

UNIVERSITATEA “Vasile Alecsandri”

BACĂU

FACULTATEA DE INGINERIE

CORNELIA NOVAC UDUDEC

**INGINERIA SISTEMELOR DE
PROGRAME**

*Ingineria programării
Ediție adăugită și revizuită*

EDITURA ALMA MATER BACĂU

2011

PREFAȚĂ

„Dacă constructorii ar construi casele în felul în care programatorii concep programe, atunci prima ciocănitoare care ar veni, ar distruge civilizația”.

Acest enunț de altfel un „murphysm”, cunoscut și sub numele de legea lui Weinberg ne poate induce o percepție de loc optimistă asupra modul de concepere și realizare a aplicațiilor software. Situația reală nu este nici pe departe atât de sumbră, dar nici nu putem manifesta un optimism exagerat.

Una dintre “legile” neîncrederii din cunoscuta antologie a lui Murphy se enunță astfel:

„Computerele nu sunt “piese” de încredere, iar oamenii sunt și mai puțin”. Mai mult, “legea” are și următorul corolar:

„La originea oricărei erori care este atribuită calculatorului, vei găsi cel puțin două greșeli umane, incluzând-o pe aceea de a da vina pe calculator”.

“Ingineria software”, prin argumentele pe care le aduce, încearcă să prezinte și celelalte fațete ale problemei. Tocmai pentru că se știe că un “program face ceea ce îi spui tu să facă, iar nu ceea ce ai vrea să facă”, este absolut necesar ca exigențele utilizatorului să se regăsească în final în funcționalitatea pachetului software oferit.

Cert este că, s-a renunțat de mult la amatorism și la proiectarea ad-hoc a sistemelor software fie și numai dacă ne gândim la complexitatea acestora și a faptului că realizarea lor scăpa de sub control.

Mi-am propus în lucrarea de față, intitulată *Ingineria sistemelor de programe*, o denumire mai completă decât vechea, cea de „Inginerie a programării”, să împărtășesc cititorului câte ceva din experiența acumulată și totodată să-i pun la dispoziție un ghid de proiectare și realizare a sistemelor informatice cu metodele și tehnicile actuale. Este motivul pentru care, urmărind ciclul de viață al produsului software, am făcut o trecere în revistă a paradigmelor

din ingineria software, a metodologiei UML, a costului, a calității și a metricilor de evaluare a sistemelor software.

Capitolele 6 a fost dedicat șabloanelor de proiectare, metodologie cunoscută în literatura de specialitate ca Design Patterns.

Cartea se adresează în egală măsură atât specialiștilor, cât și utilizatorilor care doresc să cunoască mai bine caracteristicile de realizare și implementare ale sistemelor de programe.

De asemenea, lucrarea este destinată studenților de la secțiile ingineriești și de la Informatică, care au ca disciplină de studiu "**Ingineria software**" sau „**Ingineria programării**” și pentru care sper să constituie un instrument de lucru util.

Autoarea

Martie 2011

CUPRINS

Capitolul 1 – Sisteme informatice. Probleme și perspective	Pag.
1.1. Introducere	5
1.2. Probleme ale software-ului	11
1.3. Satisfacerea cerințelor utilizatorului și costul software	13
1.4. Performanța, portabilitatea și mentenanța software	16
1.5. Fiabilitatea	18
1.6. Cerințe pentru ingineria sistemelor de programe	18
1.7. Teorema pentru ingineria software	19
1.8. Clasificarea sistemelor de programe	25
1.9. Documentația sistemelor de programe	30
1.10. Dreptul de proprietate și garanții	33
1.11. Exerciții propuse	35
Capitolul 2 – Etapele de dezvoltare a sistemelor de programe	37
2.1. Ciclul de viață	37
2.2. Cerințe- Specificații	46
2.3. Concepte ale specificațiilor de programe	52
2.4. Specificarea formală	57
2.5. Exemple de specificare formală	59
2.6. Exerciții	66
Capitolul 3 – Paradigmele de dezvoltare a sistemelor software	69
3.1. Etapele de dezvoltare software	69
3.2. Paradigmele de dezvoltare software	71

