efecte care nu rezultă din specificaţie;
- performanţele componentei (dacă sunt specificate).

Deoarece în marea majoritate a cazurilor testarea nu poate fi efectuată pentru toate seturile de date de intrare (testare exhaustivă), în alegerea datelor de test plecând de la specificaţii se aplică unele metode fundamentate teoretic precum şi o serie de euristici.

Alegerea cazurilor de test folosind clasele de echivalenţă

Cazurile de test pot fi alese partiţionând atât datele de intrare cât şi cele de ieşire într-un număr finit de clase de echivalenţă.

Se grupează într-o aceeaşi clasă datele care, conform specificaţiei, conduc la o aceeaşi comportare a programului.

O dată stabilite clasele de echivalenţă ale datelor de intrare, se alege câte un eşantion (de date de test) din fiecare clasă.

De exemplu, dacă o dată de intrare trebuie să fie cuprinsă între 10 şi 19, atunci clasele de echivalenţă sunt:

1) valori < 10

2) valori între 10 şi 19

3) valori > 19

Se pot alege ca date de test: 9, 15, 20.

Experienţa arată că este util să se aleagă date de test care sunt la frontiera claselor de echivalenţă. Astfel, pentru exemplul de mai sus ar fi util să se testeze programul pentru valorile de intrare 10 şi 19.

Alte cazuri de test se aleg astfel încât la folosirea lor să se obţină eşantioane din clasele de echivalenţă ale datelor de ieşire (din interiorul si de la frontierele lor).

Exemplu:

Fie un program care trebuie să genereze între 3 şi 6 numere cuprinse între 1000 şi 2500. Atunci se vor alege intrări astfel încât ieşirea programului să fie:

- 3 numere egale cu 1000
- 3 numere egale cu 2500
- 6 numere egale cu 1000
- 6 numere egale cu 2500
- rezultat eronat: mai puţin de 3 numere sau mai mult de 6 numere cu valori în afara intervalului [1000..2500]
- între 3 şi 6 numere cuprinse între 1000 şi 2500

În alegerea datelor de test trebuie să se elimine redundanţele rezultate din considerarea atât a claselor de echivalenţă de intrare cât şi a celor de ieşire.

Unele programe tratează datele de intrare în mod secvenţial.

În aceste cazuri se pot detecta erori în program testându-l cu diverse combinaţii ale intrărilor în secvenţă.

Metoda de partiţionare în clase de echivalenţă nu ajută în alegerea unor astfel de combinaţii.

Numărul de combinaţii posibile este foarte mare chiar şi pentru programe mici.

De aceea nu pot fi efectuate teste pentru toate combinaţiile.

În aceste cazuri este esenţială experienţa celui care alcătuieşte setul de cazuri de test.

Testarea "cutie neagră" este favorizată de existenţa unei specificaţii formale a componentei testate.

Exemplu: Fie o funcţie de căutare a unui număr întreg într-un tablou de numere întregi, specificată astfel:

int cauta (int x[], int nrelem, int numar);

pre : nrelem > 0 **and exist** i **in** [0..nrelem-1] : x[i] = numar

post : x “[cauta(x,nrelem,numar)] = numar **and** x' = x’’

error : cauta(x,nrelem,numar) = -1 **and** x' = x’’

Intrările valide sunt cele care satisfac pre-condiţia.

Efectele laterale ale funcţiei "cauta" s-ar putea reflecta în modificarea unor variabile globale.

Pentru evidenţierea lor ar trebui ca la fiecare execuţie de test a funcţiei să se vizualizeze valorile variabilelor globale înainte şi după apelul funcţiei.

Aceste verificări pot fi limitate, înlocuindu-se cu inspectarea_codului funcţiei, din care rezultă evantualitatea modificării unor variabile globale.

Setul minim de date de test trebuie să verifice funcţia în următoarele situaţii:

1. tablou vid (nrelem=0)
2. tablou cu un singur element (nrelem=1)
   a). valoarea parametrului "numar" este în tablou
   b). valoarea parametrului "numar" nu este în tablou
3. tablou cu număr par de elemente ("nrelem" este un număr par)
   a). "numar" este primul în tablou
   b). "numar" este ultimul în tablou
   c). "numar" este într-o poziţie oarecare a tabloului
   d). "numar" nu este în tablou

4. tablou cu număr impar de elemente ("nrelem" este impar)
şi a,b,c,d ca la punctul 3.

Din specificaţie rezultă că funcţia nu trebuie să modifice tabloul; de aceea, apelul său în programul de test trebuie să fie precedat şi urmat de vizualizarea parametrului tablou.

Setul cazurilor de test ar putea fi:

1) intrări: orice tablou
nrelem=0
orice număr
ieşiri : valoarea funcţiei = -1
tabloul nemodificat

2) intrări: x[0] = 10
nrelem=1
număr = 10
ieşiri : valoarea funcţiei = 0
tabloul nemodificat: x[0]= 10

3) intrări: x[0]= 10
nrelem=1
număr = 15
ieşiri : valoarea funcţiei = -1
tabloul nemodificat: x[0] = 10

4) intrări: x[0] = 10, x[1] = 20
nrelem=2
număr = 10
ieşiri : valoarea funcţiei = 0
tabloul nemodificat: x[0] = 10, x[1] = 20

.....

În alegerea cazurilor de test s-au folosit următoarele euristici:

- Programele de căutare prezintă erori atunci când elementul căutat este primul sau ultimul in structura de date;
- De multe ori programatorii neglijează situaţiile în care colecţia prelucrată în program are un număr de elemente neobişnuit, de exemplu zero sau unu;

- Uneori, programele de căutare se comportă diferit în cazurile: număr de elemente din colecţie par, respectiv număr de elemente din colecţie impar; de aceea sunt testate ambele situaţii

Concluzii:

- Setul de cazuri de test a fost determinat numai pe baza specificaţiei componentei şi a unor euristici (este vorba de experienţa celui care efectuează testele), deci fără să se cunoască structura internă a componentei, care este tratată ca o “cutie neagră”.

Eventuala examinare a codului sursă nu urmăreşte analiza fluxului datelor şi a căilor de execuţie, ci doar identificarea variabilelor globale cu care componenta interacţionează.

- Clasele de echivalenţă se determină pe baza specificaţiei.
- Se aleg drept cazuri de test eşantioane din fiecare clasă de echivalenţă a datelor de intrare. Experienţa arată că cele mai utile date de test sunt acelea aflate la frontierele claselor de echivalenţă.
- Cazurile de test alese verifică componenta doar pentru un număr limitat de eşantioane din clasele de echivalenţă ale datelor de intrare; faptul că din testare a rezultat că ea funcţionează corect pentru un membru al unei clase nu este o garanţie că va funcţiona corect pentru orice membru al clasei.

- Se determină de asemenea clasele de echivalenţă ale datelor de ieşire şi se aleg pentru testare datele de intrare care conduc la date de ieşire aflate la frontierele acestor clase.
- Partiţionarea în clase de echivalenţă nu ajută la depistarea erorilor datorate secvenţierii datelor de intrare. In aceste cazuri este utilă experienţa programatorului.

Reprezentarea comportării programului printr-o diagramă de stări-tranziţii poate fi folositoare în determinarea secvenţelor de date de intrare de utlizat în testarea programului.

Testele “cutie neagră”, numite şi teste funcţionale sunt utile nu numai pentru testarea programului. Ele pot fi utilizate (ca teste statice) pentru verificarea unei specificaţii intermediare faţă de o specificaţie de nivel superior.

“Slăbiciunile” testelor funcţionale:

- Nu este suficient să se verifice că programul satisface corect toate funcţiile specificate; unele proprietăţi interne, nespecificate, ca de exemplu timpul de răspuns, nu sunt verificate;
- Programul este testat pentru “ceea ce trebuie să facă” conform specificaţiilor şi nu şi pentru ceea ce face în plus, de exemplu pentru ca implementarea să fie mai eficientă sau pentru a facilita reutilizarea;

- În absenţa unui standard în privinţa specificaţiilor formale, tehnicile de testare functională sunt eterogene

8.4.2. Testarea structurală

Acest tip de testare se bazează pe analiza fluxului controlului la nivelul componentei testate.

Principalul instrument folosit în analiză este *graful program* sau *graful de control.*

Acesta este un graf orientat, ale cărui noduri sunt de două tipuri:

- noduri care corespund “blocurilor de instrucţiuni indivizibile maximale” şi
- noduri care corespund instrucţiunilor de decizie.

Fiecare bloc are un singur punct de intrare şi un singur punct de ieşire.

Arcele reprezintă transferul controlului între blocurile/instrucţiunile componentei program.

Graful program are un nod unic de intrare şi un nod unic de ieşire.

Dacă există mai multe puncte de intrare sau mai multe puncte de ieşire, acestea trebuie să fie unite într-unul singur (care se adaugă suplimentar).

Nodul de intrare are gradul de intrare zero, iar cel de ieşire are gradul de ieşire zero.

Fie următorul text de program:

```
f1=fopen(….);
f2=fopen(….);
fscanf(f1, ”%f”, x);
fscanf(f1, “%f”, y);
z=0;
while(x>=y)
{ x-=y;
  z++;
}
fprintf(f2, “%d”, z);
fclose(f1); fclose(f2);
```

Textul este decupat în următoarele blocuri:

```
1. f1=fopen(….);
   f2=fopen(….);
   fscanf(f1, ”%f, x);
   fscanf(f1, “%f, y);
```

```
   z=0
2. while(x>=y)
3. x-=y;
    z++;
4. fprintf(f2, "%d", z);
    fclose(f1);
   fclose(f2);
```

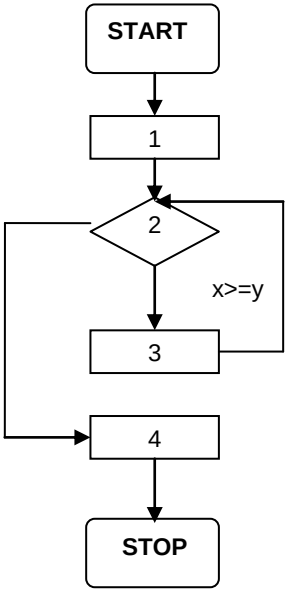


Fig.8.2. Graful de control

Graful de control este redat în Fig.8.2.