- Metodologia cascadă	78
- Metodologia spirală	81
- Metodologia spirală WinWin	83
- Prototipizarea	84
- Metode formale	88
- Metoda V	89
- Programarea extremă	89
- Metoda Open Source	93
- Reverse Engineering	94
- Metoda de dezvoltare Offshore	94
- Metodologia orientată pe obiect	95
Capitolul 4 – UML- Limbaj unificat de modelare	101
4.1. Introducere în UML	101
4.2. Diagrame și concepte UML	104
- Diagrama claselor	105
- Diagrama cazurilor de utilizare	112
- Diagrama de stare	115
- Diagrama de activitate	119
- Diagrama secvențiale	122
- Diagrama de colaborare	125
- Diagrama de aplicație	129
Capitolul 5 – Principii de proiectare orientată pe obiect	133
5.1. Principiul Deschis-Închis	133
5.2. Principiul substituției Liskov	137
5.3. Principiul Inversării Dependențelor	144
5.4. Stabilitate. Principiul dependențelor stabile	151
Capitolul 6 – Șabloane de proiectare software	161
6.1. Elementele unui șablon de proiectare	161
6.2. Cum rezolvă șabloanele problemele de proiectare	166
6.3. Cum se selectează un șablon	167
6.4. Cum se folosește un șablon de proiectare	169

Capitolul 7 – Proiectarea sistemelor software	173
7.1. Procesul de proiectare	173
7.2. Proiectarea arhitecturală	177
7.3. Proiectarea calității	180
7.4. Modularizarea proiectului	186
Capitolul 8 – Testarea sistemelor software	191
8.1. Introducere	191
8.2. Testarea pe parcursul ciclului de viață al unui program	192
8.3. Testele de sistem	196
8.4. Determinarea cazurilor de test	197
Capitolul 9 – Estimarea costurilor unui proiect software	211
9.1. Costuri și efort	211
9.2. Modelul Halstead	212
9.3. Modele algoritmice clasice – Modele liniare	214
- Modelul Nelson	214
- Modelul Wolverton	215
9.4. Modele algoritmice moderne – Modele neliniare	215
- Modelul Walston – Felix	216
- Modelul COCOMO	217
- Modelul Putnam – Norden	221
- Legea lui Brooks	223
- Mărimea echipei și productivitatea	223
Capitolul 10 – Calitatea sistemelor software	225
10.1. Indicatori de calitate	225
10.2. Productivitatea	228
10.3. Asigurarea fiabilității produselor software	229
10.4. Metrice software pentru paradigma orientată obiect	233
10.5. Modele de studiere „a posteriori” a fiabilității	237
10.6. Modele software pentru reducerea erorilor	247
10.7. Utilizarea datelor eronate pentru îmbunătățirea deciziilor	252
10.8. Reducerea erorilor datorită măsurătorilor	254

10.9. Aplicarea analizei cauzale procesului de modificare a software-ului	256
Capitolul 11 – Evaluarea sistemelor software	267
11.1. Set de metrici software pentru conducerea proceselor de mentenanță a programelor	267
- Definirea setului de metrici	268
11.2. Program de implementare a metricilor	281
Bibliografie	283

1

Sistemele informatice. Probleme și perspective

1.1 Introducere

Știința calculatoarelor, domeniu de activitate recunoscut ca având o dinamică extrem de ridicată, a evoluat în ultimii ani pe coordonate cum ar fi conceperea de noi tehnologii pentru realizarea aplicațiilor software sau noi modalități de lucru în echipă. Dacă în anul 1946 Goldstine și von Neumann apreciau că 1000 de instrucțiuni reprezintă o limită superioară rezonabilă pentru complexitatea problemelor ce pot fi concepute ca rezolvabile cu ajutorul calculatorului astăzi o asemenea dimensiune este atinsă doar cu scop didactic. Mai mult, după ce a estimat în 1981 că nici un program pentru calculatoare personale nu va necesita vreodată mai mult de 640 KB de memorie RAM, Bill Gates, în anul 1995 a recunoscut că lucrurile s-au schimbat mult în ultimele două decenii.

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, 2000000 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.

Creșterea în dimensiune și complexitate a aplicațiilor software a depășit cu mult progresele făcute în domeniul tehnicilor de programare. De aceea, o paralelă între ingineria software și ingineria construcțiilor ar fi atractivă.

Dacă dorim să construim o cușcă pentru câine, putem să mergem în grădină, să căutam lemne și cuie, să luăm un ciocan și să începem să lucrăm. Avem șanse destul de bune să reușim, mai ales dacă suntem îndemânatici. Dacă totuși nu ne iese, putem încerca a doua zi din nou cu alte lemne și alte cuie. Și totuși, dacă câinele nu încapă în cușcă, putem să ne cumpărăm alt câine.

Lucrurile stau radical diferit atunci când dorim să construim o casă pentru familia noastră. În acest caz va trebui sau să angajăm un arhitect care să ne facă un proiect, sau să cumpărăm un proiect standard de casă. Va trebui să 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).

Cu atât mai mult, dacă o firmă plătește câteva milioane de dolari pentru a ridica un zgârie nori, reprezentanții acesteia vor fi foarte atenți cu cine și în ce condiții vor lucra. Ei vor dori garanții că proiectul este viabil, vor angaja mai multe firme de arhitectură pentru a-l verifica. De asemenea, vor fi obligatorii studiile geologice, de fizică a pământului sau meteorologie. Se vor folosi cele mai performante materiale și se vor angaja cei mai competenți și cu experiență constructori. Eșecul nu mai este o opțiune pentru contractantul proiectului.

Știm că inginerii constructori întocmesc planuri, construiesc machete, studiază proprietățile materialelor folosite și fac rapoarte privind progresul operațiunilor. Construcții de o complexitate foarte mare au fost realizate în acest fel într-un mod rațional și economic. “Constructorii” de aplicații software trebuie să procedeze la fel pentru ca dezvoltarea programelor să nu mai fie un proces impredictibil.

Pe măsură ce complexitatea programelor creștea, la sfârșitul anilor '60 începea să se prefigureze deja o criză a software-ului. Un raport prezentat de către o companie de software, în care erau analizate diverse proiecte și 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.

Pentru a contracara aceste tendințe, la conferința organizată de comitetul științific al NATO în anul 1968, a fost propus termenul de *ingineria software* (engl. „software engineering”), într-un mod oarecum provocator. Se dorea ca programarea să împrumute din rigoarea științelor ingineresti pentru a putea livra programe la timp și eficient. Prima definiție dată ingineriei software a fost formulată astfel (F. L. Bauer):

Ingineria software este stabilirea și utilizarea de principii ingineresti solide pentru a obține în mod economic programe sigure și care funcționează eficient pe mașini 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 sau ingineria programării cum a fost cunoscută în România, reprezintă abordarea sistematică a dezvoltării, funcționării, întreținerii, și retragerii din funcțiune a programelor.

Remarcăm că a doua definiție este mai vagă decât prima, întrucât nu face referire la cost și la eficiență. Mai mult, se pare că experiența în general negativă acumulată a făcut să se renunțe la formularea „principii ingineresti solide”, întrucât acestea nu pot fi identificate fără a fi supuse contestațiilor.

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

Se consideră că ingineria software are următoarele caracteristici importante:

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

Programele mici se pot scrie relativ ușor, de către un singur programator, într-o perioadă destul de scurtă de timp. Un program de 100 de instrucțiuni este cu siguranță un program mic. Nu putem identifica precis granița dintre un program mic și unul mare, însă pe măsură ce dimensiunea programului crește, apar provocări noi, calitativ diferite.

Deoarece un singur programator, sau chiar mai mulți, nu pot avea timpul fizic pentru terminarea programului, este necesară crearea uneia sau mai multor echipe de lucru. Este necesară coordonarea și comunicarea între echipe. Complexitatea sistemului software și a organizației care realizează sistemul software devine importantă, putând depăși capacitatea de înțelegere a unui singur individ.

Dacă avem în vedere rapoartele publicate de grupul Standish (www.standishgroup.com) care precizează că în Statele Unite ale Americii se cheltuiesc anual 250 de miliarde de dolari pentru producția de software și că 33% dintre proiectele informatice eșuează, adică 80 de miliarde de dolari se pierd, atunci devine extrem de important modul în care este proiectat și realizat software-ul. Același studiu Standish arată că 83% dintre proiectele informatice au probleme.