Se consideră căile care încep cu nodul de intrare şi se termină cu nodul de ieşire.

Testarea structurală constă în execuţia componentei testate pentru date de intrare alese astfel încât să fie parcurse unele dintre aceste căi.

Nu este necesar, şi în general este imposibil, să se execute toate căile de la intrare la ieşire ale unui program sau ale unei componente program.

Prezenţa ciclurilor conduce la un număr foarte mare (deseori infinit) de căi.

De asemenea, anumite căi sunt ne-executabile.

Testele structurale favorizează evidenţierea următoarelor tipuri de erori logice:

1. *Căi absente în graful program* - ca urmare a ignorării unor condiţii (de exemplu: test de împărţire la zero);
2. *Selectarea unei căi necorespunzătoare* - datorită exprimării incorecte (de multe ori incomplete) a unei condiţii;
3. *Acţiune necorespunzătoare sau absentă* (de exemplu: calculul unei valori cu o metodă incorectă, neatribuirea unei valori unei anumite variabile, apelul unei proceduri cu o listă de argumente incorectă, etc.).

Dintre acestea, cel mai simplu de depistat sunt erorile de tip 3. Pentru descoperirea lor este suficient să se asigure execuţia tuturor instrucţiunilor programului.

Alegerea căilor de testat are loc pe baza unor “criterii de acoperire” a grafului de control.

Dintre acestea cele mai cunoscute sunt:

Execuţia tuturor instrucţiunilor

Cazurile de test se aleg astfel încât să se asigure execuţia tuturor instrucţiunilor componentei testate.

Acesta este un criteriu de testare minimal, dar el nu este întotdeauna satisfăcător.

De exemplu, pentru testarea componentei cu graful din Fig.8.3, pe baza criteriului execuţiei tuturor instrucţiunilor, este suficient să se aleagă date de test care asigură execuţia conditiei si a instrucţiuilor I1 şi I2.

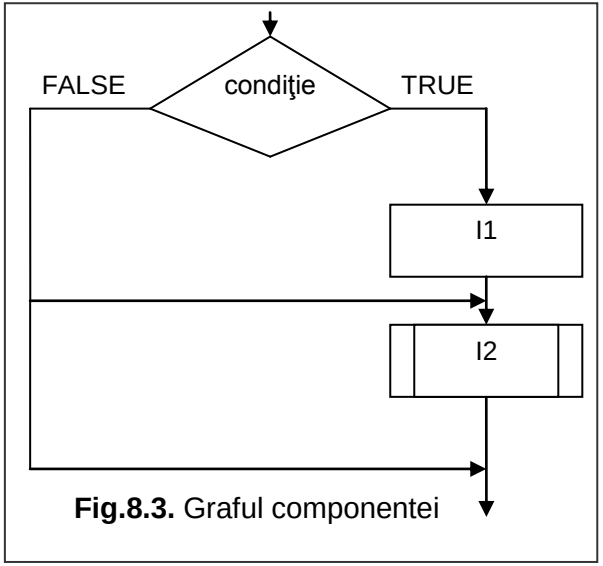


Fig.8.3. Graful componentei

Nu se testează cazurile în care condiţia are valoarea FALSE.

Transferul controlului pentru astfel de cazuri poate fi eronat.

Traversarea tuturor arcelor

Cazurile de test se aleg astfel încât să se asigure traversarea fiecărui arc al grafului program cel puţin pe o cale.

Pe baza acestui criteriu se va putea verifica dacă transferul controlului este corect pentru valoarea FALSE a condiţiei, în exemplul de mai sus.

În acelaşi timp, criteriul nu impune traversarea mai mult de o dată a unui ciclu. Anumite erori apar numai atunci când un ciclu este traversat de mai multe ori.

Traversarea tuturor căilor

Pe baza acestui criteriu ar trebui ca testele să asigure traversarea tuturor căilor componentei testate.

Pentru majoritatea programelor nu se poate aplica acest criteriu, deoarece numărul de căi de execuţie este infinit.

Criteriul poate fi restricţionat, de exemplu asigurând traversarea tuturor căilor cu o lungime mai mică sau egală cu o constantă dată.

Problema alegerii cazurilor de test constă din trei subprobleme distincte:

- selectarea căilor de testat;
- alegerea datelor de test pentru execuţia fiecărei căi selectate;
- determinarea rezultatelor care trebuie să se obţină la fiecare test.

În continuare prezentăm o metodă de alegere a datelor de test pe baza criteriului traversării tuturor arcelor grafului de control. Metoda presupune parcurgerea următoarelor etape:

1. Se determină setul căilor de testat, C, astfel încât:
   a) $\forall$ c $\in$ C este o cale de la nodul de intrare la nodul de iesire;
   b) fiecare arc din graful de control este cuprins într-o cale c $\in$C
   c) Σ lung (c) = minima
      c $\in$ C
2. Se determină predicatul fiecărei căi, c $\in$ C, $R_c(I)$, unde I = (i_1, i_2, …, i_n) este vectorul variabilelor de intrare;
3. Pentru fiecare $R_c(I)$, c $\in$ C, se determină un set de atribuiri X_c pentru variabilele de intrare care satisfac predicatul căii:

$R_c(X_c)$ = true

Atunci setul datelor de test este:

DT = { X_c | c $\in$ C, $X_c \in D_I$, $R_c(X_c)$ = true },

n

unde $D_I = \sum_{k=1}^{n} D_{ik}$, iar D_{ik} este domeniul de valori al variabilei de intrare $i_{k.}$

1. Pentru fiecare $X_c \in$ DT, se determina valorile corecte ale variabilelor de ieşire

Exemplu

Fie urmatorul graf program :

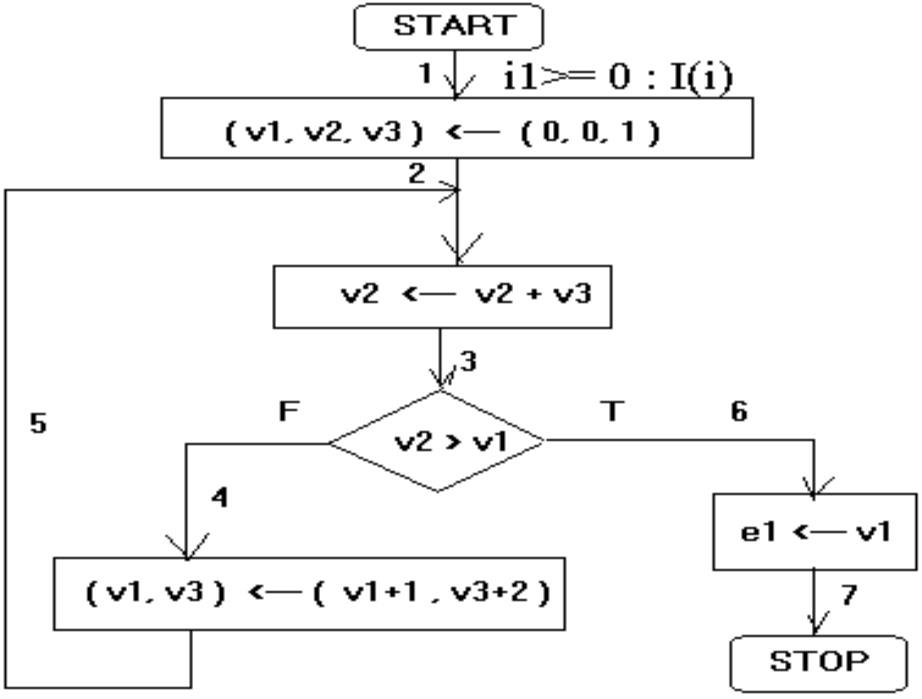


Fig. 8.4. Graf program

Obs: testul v2>v1 trebuie inlocuit cu v2>i1.

1) Alegerea cailor :

Se poate alege setul de căi C:

$$C = \{c_1, c_2\},$$

unde $c_1 = \{1,2,3,4,5,3,6,7\}$

$c_2 = \{1,2,3,6,7\}$

În afara condiţiilor menţionate s-au mai avut în vedere următoarele criterii:

- alegerea unei căi care să nu conţina ciclul
- alegerea unei căi care să conţina ciclul, parcurgându-l o singura dată, pentru ca lungimea căii să fie minimă.

2) **Determinarea predicatelor de cale**

O metodă de determinare a predicatului unei căi constă în concatenarea condiţiilor de ramificare de pe calea respectivă pe măsura ce sunt întâlnite atunci când calea este parcursa de la nodul de

ieşire pană la cel de intrare. Totodată, la întâlnirea unei atribuiri, y ← E, dacă y face parte din expresia curentă a predicatului, se substituie y cu E.

In unele cazuri, predicatul obţinut printr-o singură parcurgere a unui ciclu este fals pentru orice atribuire a variabilelor de intrare. Un astfel de caz corespunde unei căi neexecutabile. In consecintă se va alege un predicat în care ciclul este parcurs de două sau de mai multe ori.