Este limpede, în acest context că proiectarea și realizarea aplicațiilor informatice nu pot fi făcute oricum ci trebuie să respecte normele și metodologiile standardizate, în primul rând PMI (Project Management Institute) și apoi pe cele specifice aplicațiilor software.

Termenul de inginerie software, apărut în 1968 și ales ca titlu pentru conferința despre criza software reunită la inițiativa comitetului științific NATO, sugera analogia cu disciplinele tradiționale și avea ca scop atenționarea și

alertarea informaticienilor în privința caracterului rudimentar al tehnicilor folosite de ei în dezvoltarea software-ului.

Dar în timp ce conceptele au progresat și au fost aplicate metodele cele mai eficiente, instrumentele nu au evoluat pe măsură.

Câteva instrumente au apărut foarte repede la începutul anilor '70, cum ar fi generatoarele ARIANE, ATLAS, PAC, dar acelea cu excepția lui PAC, au « sucombat » datorită faptului că nu s-au putut adapta ușor trecerii de la lucrul « batch » la cel conversațional .

Apare ca dezirabilă o abordare riguroasă a problemelor software-ului, care să includă stilul de lucru, modul de scriere a codului etc.

Nerespectarea cerințelor poate avea efecte serioase. Un sistem de livrare a insulinei pentru diabetici poate provoca moartea pacientului dacă nu funcționează corect. Funcționarea incorectă a unui sistem de control al unui satelit poate provoca pagube de milioane de dolari.

Un program este *fiabil* dacă funcționează și continuă să funcționeze fără întreruperi și erori un interval de timp. Această noțiune exprimă de fapt 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 știa ce ar fi trebuit să execute programul, iar comportamentul greșit nu este intenționat.

Iată câteva erori celebre:

- În primii ani în care calculatoarele au fost introduse la stațiile de benzină din SUA, consumatorii primeau cekuri pe sume enorme. Faptul era privit în general cu umor și reclamațiile erau rezolvate repede;
- Sistemul de operare IBM OS360 conținea aproximativ 1.000 de erori la fiecare nouă versiune care încerca să rezolve erorile din versiunea precedentă;
- Un vehicul de explorare a planetei Venus a fost pierdut deoarece programul primit de pe Pământ pentru rectificarea orbitei conținea instrucțiunea 'DO 3 I = 1.3'; instrucțiunea corectă în limbajul FORTRAN ar fi trebuit să conțină virgulă în loc de punct;
- În 1979 s-a descoperit o eroare în programele pentru sistemele de răcire în centralele nucleare din SUA; din fericire, nu fusese niciodată nevoie de execuția rutinelor ce conțineau erorile;
- Din cauza unei erori în sistemul de avertizare împotriva atacului cu rachete balistice, procedurile de contraatac au fost declanșate înainte de a se descoperi că a fost o eroare;
- Racheta Ariane 5 a explodat în iunie 1996 din cauza unei greșeli de programare; costurile s-au ridicat la 500 milioane dolari.

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

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

Ingineria software se ocupă de toate etapele dezvoltării programelor, de la achiziția cerințelor de la client până la întreținerea și retragerea din folosință a produsului livrat. Pe lângă cerințele funcționale, clientul dorește (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, și 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 și 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 și cunoștințele utilizatorului.

Optimizarea tuturor acestor atribute e dificilă deoarece unele se exclud pe altele (de exemplu, o mai bună interfață pentru utilizator poate micșora eficiența produsului). În cazurile în care eficiența este critică, acest lucru trebuie specificat explicit încă din faza de preluare a cerințelor utilizatorului, precum și 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 greșită" (în engl. "bad design"). Pentru a rezolva problema unui design greșit trebuie mai întâi definită această problemă și analizate cauzele ei.

Definiția unei "proiectări greșite"

O piesă software care satisface cerințele impuse, legate de funcționalitate, are un "design greșit", dacă îndeplinește 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 numește **fragilitate**;

3. Este dificil de refolosit în altă aplicație pentru că nu poate fi detașată de aplicația curentă. Această caracteristică se numește **imobilitate**.

Cauze ale unui "design greșit"

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

Un design este **rigid** dacă nu poate fi modificat cu ușurință. Această 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 de către proiectant sau de către persoana care se ocupă de mentenanța sistemului, atunci impactul modificării nu poate fi estimat, și 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 dorește a fi detașat și 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 pentru un sistem software poate duce la dezvoltarea unui design greșit. De asemenea, modificarea cerințelor înseamnă modificarea specificațiilor. Specificarea cerințelor se realizează într-un document scris care trebuie să poată fi citit și înțeles atât de beneficiar cât și de proiectant. Altfel, beneficiarul nu poate să-l avizeze. Există mai multe motive pentru care cerințele se schimbă:

- Beneficiarul (clientul) software-ului vine cu noi cerințe, fie în sensul adăugării de noi funcționalități aplicației, fie în sensul modificării funcțiilor deja specificate;
- Analistul sau persoana care se ocupă de proiect (și a formulat inițial cerințele aplicației) a interpretat greșit cererile clientului;
- La inițierea oricărui proiect, se discută cu beneficiarul software-ului despre așteptările acestuia de la viitoarea piesă de software, scopul fiind formularea cât mai exactă a cerințelor sale. Aceste cerințe sunt notate și analizate. În timpul acestui proces, unele din cerințele inițiale ale clientului pot fi "uite". În faza de feedback de la client, aceste cerințe sunt reconsiderate. Ca urmare, apar modificări ale programului;
- Există numeroase alte cauze ale modificării cerințelor unei aplicații. Spre exemplu, cauze care nu sunt legate direct de proiect, dar care se răsfrâng asupra acestuia: o lege nouă, o nouă politică de aprovizionare a firmei, o reorganizare sau o fuziune a firmei pentru care este dezvoltată aplicația, etc. Toate aceste aspecte pot modifica cerințele aplicației. 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 și cele cu design greșit sunt supuse modificărilor. Diferența dintre ele este că proiectarea bună este stabilă când sunt supuse modificării.

1.3 Satisfacerea cerințelor utilizatorilor și costul software-ului

Nici un produs, deci nici produsele software, nu vor fi solicitate și utilizate dacă ele nu răspund unor nevoi ale utilizatorilor. Cu cât sunt mai bine acoperite cerințele celor care beneficiază de facilitățile produsului software respectiv și 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 creștere. Acest cost este influențat, și determinat totodată, de o serie de factori cum ar fi:

productivitatea de programare, 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 în mare parte determinat de nivelul salariilor celor implicați în producția de software. Productivitatea medie a programatorilor este de 10-20 instrucțiuni (într-un limbaj de programare) pe zi. Acestea includ clarificarea specificațiilor problemei, proiectarea programului, codificare, testare și documentare. Surprinzător, această productivitate nu depinde de limbajul utilizat. Productivitatea este influențată de lucrul individual sau în echipă al programatorului și de tipul aplicației proiectate (software de aplicații sau software de bază). Se știe, de exemplu, că sistemele de operare se realizează mai greu decât diversele aplicații software.

Costul software-ului comparativ cu cel al hardware-ului a iscat controverse de-a lungul timpului. Faimoasa curbă "S - shaped" a arătat schimbările relative intervenite în cost de-a lungul anilor. De exemplu, în anii '55 software-ul reprezenta aproximativ 10% din costul proiectului (fig. 1.1).

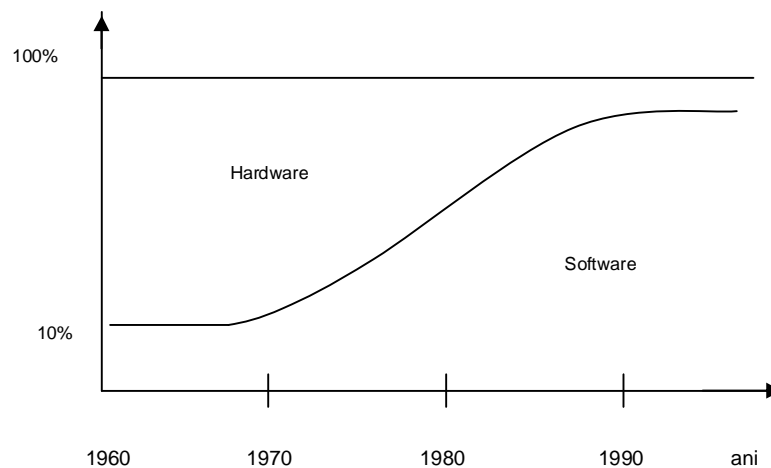


Fig.1.1. Relația între costul hardware-ului și software-ului în timp

Odată cu impactul social al microcalculatoarelor (al calculatoarelor personale), percepția omului obișnuit în ceea ce privește costul software-ului s-a modificat.