Construirea predicatelor de cale

Rc_1 = v2 > i1

↓ 3 se substituie v_2 cu (v_2 + v_3)

Rc_1 = (v2+ v3) > i1

↓ 5

Rc_1 = (v2+ (v3+2)) > i1

↓ 4

Rc_1 = (v2+ v3+2) > i1 Λ ~(v2 > i1)

↓ 3

Rc_1 = ((v2+ v3)+v3+2) > i1 Λ ~((v2+ v3) > i1)

↓ 2

Rc_1 = (4 > i1) Λ ~ (1 > i1)

↓ 1

Rc_1 = (4 > i1) Λ (i1 >= 1) Λ (i1 >= 0)

<=>

Rc_1 = (1 <= i1 < 4) . Calea c1 este executată pentru orice valoare a lui i1 care satisface acest predicat.

Stabilirea rezultatelor execuţiei fiecărei căi pentru fiecare set de date de test ales, necesită cunoaşterea transformării realizate de componenta analizată asupra datelor de intrare. Aceasta rezultă din specificaţia componentei. In cazul de faţă trebuie să se stabilească ce valoare trebuie sa aibă iesirea e1 pentru fiecare valoare aleasă pentru i1. Programul reprezentat prin graful de control analizat calculează numarul maxim de termeni ai sumei 1+3+5+…+(2j+1), ∀ j >= 0, care pot fi adunati a.î. suma lor să fie mai mica decât i1. Astfel se poate verifica că pentru orice 1<= i1 <=4, e1 = 1, căci dacă s-ar aduna doi termeni, 1+3 = 4, 4>i1.

Cazul de test al căii c1 poate fi: i1=2, e1 = 1.

Calea c2 = {1,2,3,6,7}

Rc_2 = v2 > i1
↓ 3
Rc_2 = (v2+ v3) > i1

$$\downarrow 2$$
$$Rc_2 = 0 + 1 > i1$$
$$\downarrow 1$$
$$Rc_2 = (i1 < 1) \ \Lambda \ (i1 >= 0)$$

<=>

$$Rc_2 = (0 <= i1 < 1) \quad => \quad Rc_2 = (i1 = 0)$$

Cazul de test: i1 = 0, e1 = 0.

“Slabiciunile” testelor structurale sunt:

- testele selectionate depind numai de structura programului; ele trebuie recalculate la fiecare modificare; nu pot fi reutilizate eficient de la o versiune la alta;
- testele selecţionate acoperă cazurile legate de ceea ce “face” componenta testată şi nu de ceea ce “trebuie să facă”; astfel, dacă un caz particular a fost uitat în faza de implementare, testul structural nu releva aceasta deficienţă; el trebuie deci completat cu testul funcţional. Mai exact trebuie să se înceapă cu testarea funcţională care se completează cu testarea structurală.

Testele statistice

Testele statistice se efectueaza in timpul testarii de sistem.

Se numesc astfel testele în care datele de test se aleg aleator, după o lege de probabilitate care poate fi:

- uniformă pe domeniul datelor de intrare al programului testat- aceste teste se mai numesc **teste statistice uniforme**;
- similară cu distribuţia datelor de intrare estimate pentru exploatarea programului – aceste teste se mai numesc **teste statistice operaţionale.**

Rezultatele experimentale ale testelor statistice uniforme sunt inegale.

Pentru unele programe s-au dovedit foarte eficiente, ele conducând la descoperirea unor erori însemnate.

Pentru altele s-au dovedit ineficiente.

O posibilă explicaţie ar consta în faptul că ele asigură o bună acoperire în cazul programelor pentru care domeniile de intrare ale căilor de execuţie au o probabilitate comparabilă.

Ele nu acoperă căile care corespund tratării excepţiilor.

De aceea trebuie completate cu teste folosind date de intrare în afara domeniului intrărilor.

Testele statistice operaţionale sunt în general **teste de fiabilitate**.

Prin ele nu se urmăreşte în general descoperirea de erori ci mai ales comportarea programului în timp.

Căderile observate în timpul acestor teste permit estimarea măsurilor de fiabilitate, cum ar fi MTBF (Mean Time Between Failures).

Estimarea costului unui proiect software

9.1. Costuri şi efort

Estimarea costurilor unei aplicaţii software este greu de realizat cu precizie

- este presupusă o relaţie simplă între cost şi efort;
- efortul poate fi măsurat în luni-om.
- Există o relaţie între efortul necesar şi mărimea produsului (KLOC – kilolines of code, kilo-linii de cod).

Pentru a determina ecuaţiile unui model algoritmic de estimare a costului există mai multe abordări:

1. Un parametru variază în timp ce ceilalţi parametri rămân constanţi şi se determină influenţa parametrului variabil asupra rezultatului
    - De ex. comentariile asupra depanării şi întreţinerii
2. Se analizează datele unor proiecte anterioare
    - De ex. timpul necesar fazelor de dezvoltare, calificarea personalului implicat, mărimea produsului final;

Analiza costurilor permite identificarea unor strategii de creştere a productivităţii software:

- Scrierea de mai puţin cod
- Stimularea oamenilor să lucreze la capacitatea maximă
- Evitarea refacerii componentelor dezvoltate anterior
- Dezvoltarea şi folosirea mediilor de dezvoltare integrate

9.2. Modelul Halstead

Îşi propune o estimare mai obiectivă a mărimii unui program pe baza unui număr de unităţi sintactice: operanzi şi operatori

- Entităţi de bază:

  n_1 = numărul de operatori diferiţi

  n_2 = numărul de operanzi diferiţi

  N_1 = numărul total de apariţii ale operatorilor

  N_2 = numărul total de apariţii ale operanzilor

  Vocabularul: $n = n_1 + n_2$

  Lungimea implementării: $N = N_1 + N_2$

Ecuaţia lungimii: $N' = n_1 \log_2 n_1 + n_2 \log_2 n_2$

Volumul programului: $V = N \log_2 n$

- Variabile

- Estimare empirică a lungimii programului în linii de cod:

LOC = 102 + 5,31 * *VARS*

- Fiecare program care va conţine aproximativ 100 de linii de cod, plus 5 linii suplimentare pentru fiecare variabilă care apare în program.

Generalizarea acestor rezultate la programe mai mari nu este indicată.

În programele mai complexe, factori precum interfaţa dintre componente şi comunicarea necesară între persoanele implicate joacă un rol ce nu poate fi neglijat.

Într-o abordare naivă am putea considera că un proiect ce necesită 60 de luni-om poate fi realizat într-un an cu o echipă de 5 persoane sau într-o lună cu o echipă de 60 de persoane.

Această abordare însă este prea simplistă.

Costurile estimate au deseori o culoare politică, iar rezultatele sunt determinate de argumente care nu au o natură tehnică.

Dacă avem 12 luni pentru a finaliza o lucrare, ea va necesita 12 luni.

Acest motiv poate fi privit ca o variantă a *legii Parkinson*: munca ocupă tot timpul disponibil.

Dacă ştim că o ofertă de 100.000 de euro a fost făcută de concurenţă, noi vom face o ofertă de 90.000 de euro. Acesta este cunoscut sub denumirea de *preţ de câştig.*

Dorim să ne promovăm produsul la un anumit târg de tehnică de calcul şi din acest motiv programul trebuie scris şi testat în următoarele 9 luni, deşi realizăm că timpul este limitat.

Această situaţie este cunoscută sub denumirea de *metoda bugetului* de estimare a costului.

Proiectul poate fi dezvoltat într-un an, dar şeful nu ar accepta acest termen. Ştim că termenul de 10 luni este acceptabil şi atunci îl programăm pentru 10 luni.

Simpla comparare a caracteristicilor unui proiect cu un proiect precedent nu garantează o estimare corectă a costului său.

Dacă o echipă lucrează în mod repetat la proiecte asemănătoare, timpul de lucru necesar va scădea, datorită acumulării experienţei.

În 1968, unei echipe de programatori i s-a cerut să dezvolte un compilator FORTRAN pentru trei maşini diferite.

Rezultate:

Compilatorul	Efortul (în luni-om)
1	72
2	36
3	14

Consultarea experţilor

- Metoda Delphi

Fiecare expert îşi expune opinia în scris.

Un moderator colectează estimările obţinute astfel şi le redistribuie celorlalţi experţi.

Numele experţilor nu sunt asociate cu estimările lor.

Fiecare expert va preda o nouă estimare bazată pe informaţiile primite de la moderator.

Procesul continuă până când se ajunge la un consens

- Distribuţia beta

Un expert realizează mai multe estimări:

- estimare optimistă *a*,
- o estimare realistă *m* şi
- o estimare pesimistă *b*.

Efortul aşteptat va fi:

$$E = (a + 4m + b) / 6,$$

o estimare mai bună probabil decât dacă s-ar fi considerat numai media aritmetică a lui *a* şi *b.*

9.3.Modele algoritmice clasice - Modele liniare

Efortul pentru realizarea unui proiect este:

$$E = a_0 + \sum_{i=1}^{n} a_i x_i$$

- Coeficienţii a_i sunt constante, iar x_i reprezintă factorii care au impact asupra efortului necesar;
- *E* reprezintă efortul (de exemplu, numărul necesar estimat de luni-om);

Modelul Nelson

$$E = -33{,}63 + 9{,}15x_1 + 10{,}73x_2 + 0{,}51x_3 + 0{,}46x_4 + 0{,}40x_5 + 7{,}28x_6 - 21{,}45x_7$$
$$+ 13{,}5x_8 + 12{,}35x_9 + 58{,}82x_{10} + 30{,}61x_{11} + 29{,}55x_{12} + 0{,}54x_{13} - 25{,}20x_{14}$$

Factor	Descriere	Valori posibile
x_1	Instabilitatea specificaţiilor cerinţelor	0-2
x_2	Instabilitatea proiectării	0-3
x_3	Procentajul de instrucţiuni matematice	0-100
x_4	Procentajul de instrucţiuni I/O	0-100
x_5	Numărul subprogramelor	număr
x_6	Utilizarea unui limbaj de nivel înalt	0(da) / 1(nu)
x_7	Aplicaţie comercială	0(da) / 1(nu)
x_8	Program de sine stătător	0(da) / 1(nu)
x_9	Primul program pe această maşină	1(da) / 0(nu)
x_{10}	Dezvoltare concurentă de hardware	1(da) / 0(nu)
x_{11}	Utilizarea dispozitivelor random-access	1(da) / 0(nu)
x_{12}	Maşină gazdă diferită de maşina ţintă	1(da) / 0(nu)
x_{13}	Număr de erori	număr
x_{14}	Dezvoltare pentru o organizaţie militară	0(da) / 1(nu)

Dacă avem o estimare *E*, atunci efortul real *R* va verifica formula:

$$P((1-\alpha)E \le R \le (1+\alpha)E) \ge \beta,$$

unde valori acceptabile pentru α şi β sunt:

α = 0,2 şi β = 0,9.