"Dacă copilul meu poate să scrie un program-joc în două săptămâni, DE CE este software-ul așa de scump ?" , iată o întrebare la care nu se poate răspunde foarte ușor, pe înțelesul tuturor.

În altă ordine de idei, cheltuielile legate de producția de software sunt influențate de utilizarea "pachetelor de programe" specializate pentru diferite probleme (ex. grafică, interfețe utilizator inteligente, etc.), precum și de folosirea generatoarelor de programe de aplicații.

Rezumând, se poate spune că software-ul este scump dacă ne raportăm la produsul național brut (este vorba de țările puternic industrializate) în primul rând din cauza productivității scăzute a programatorilor. Este ceea ce percepe omul obișnuit. Astfel se naște, 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.

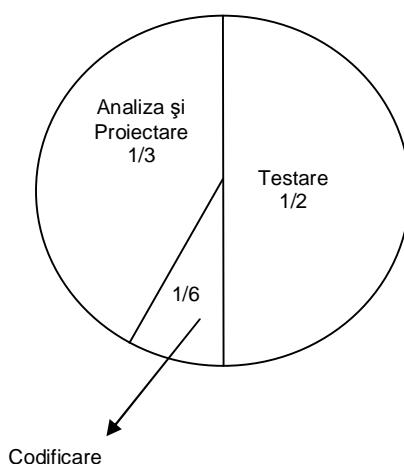


Fig. 1.2 Costul relativ în diferite stadii de dezvoltare ale software-ului

Dacă totuși 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 și constă în a stabili cât costă depistarea unei erori. Se știe că o eroare nedescoperită costă mai mult decât ar fi costat ea dacă ar fi fost descoperită și înlăturată la timp.

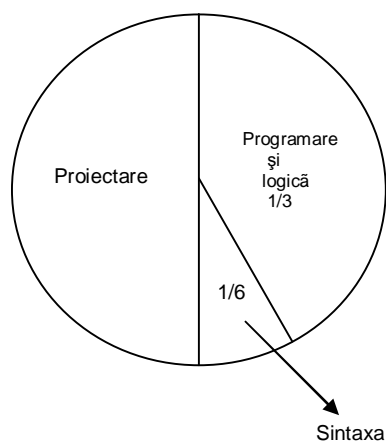


Fig. 1.3. Numărul relativ de erori de-a lungul stadiilor de evoluție ale software-ului

Erorile de sintaxă sunt descoperite automat de către compilatoare, la prima compilare, și pot fi corectate cu ușurință. 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.

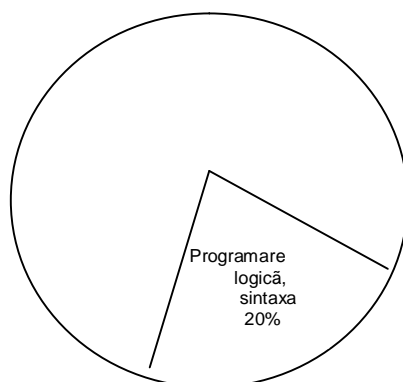


Fig.1.4 Costul relativ pentru depistarea diferitelor tipuri de erori software

1.4 Performanța, portabilitatea și mentenanța software-ului

Performanța unui program este numită adesea *eficiență*. Această 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 și procesorului central, printr-un program cât mai eficient care conducea în final, la scurtarea timpului de execuție

și 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 ieșire (într-un timp rezonabil);

Timpul scurt de execuție și utilizarea unei memorii mici sunt două cerințe mutual contradictorii. De regulă există programe lente, dar mici ca memorie ocupată, sau rapide, dar de dimensiune mare. În general, este necesar să se facă o apreciere adecvată a cerințelor de performanță particulară a unei "piese" software. Visul producătorilor de software a fost întotdeauna de a transfera software-ul de pe un tip de calculator pe altul, cu minimum de efort. Odată cu apariția limbajelor de nivel înalt și cu stabilirea de standarde internaționale s-a ajuns la o portabilitate completă, în majoritatea aplicațiilor software.