Exemplu: Să presupunem că estimarea este de 100 luni-om.

Atunci probabilitatea ca proiectul să necesite în realitate între 80 şi 120 de luni-om este mai mare ca 90%.

Există o probabilitate diferită de zero ca efortul real să fie în afara intervalului.

Modelul Wolverton

Este o abordare bottom-up cu o matrice de costuri, care are un număr limitat de tipuri diferite de module şi un număr de nivele de complexitate.

Tipul modulului	**Complexitate**				
	Mică		↔		**Mare**
1. Management de date	11	13	15	18	22
2. Management de memorie	25	26	27	29	32
3. Algoritm	6	8	14	27	51
4. Interfaţă utilizator	13	16	19	23	29
5. Control	20	25	30	35	40

Fiind dată o matrice de costuri C, un modul de tip I, complexitate j şi mărime S_k, va rezulta un cost al modulului $M_k = S_k \cdot C_{ij}$

9.4. Modele algoritmice moderne – Modele neliniare

Forma generală este: $E = (a + b \cdot KLOC^c) \cdot f(x_1, \ldots, x_n),$

Valorile constantelor *a*, *b*, *c* rezultă pe baza unei analize de regresiune a datelor proiectelor disponibile. De obicei, *f* nu se ia în considerare.

Constanta *c*

Autor	Formula
Halstead	$E = 0.7 \cdot KLOC^{1.50}$
Boehm	$E = 2.4 \cdot KLOC^{1.05}$
Walston-Felix	$E = 5.2 \cdot KLOC^{0.91}$

- *c* < 1: analogie cu producţia de masă (costurile fixe se împart pe mai multe unităţi de produs);
- *c* > 1: efortul creşte exponenţial cu mărimea datorită creşterii complexităţii
  - Pentru proiecte mari, această relaţie pare mai plauzibilă.

Exemple

KLOC	Halstead	Boehm	Walston-Felix
1	0,7	2,4	5,2
10	22,1	26,9	42,3
50	247,5	145,9	182,8
100	700	302,1	343,6
1000	22135,9	3390,1	2792,6

Modelul Walston-Felix

A fost creat prin analiza a 60 de proiecte de la IBM.

Proiectele erau complet diferite ca mărime, iar programele au fost scrise în mai multe limbaje de programare.

Au fost identificate 29 de variabile care influenţează productivitatea.

Pentru fiecare din aceste variabile au fost considerate trei niveluri: mare, mediu şi mic.

Variabila	Productivitatea medie pentru valoarea variabilei			PC
Complexitatea interfeţei utilizator	< normală 500	normală 295	> normală 124	376
Calificarea şi experienţa personalului	mică 132	medie 257	mare 410	278
Experienţa anterioară cu aplicaţii similare	minimă 146	medie 221	vastă 410	264
Procentajul de programatori participanţi în faza de proiectare	< 25% 153	25 - 50% 242	> 50% 391	238
Raportul dintre mărimea medie a echipei şi durata proiectului (persoane/lună)	< 0,5 305	0,5 – 0,9 310	> 0,9 171	134

Walston şi Felix consideră că indexul productivităţii *I* poate fi determinat pentru noile proiecte după următoarea relaţie

$$I = \sum_{i=1}^{29} W_i X_i$$

unde ponderile W_i sunt definite astfel:

$$W_i = 0{,}5 \cdot \log_{10}(PC_i)$$

Modelul COCOMO

Modelul COCOMO – **CO**nstructive **CO**st **MO**del a lui Boehm este unul dintre cele mai cunoscute modele de cost care se aplică proiectelor.

Sunt trei clase de proiecte:

- *Organice:* o echipă relativ mică dezvoltă programul într-un mediu cunoscut. Sunt de obicei programe relativ mici.
- *Integrate:* sisteme pentru care mediul impune constrângeri severe (de ex. programe de control al traficului aerian sau aplicaţiile militare);
- *Semidetaşate:* o formă intermediară;

Exemple

Clasa de proiect	**b**	**c**
organică	2,4	1,05
semidetaşată	3,0	1,12
integrată	3,6	1,20

KLOC	organic	semidetaşat	integrat
1	2,4	3,0	3,6
10	26,9	39,6	57,1
50	145,9	239,4	392,9
100	302,1	521,3	904,2
1000	3390	6872	14333

Analiza punctelor funcţionale

Se bazează pe numărarea diferitelor structuri de date utilizate.

Este potrivită mai ales pentru aplicaţiile comerciale, în care structura datelor are o foarte mare importanţă.

Este mai puţin indicată pentru proiectele în care algoritmii joacă rolul dominant (de ex. compilatoarele sau aplicaţiile în timp real).

Analiza se bazează pe modalităţile în care diverşi utilizatori interacţionează cu aplicaţiile.

Se consideră că sistemul îndeplineşte cinci funcţii fundamentale:

- Funcţii referitoare la date
    1. Fişiere interne logice;
    2. Fişiere externe de interfaţă ;
- Funcţii tranzacţionale
    3. Intrări externe;
    4. Ieşiri externe;
    5. Cereri externe;

1. **Fişiere interne logice** – în engleză “Internal Logical Files”, FIL. Permit utilizatorilor să folosească datele pe care trebuie să le întreţină. De exemplu, un pilot poate introduce datele de navigare la un terminal din carlingă înainte de plecare. Datele sunt stocate într-un fişier şi pot fi modificate în timpul misiunii. Pilotul este deci responsabil pentru întreţinerea acestor date.
2. **Fişierele externe de interfaţă –** în engleză “External Interface Files”, FEI. Utilizatorul nu este responsabil pentru întreţinerea datelor; acestea sunt localizate în alt sistem care le întreţine.

Utilizatorul sistemului analizat solicită datele doar pentru informare. De exemplu, un pilot se poate informa asupra poziţiei cu ajutorul sateliţilor GPS sau al sistemelor de la sol. El nu are responsabilitatea actualizării acestor date, însă le poate accesa în timpul zborului.

3. **Intrările externe,** în engleză , “External Input”, IE. Permit utilizatorului să întreţină fişierele interne logice prin operaţii de adăugare, modificare şi ştergere.
4. **Ieşirile externe,** în engleză, “External Output”, EE. Permit utilizatorului să producă date de ieşire. De exemplu, pilotul poate să afişeze separat viteza la sol şi viteza reală în aer, informaţii derivate din datele interne (pe care le poate întreţine) şi cele externe (pe care le poate accesa).
5. **Cererile externe,** în engleză, “External Inquiries”, CE. Pentru ca utilizatorul să poată selecta şi afişa datele din fişiere, el trebuie să introducă informaţii de selecţie pentru a găsi datele în conformitate cu anumite criterii. În această situaţie datele din fişiere nu sunt modificate, ci doar căutate şi furnizate. De exemplu, dacă pilotul afişează date cu privire la relieful solului, date stocate anterior, rezultatul este regăsirea directă a informaţiilor.

Puncte funcţionale neajustate

Prin încercări repetate, s-au stabilit ponderi pentru fiecare dintre aceste entităţi. Numărul de puncte funcţionale neajustate este:

$$PFN = 10^{\cdot}FIL + 7^{\cdot}FEI + 4^{\cdot}IE + 5^{\cdot}EE + 4^{\cdot}CE$$

În funcţie de complexitatea tipurilor de date, se disting o serie de valori pentru aceste puncte funcţionale, prezentate în tabelul următor:

Tip	**Nivel de complexitate**		
	Simplu	Mediu	Complex
FIL	7	10	15
FEI	5	7	10
IE	3	4	6
EE	4	5	7
CE	3	4	6

Puncte funcţionale ajustate

Pentru ajustarea suplimentară a estimărilor, se iau în calcul şi alte 14 caracteristici care influenţează dezvoltarea aplicaţiilor:

- Comunicaţiile de date
- Funcţiile distribuite
- Performanţa
- Folosirea masivă a configuraţiilor
- Rata tranzacţiilor
- Intrările de date online
- Eficienţa utilizatorilor finali
- Actualizările online
- Prelucrările complexe
- Refolosirea
- Uşurinţa la instalare
- Uşurinţa la folosire
- Locaţiile multiple
- Facilitarea modificărilor

Influenţa fiecărei caracteristici este evaluată pe o scară de la 0 (nu influenţează) la 5 (influenţă puternică). Gradul de influenţare *GI* este suma acestor puncte pentru toate caracteristicile

- Se calculează apoi factorul de complexitate tehnică: *FCT* = 0,65 + 0,01 * *GI*
- Punctele funcţionale ajustate (*PF*) se obţin astfel: *PF = PFN * FCT.*

Avantaje

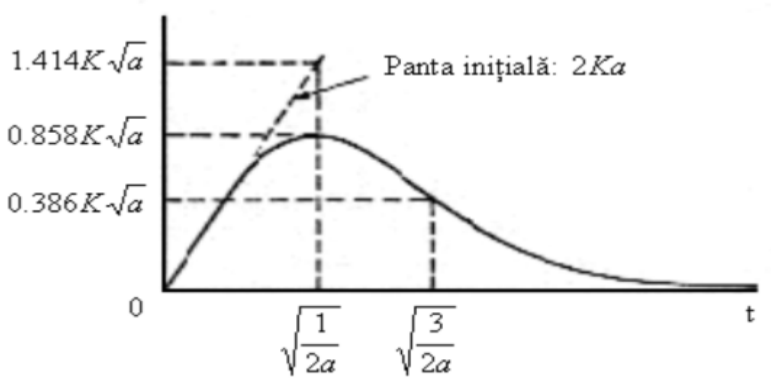
- Măsura productivităţii este un rezultat natural, deoarece punctele funcţionale sunt independente de tehnologie şi deci pot fi utilizate pentru a compara productivitatea pe platforme diferite şi cu instrumente de dezvoltare diferite;
- Ele pot fi folosite pentru a stabili o rată de productivitate (*PF* / *h*) care facilitează estimările privind durata proiectului ca întreg;

Distribuirea forţei de muncă în timp

Modelul Putnam-Norden

Distribuţia forţei de muncă în timp are de multe ori o formă caracteristică, bine aproximată de distribuţia Rayleigh. Astfel, forţa de muncă necesară la un moment de timp *t* este:

$$FM(t) = 2 \cdot K \cdot a \cdot t \cdot \mathrm{e}^{-at^2}$$



Maximul curbei este apropiat de momentul de timp în care proiectul va fi predat clientului.