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.

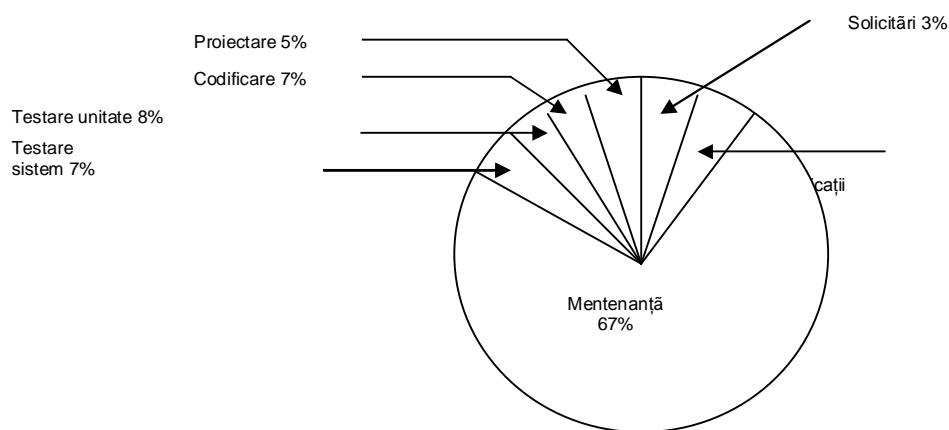


Fig.1.5 Proportia activităților în cadrul realizării unui produs software

1.5 Fiabilitatea

În termeni generali fiabilitatea software reprezintă capacitatea sistemului de a răspunde cerințelor utilizatorului (de a-și 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.

De exemplu: în cazul sistemului OS-360, sistemul de operare lansat de IBM, după depistarea unui număr de 1000 erori, firma lansează o nouă versiune.

În timpul desfășurării programului spațial american, un vehicul spațial fără om a fost trimis spre planeta Venus. A fost necesară o corecție de traiectorie, iar calculatorul, care avea misiunea de control, trebuia să execute următoarea instrucțiune:

DO 3 I=1.3

Aceasta este o instrucțiune de atribuire perfect validă în FORTRAN, dar programatorul avea intenția să execute o structura repetitivă (DO) de trei ori (I=1,3), numai că în loc de virgulă după cifra 1 (care desemna valoarea inițială a contorului) a pus punct . Calculatorul a executat deci în locul unui ciclu, o instrucțiune de atribuire, dând variabilei DO3I valoarea 1.3 ! Ca urmare naveta spațială a luat o traiectorie greșită și nu a mai fost găsită niciodată. De notat că acest program a fost compilat și testat cu succes!

Cu câțiva ani în urmă, sistemul de securitate american a intrat în alarmă sesizând un atac împotriva S.U.A. S-a dovedit a fi o alarmă falsă ca urmare a unei erori a computerului, dar până a fost descoperită, populația a avut "o zi lungă", intrând într-o procedură de autoapărare .

1.6 Cerințe pentru ingineria sistemelor de programe

Așa cum a reieșit din cele expuse până acum există câteva cerințe obligatorii pentru ca un produs informatic, un sistem software să fie utilizat și 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, așa cum se poate observa în fig. 1.6. sunt în conflict, adică nu pot fi satisfăcute simultan la maximum. În consecință, în funcție de domeniul de aplicație și de cerințele problemei se va face o ierarhie a cerințelor acordând prioritate maximă celor care sunt critice pentru sistem.