Acest rezultat este foarte apropiat de o regulă euristică foarte des utilizată: 40% din efortul total este cheltuit pentru dezvoltarea efectivă, în timp ce 60% este cheltuit pentru întreţinere.

Ecuaţia software-ului

Putnam a folosit observaţii empirice legate de nivelurile de productivitate pentru a deriva *ecuaţia software-ului* din curba Rayleigh:

$$\boxed{D = k \cdot E^{1/3} \cdot t^{4/3}},$$

Unde D este dimensiunea proiectului, E este efortul total în ani-om, t este timpul scurs până la lansare în ani iar K este un factor tehnologic bazat pe 14 componente, precum:

- Maturitatea generală a proiectului şi tehnicile de management;
- Gradul de utilizare a tehnicilor de ingineria programării;
- Nivelul limbajelor de programare folosite;
- Capacitatea şi experienţa echipei de dezvoltare;
- Complexitatea aplicaţiei.

Efortul

Pentru estimarea efortului, Putnam a introdus *ecuaţia acumulării forţei de muncă:*

$$A = E/t^3,$$

unde A este numită accelerarea forţei de muncă iar E şi t au semnificaţiile de mai sus.

Accelerarea forţei de muncă este 12,3 pentru proiecte software noi, cu multe interfeţe şi interacţiuni cu alte sisteme, 15 pentru sisteme de sine stătătoare şi 27 pentru reimplementări ale sistemelor existente.

Pe baza celor două ecuaţii putem elimina timpul şi determina efortul:

$$E = (D/k)^{9/7} \cdot A^{4/7}$$

Acest rezultat este interesant deoarece arată că efortul este proporţional cu dimensiunea la puterea 9/7 ≈ 1,286, valoare similară cu factorul Boehm, între 1,05 şi 1,20.

Consecinţe

- Scurtarea timpului de dezvoltare implică un număr mai mare de persoane necesare pentru proiect;
- Referindu-ne la modelul curbei Rayleigh, scurtarea timpului de dezvoltare conduce la mărirea valorii *a*, factorul de accelerare care determină panta iniţială a curbei; vârful curbei Rayleigh se deplasează spre stânga şi în acelaşi timp în sus;
- Astfel obţinem o creştere a puterii necesare la începutul proiectului şi o forţă de muncă maximă mai mare.

Legea lui Brooks

Mai multe studii au arătat că productivitatea individuală scade odată cu creşterea echipei. Conform lui Brooks, există două cauze ale acestui fenomen:

- Creşte timpul acordat comunicării cu ceilalţi membri ai echipei (pentru consultare, sincronizarea sarcinilor etc.);
- Mai întâi scade productivitatea, deoarece noii membri ai echipei nu sunt productivi de la început. În acelaşi timp ei necesită ajutor, deci timp, de la ceilalţi membri ai echipei în timpul procesului de învăţare

- *Legea lui Brooks*: adăugarea de personal la un proiect întârziat îl va întârzia şi mai mult.

Mărimea echipei şi productivitatea

Productivitatea individuală (măsurată în linii de cod pe lună-om) scade cu mărimea echipei.

Mărimea echipei	Productivitatea individuală	Productivitate totală
1	500	500
2	450	900
3	400	1200
4	350	1400
5	300	1500
5,5	275	1512
6	250	1500
7	200	1400
8	1500	1200

Concluzii

- Nu există o relaţie simplă între preţul unui sistem şi costul său de dezvoltare;
- Factorii care afectează productivitatea includ aptitudinile individuale, experienţa în domeniu, natura proiectului, dimensiunea proiectului, instrumentele utilizate şi mediul de lucru;
- Preţul unui produs software poate fi stabilit astfel încât să se câştige un contract şi apoi funcţionalitatea este ajustată conform preţului;
- Pentru estimarea costului pot fi folosite tehnici diferite;
- Modelele algoritmice de estimare a costurilor se bazează pe analiza cantitativă a caracteristicilor proiectelor, permiţând compararea influenţei acestor caracteristici;
- Timpul necesar terminării unui proiect nu este proporţional cu numărul de persoane care lucrează la proiect.

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.

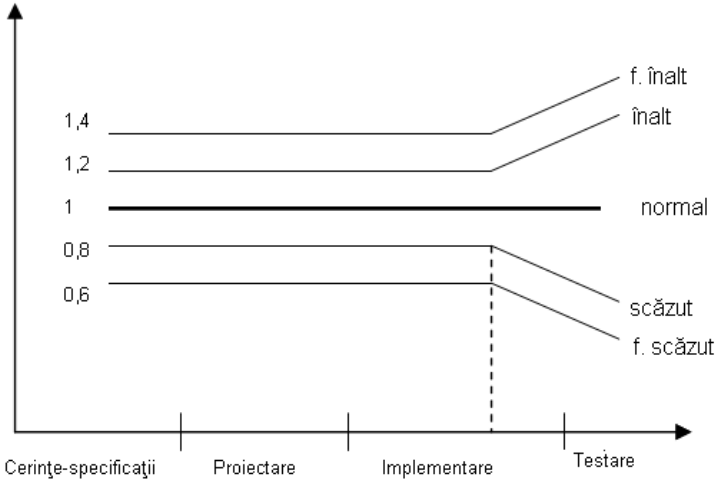


Fig.10.1 Reprezentarea efortului pentru asigurarea nivelului de fiabilitate prescris

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:

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

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.

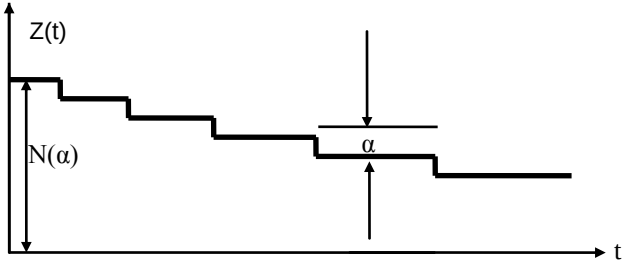


Fig.10.2. Curba lui z(t)

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.

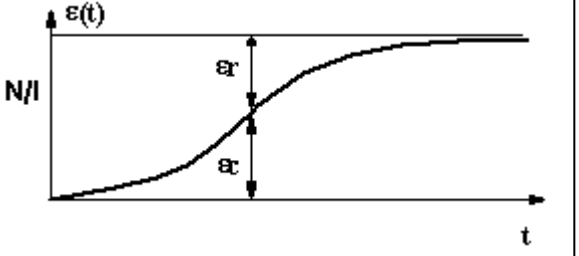


Fig.10.3. Curba erorilor reziduale

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

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.

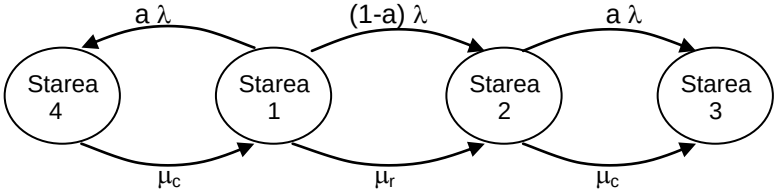


Fig.10.4. Graful de stare

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

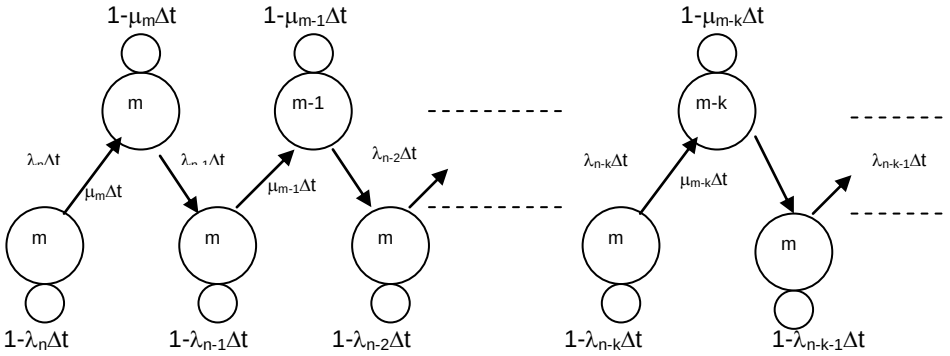


Fig.10.5. Graful de tranziţie în cazul modelului V

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:

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

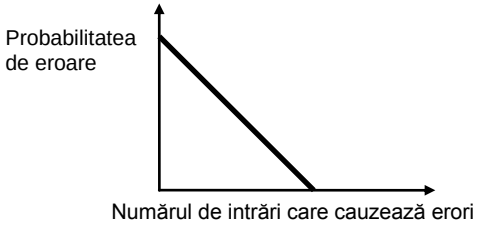


Fig.10.6. Probabilitatea de eroare

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).

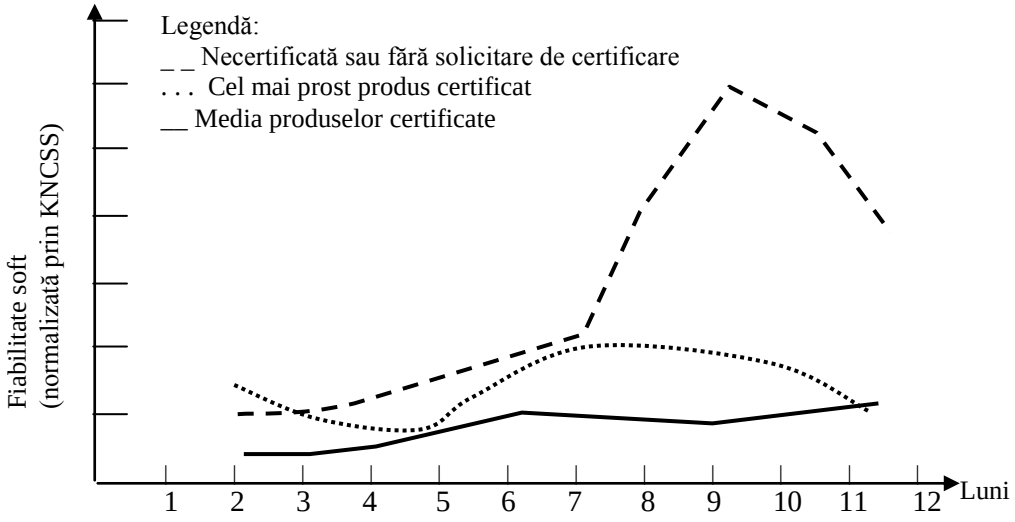


Fig.10.7. Spectrul tehnic al unui program

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.

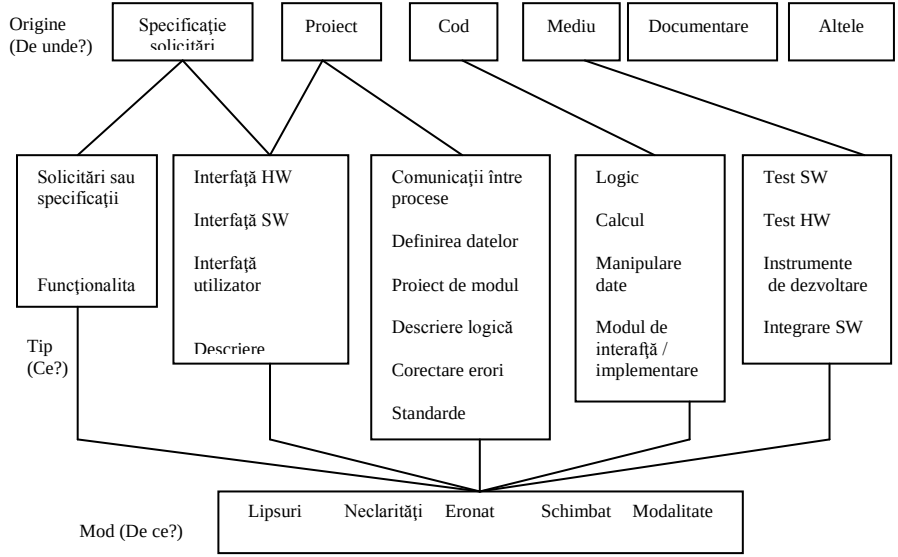


Fig.10.10. Categoriile surselor de defecte software

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.

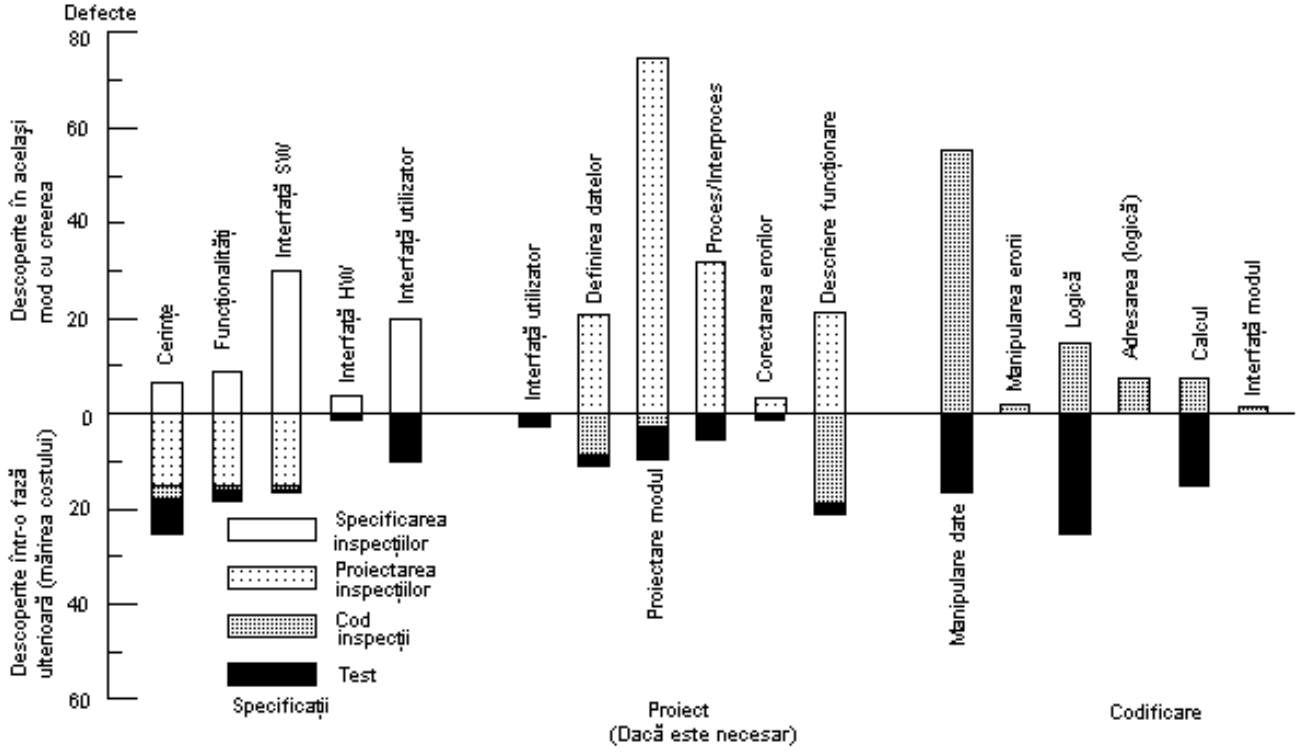


Fig.10.12. Profilul erorilor

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.

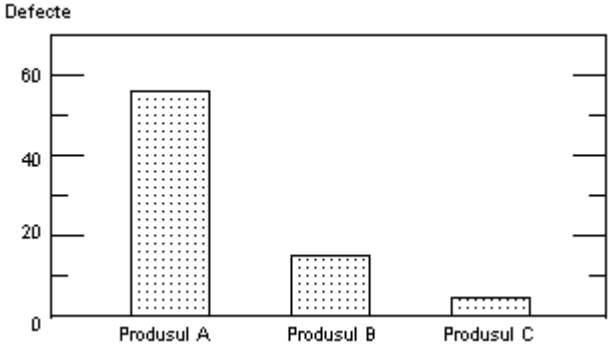


Fig.10.13. Erori de manevrare

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.

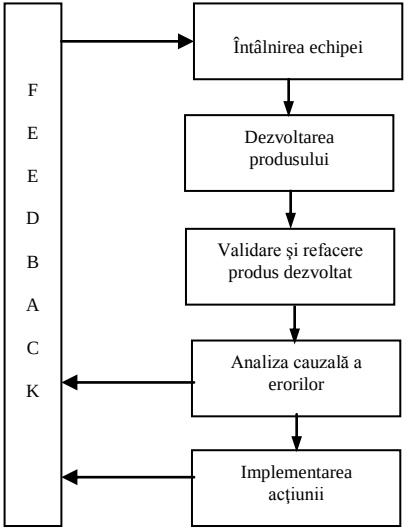


Fig.10.14. Procesul de analiză cauzală

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.

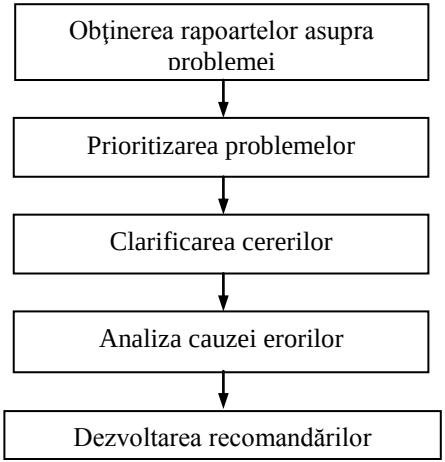


Fig.10.15. Analiza cauzală în procesul de modificare software

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.

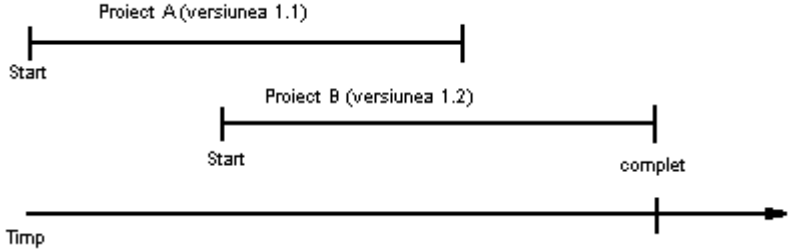


Fig.10.16. Evoluţia sistemului software

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.

Evaluarea sistemelor software

11.1. Set de metrici software pentru conducerea proceselor de mentenanţă a programelor

Problema evaluării sistemelor software a devenit cu atât mai acută şi importantă cu cât dimensiunea, costul şi locul strategic ocupat de sistem a crescut în timp.

Aşa cum fiecărui produs i se ataşează la livrarea către beneficiar un certificat de calitate, tot aşa sistemele software, indiferent de tehnica de realizare (clasică sau din domeniul inteligenţei artificiale), trebuie să fie garantate şi studiate din punctul de vedere al fiabilităţii, calităţii şi întreţinerii lor.

Un studiu de specialitate (Stark, 1994) face cunoscute rezultatele cercetărilor unui grup însărcinat cu administrarea programelor din cadrul NASA, care a definit şi implementat un set de metrici software conţinând 13 metrici destinate să corecteze şi să adapteze acţiunile de mentenanţă.

Echipa numită MOD a răspuns de astfel de activităţi de mentenanţă software la diferite niveluri (sisteme, subsisteme, module) utilizând paradigma solicitare (întrebare) metrică, care identifică o mulţime de 13 metrici pentru utilizarea curentă şi de viitor în studiul sistemelor.

Aceste metrici permit evaluarea stării curente a activităţilor de mentenanţă şi predicţionează rezultate viitoare (solicitări) cum ar fi :

-"Cât de mare trebuie să fie echipa de care este nevoie pentru evaluare? "

-"Cât timp va dura lucrul pentru încheierea raportului software?"

Definirea setului de metrici

Literatura de specialitate în mentenanţă software precizează metricile pentru astfel de activităţi.

De asemenea, în lucrarea lui Gill (1991) se pune problema datelor care trebuie păstrate în timpul mentenanţei sistemului, sau care determină activităţi de mentenanţă (Zuse 1992).

Pornind de la acestea, se pot sintetiza problemele legate de activităţile de mentenanţă a software-ului inteligent ca evoluând pe direcţia "cerere- întrebare- metrici" (Tabelul 11.1).

Tabelul 11.1 **Concluziile privind paradigma cerere/întrebare/metrică**

Cerere	Întrebare	Metrici
Maximizarea satisfacţiei clientului	Câte probleme afectează utilizatorul (clientul)?	- Raportul discrepanţelor (DR) şi solicitările de service (SR) de-a lungul desfăşurării. - Fiabilitate software -Raport întrerupere/ funcţionare
	Cât timp durează pentru a definitiva o problemă?	- Terminarea DR/SR - DR/SR de-a lungul desfăşurării.
	Unde sunt "îngustările"	- Utilizarea staff-ului - Utilizarea resurselor calculatorului
Minimizarea efortului şi proiectului	Unde se consumă resursele?	- Utilizarea staff-ului - Planificarea SR - Instanţieri gen ADA - Distribuţia tipului de erori
	Cât de mentenabil este sistemul?	- Utilizarea resurselor calculatorului - Dimensiunea software-ului - Densitatea erorilor - Volatilitatea soft-ului - Complexitatea software-ului
Minimizarea defectelor	Este software-ul susţinut (confirmat) efectiv inginereşte?	- Densitatea erorilor - Raport întrerupere/ funcţionare - Instanţieri ADA - Fiabilitate software

Observaţii:

Metricile sunt utilizabile individual, dar cel mai mare beneficiu derivă din faptul că ele sunt folosite ca o mulţime de paşi în solicitări de concurenţă cum ar fi calitate faţă de productivitate sau calitatea în raport cu datele realizate, etc.

Pentru a obţine aceste beneficii se raportează mai mult de o metrică la fiecare cerere, iar alte metrici întreţin desfăşurarea cererii (de exemplu fiabilitatea software).

Aceste desfăşurări pot furniza nu numai observaţii din interiorul proiectului, dar permit de asemenea verificări asupra consistenţei datelor.

Şefii proiectului pot să combine metricile pentru a investiga paşi nevizibili numai cu o singură metrică.

1.Dimensiunea software-ului

Mărimea, dimensiunea software-ului este un parametru primar de intrare în mai multe modele de estimare a costului şi calităţii software-ului şi este strâns legată de alte metrici din mulţime (cum ar fi densitatea erorilor, complexitate, etc).

Metrica este definită ca numărul de linii de cod sursă (NLCS) care au fost întreţinute în proiect. Aceasta poate fi utilizată în proiectul de management pentru următoarele activităţi:

Mulţimea de metrici pentru mentenanţa software-ului

1a). Urmărirea efortului necesar pentru întreţinerea codului.

- O creştere în dimensiune a software-ului poate conduce la menţinerea unui personal mai numeros de întreţinere.

- Dacă se poate stabili o relaţie între specificaţiile proiectului, prin utilizarea unui model de cost sau a unei regresii liniare simple, atunci mărimea privind schimbarea staff-ului de mentenanţă poate fi predicţionată prin schimbarea în dimensiune a software-ului.

1b). Anticiparea problemele de performanţă în hard, în timp real.

- O tendinţă către creşterea dimensiunii software-ului ar iniţia acţiuni corective care, fiecare, ar contoriza sau s-ar armoniza cu efortul crescut de depanare şi/sau puterea calculatorului.

2. Echipa de software

Schimbările în echipa de software au un impact direct asupra costurilor proiectului şi asupra progresului în mentenanţa software.

Această metrică este reprezentată de numărul de ore de-a lungul unei luni consumate de inginerii software direct implicaţi în activităţile de mentenanţă şi furnizează informaţii de management legate de datele care ar fi necesare pentru o predicţie a viitoarelor cereri ale staff-ului proiectului.

De asemenea, poate fi folosită pentru următoarele acţiuni:

2a). Să estimeze eficacitatea în activităţile de inginerie software prin evaluarea numărului de paşi şi a resurselor necesare în fiecare proces de mentenanţă.

- Este posibil să se continue sau să se elimine paşi din proces, astfel încât să-l facă mai eficient şi să răspundă mai bine utilizatorilor.

2b). Să identifice cele mai intensive resurse din activităţile de mentenaţă (de exemplu, solicitări de schimbare, evaluare, planificare, implementare, test, eliberare de resurse).

2c). Să identifice cele mai intensive resurse din sistem.

Această metrică permite o "verificare de sănătate" a proiectului prin comparaţii cu regulile de acţiune recunoscute în industrie pentru cheltuiala cu personalul de-a lungul fazei de mentenanţă, din ciclul de viaţă al sistemului software.

3. Procesarea solicitării de mentenanţă

Această metrică monitorizează fluxul activităţii de mentenanţă software şi determină nivelul de satisfacere a clientului.

Maximizând satisfacţia utilizatorului, este important să se manipuleze solicitările cele mai vizibile pentru client, asigurându-i astfel o asistenţă permanentă.

Metrica permite şi unui analist să execute următoarele funcţii:

3a). Să facă o predicţie asupra cantităţii de muncă necesară datorită noilor solicitări, sau noilor rapoarte, comparându-le cu cereri similare din alte proiecte.

3b). Să determine nivelul satisfacerii clientului de către produs, conducând cu grijă evoluţia în echipa de management.

3c). Să efectueze studii de ocupare a personalului, a resurselor calculatorului, reducând posibilele rămăşiţe din modelul de simulare a evenimentelor discrete ale proceselor în cauză.

Această metrică reprezintă o bună cale de a observa tendinţa cumulată a muncii rămase de efectuat, sau pentru a estima cantitatea totală de muncă solicitată. În mod ideal, "progresul" ar trebui să fie aproape zero.

4. Planificarea mărită a software-ului

Metrica de planificare sporită a software-ului trasează mărimea timpului necesar pentru includerea unei cereri sporite şi efortul ingineresc consumat pentru această solicitare.

Sunt folosite două date primare pentru a calcula această metrică:

1). Numărul planificat şi actual de ore de inginerie cheltuite cu acestă suprasolicitare;

2). Timpul calendaristic scurs de la apariţia cererii până la rezolvarea ei, ca facilitate de utilizare.

Mulţimea de metrici include şi această metrică deoarece permite managerilor proiectului să identifice producţiile de plan cu cel mai înalt risc, să evalueze timpul necesar, pentru a oferi utilizatorilor o cerere sporită şi să predicţioneze cantitatea de muncă pe care o incumbă această cerere.

De exemplu, când se primeşte o cerere de service i se asignează o prioritate (aceasta semnifică risc pentru plan, sistem, proiect), datele de care are nevoie şi se estimează personalul necesar pentru satisfacerea completă a task-ului.

Trasând aceste estimări peste actuala desfăşurare de date a solicitării adiţionale, conducerea proiectului poate să-şi modifice procesul estimat şi să furnizeze utilizatorilor o informaţie mai bună în ceea ce priveşte schimbările efectuate în sistem.

5. Utilizarea resurselor calculatorului

Metrica privind utilizarea resurselor calculatorului (CRU) marchează modul de utilizare a resurselor sistemului.

Sunt incluse patru resurse, şi anume: CPU, disc, memorie şi utilizarea canalului I/O. Metricile CRU sunt raportate ca procentaje din capacitatea resurselor şi furnizează un cadru pentru prezentarea concluziilor analizei rezultatelor sistemului în raport cu contractul (Kern, 1992).

De asemenea, avertizează mai devreme conducerea proiectului în cazul în care utilizatorii s-au apropiat de limitele de capacitate ale resurselor sistemului.

Previziunea că volumul de date memorate se apropie de capacitatea de 90% se potriveşte cu o dreaptă de regresie de forma:

capacitate = *rata_de_ocupare* * *luna* + *constantă*, de la punctele datei până la ultima acţiune corectivă.

Această linie intersectează pragul de 90% capacitate în octombrie 1993, când a fost efectuată ultima acţiune corectivă.

6. Densitatea erorilor

Numărul rapoartelor de discrepanţe incluse într-un proiect cu un software fix, per KNLCS, în funcţie de timp, defineşte metrica de densitate sau frecvenţa erorilor. Densitatea erorilor măsoară calitatea codificării şi poate fi utilizată pentru:

6a). Predicţia numărului de erori remanente în cod prin utlizarea unui model de calitate;

6b). Stabilirea densităţii standard de erori în scop de comparaţie şi predicţie pentru fiecare nivel sever de defectare sau tip de eroare.

- Aceasta permite proiectanţilor să identifice sistemele sau procesele cărora trebuie să li se acorde o atenţie deosebită.

7. Volatilitatea software-ului

Belady şi Lehman au definit în 1976, pentru prima dată "volatilitatea software-ului".

Este un raport dintre numărul de module schimbate din cauza cererii de mentenanţă şi numărul total de module eliberate în timp.

Sarcina acestei metrici este de a măsura cantitatea de schimbări în structura software-ului, în timp.

Utilizată împreună cu fiabilitatea, CRU şi complexitatea proiectată, ea ajută managerii să decidă dacă o procedură calificată de test ar putea fi reexecutată şi să identifice nevoia de reconstruire a software-ului.

Se pot stabili două reguli de bază pentru această metrică, şi anume:

(1) Când volatilitatea depăşeşte 30% este necesară o recalificare pentru o utilizare operaţională;

(2) Dacă metrica depăşeşte 80% pentru produsul realizat, sistemul va trebui reconstituit.

8. Durata raportului de discrepanţe

Această metrică (DR) oferă o măsură a timpului necesar pentru rezolvarea tuturor rapoartelor de discrepanţe din momentul descoperirii lor şi până la încheierea proiectului.

Durata unei discrepanţe este calculată din punct de vedere al datelor raportate, minus datele supuse acţiunii şi furnizează o privire din interior a eficienţei procesului de depanare.

O mărime semnificativă a vechilor DR-uri poate indica că au fost necesare mai multe resurse pentru a le închide, a le lichida.

Examinarea numărului şi vârstei DR-urilor cu prioritate critică ridicată permite managerului de proiect să estimeze timpul de răspuns în depanare.

De asemenea, un număr mic de DR-uri poate indica existenţa unor probleme dificile care necesită schimbări extensive pentru corecţie, sau probleme care nu au fost bine înţelese şi solicită mai multă atenţie.

În plus, managerii responsabili cu procesele de discrepanţe păstrează data şi durata vechilor rapoarte atunci când primesc altele noi.

Suma acestor "vârste" şi ciclurile individuale de timp sunt utilizate pentru a identifica ”îngustările” din domeniu şi a îmbunătăţi procesul.

9. Raportul întrerupere/funcţionare

În lucrările de specialitate s-a comunicat că o schimbare în software are numai 20% şanse de succes dacă implică mai mult sau chiar 50 NLCS şi aproape 50% şanse de modificări făcute corect, pentru mai puţin de 10 NLCS într-o primă încercare.

Raportul întrerupere/funcţionare este numărul de erori inserate în soft-ul operaţional, în liniile de bază, împărţit la numărul de modificări făcute în produsul software.

De exemplu, dacă sunt trei discrepanţe care trebuie corectate şi în urma corecţiei rezultă o eroare, raportul va fi 0,33.

Aceasta înseamnă că activitatea de corecţie a avut eficacitate 67% în rezolvarea DR-urilor. Metrica ajută de asemenea la execuţia următoarelor funcţii:

a). Estimarea numărului de probleme care afectează clientul sau a numărului de erori remanente în cod, utilizând un model de simulare;

b). Identificarea sursele care pun probleme pentru software, ca şi dezvoltarea şi aprobarea lui.

Aceasta este, de asemenea, necesară conducerii pentru urmărirea inspecţiilor care privesc codul şi testarea proiectului.

Metrica raportului măsoară eficacitatea organizării mentenabilităţii în ceea ce priveşte modificările în software-ul operaţional de bază.

10. Fiabilitatea software-ului

Fiabilitatea software-ului poate fi definită drept probabilitatea ca software-ul să nu fie defect într-o perioadă specificată de timp şi în condiţii, de asemenea specificate.

Bazându-se pe datele operaţionale de defectare, metrica "fiabilitatea software-ului" furnizează o indicaţie a numărului de erori de-a lungul unei perioade de durată dată.

Ea trasează rata erorii software-ului curent prin date.

Metrica poate fi utilizată să controleze traficul schimbărilor prin sistem astfel încât să fie menţinută o rată acceptabilă a erorilor.

Dacă sistemul este realizat bine în ceea ce priveşte o "mulţime prag" de mentenabilitate, atunci vor fi permise schimbări complexe.

Aproape toate proiectele au o cerinţă legată de fiabilitatea software, deoarece există un punct de la care rata erorilor din software face sistemul să fie inutilizabil.

11. Complexitatea proiectului

Complexitatea proiectului, ca metrică, indică numărul de module cu care complexitatea proiectului a crescut faţă de ceea ce s-a stabilit iniţial.

În acest caz se utilizează metrica complexităţii extinse a lui McCabe, care contorizează numărul de căi de execuţie independente de-a lungul unei piese date de software, ceea ce înseamnă de fapt numărul de ramificaţii logice plus 1.

De asemenea, ea permite conducerii proiectului să schiţeze posibilităţile furnizorului de a menţine un nivel acceptabil al complexităţii, la nivelul fiecărui modul în parte.

Complexitatea este strâns corelată cu efortul de mentenanţă al programatorului şi cu numărul de erori găsite de-a lungul testării şi operării.

12. Distribuţia tipului de erori

Această metrică prezintă raportul desfăşurat de discrepanţe în trei moduri:

(1). Prin desfăşurarea codului (aceasta înseamnă: hardware, software, resurse umane, neputinţă de a fi duplicat ş.a.m.d.);

(2). Prin tipul problemei care a fost greşită (adică logic, computaţional, interfaţă, data de intrare, etc.)

(3). Prin procesul care a introdus problema (adică cereri, proiect, cod, test).

Toate acestea servesc la identificarea acelor aspecte ale procesului care sunt "susceptibile" la erori, precum şi celor mai comune tipuri de erori introduse.

Informaţia provenind de la această metrică trimite înapoi în lanţul de dezvoltare şi conducere, astfel încât managerii pot să ia efectiv măsuri de reducere a riscului (de exemplu o mai bună inspecţie) şi de asemenea să reducă tipul şi cauzele erorilor.

De asemenea, metrica este utilizată pentru identificarea solicitărilor pentru instrumente de mentenanţă software mult mai utile.

Tabelul 11.2 arată comportarea unui proiect MOD.

Tabelul 11.2 **Rapoarte de probleme care au apărut la un proiect MOD**

Desfăşurarea codului	Rapoarte de probleme
Incapabil de a reproduce problema	317
Problema de duplicare	185
Configurare sau limite de proiectare	95
Software fix	265
Erori umane	28
Hardware fix	5
Altele	77

13. Instanţieri tip ADA (limbajul de programare ADA)

Metrica "instanţieri tip ADA" reprezintă un număr de unităţi generice ADA şi codificate de-a lungul activităţii de mentenanţă, dimensiunea unei unităţi generice (în NLCS) şi o mărime a timpului în care unitatea generică este instanţiată.

Cerinţa acestei metrici este de a marca numărul şi dimensiunea componentelor reutilizabile dezvoltate de proiect pentru folosirea lor în interiorul proiectului sau în alte proiecte.

Această metrică marchează unităţile generice ADA reutilizate şi poate fi utilizată atunci când sistemele de inteligenţă artificială folosesc drept "cunoştinţe" de profunzime algoritmi precompilaţi.

11.2. Programul de implementare a metricilor

După definirea mulţimii de metrici trebuie să se treacă la colectarea, analiza, raportarea şi utilizarea metricilor.

Primul pas în implementarea lor este de a emite o directivă de management. Acest pas este considerat pozitiv deoarece asigură consistenţa de-a lungul proiectului şi demonstrează că cel mai înalt nivel al proiectului, suportă efortul.

Importanţa acestui suport nu poate fi exagerată; fără el metricile programului nu ar fi luate în serios, în particular de contractanţi.

Al doilea pas este de a furniza instrumente automate pentru a susţine metricile de program. Se utilizează spreadsheet-uri pentru a rezuma şi raporta metricile datelor.

Observaţii:

Două dintre metricile de bază, şi anume complexitatea proiectului şi fiabilitatea software, sunt dificil de evaluat şi calculat manual, şi de aceea necesită instrumente automate (chiar gen sisteme expert) care să conducă competent astfel de activităţi..

Echipa MOD recomandă ca instrument "Modelarea statistică şi estimarea funcţiilor de fiabilitate pentru software" (SMERFS) pentru măsurarea fiabilităţii software.

Acest instrument include o varietate de domenii incluzând IBM-PC -urile şi calculatoarele compatibile IBM-PC.

Instrumentul de măsurare al complexităţii este UX-metric / PC-metric de la SET Laboratories (1990).

Acesta este un instrument comercial ieftin care contorizează NLCS, comentarii, linii albe şi calculează metricile de complexitate pentru o varietate de limbaje de generaţia a treia incluzând FORTRAN, C şi ADA.

El lucrează pe staţii şi suportă sistemul de operare UNIX la fel de bine ca şi pe PC-uri, IBM compatibile.

O astfel de activitate trebuie organizată sistematic, începând cu proiectarea, achiziţia de cunoştinţe şi continuând cu implementarea, testarea şi mentenanţă sistemelor de programe pentru a putea "garanta" calitatea produsului software elaborat şi a-i demonstra, şi în această manieră, superioritatea faţă de programele tradiţionale.