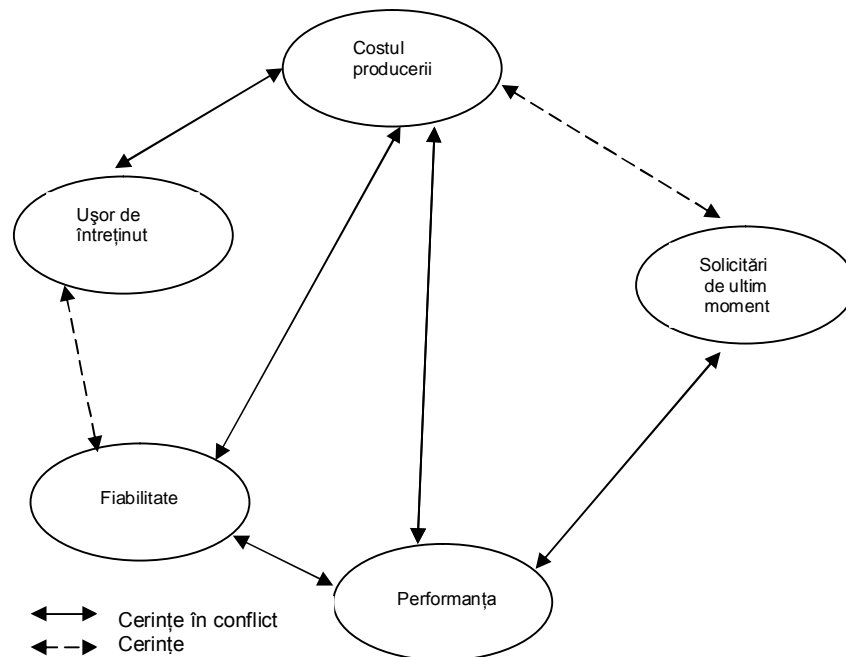


Fig. 1.6 Cerințe complementare și în conflict în ingineria software

1.7 Teoremă pentru ingineria software

S-a amintit în paragrafele precedente despre erorile care apar în sistemele software mai ales atunci când acestea sunt de complexitate mare. Mai mult, dimensiunea acestora are influență directă asupra costurilor de realizare ale software-ului.

Să presupunem că avem o măsură pentru dimensiunea unei probleme P , notată cu $M(P)$, iar costul scrierii unui program pentru problema P este $C(P)$. Dacă P și Q sunt două probleme pentru care avem $M(P) > M(Q)$ atunci între costuri există relația $C(P) > C(Q)$. Deci, cu alte cuvinte costul elaborării programelor, într-o primă aproximație este o funcție monoton crescătoare de dimensiunea programelor.

Fie acum două probleme separate și distincte, notate P și Q , și vrem să creem un program combinat. Luând împreună cele două probleme vom obține o problemă nouă $P+Q$, a cărei dimensiune este

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

Între costuri va exista relația : $C(P+Q) > C(P) + C(Q)$.

Rezultă că este mai ușor de creat două programe mici decât unul mare care cumulează funcțiile ambelor programe. Aceasta se explică prin luarea în considerație a complexității programelor. Pe măsură ce complexitatea crește, pot apare și mai multe erori, generate și de interacțiunea dintre programe. Fenomenul menționat este caracteristic tuturor domeniilor în care se rezolvă probleme.

În toate cazurile la o ușoară creștere a complexității unei probleme (plecând de la o problemă simplă) se remarcă o creștere ușoară a numărului de erori. Mai devreme sau mai târziu însă, numărul de erori începe să crească foarte repede.

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

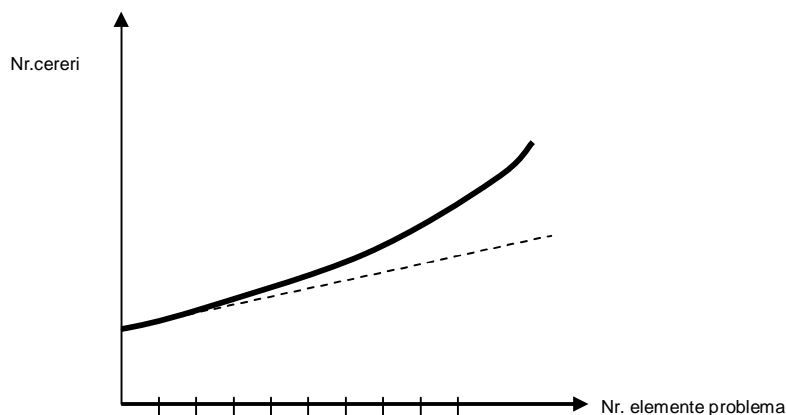


Fig.1.7. Curba erorilor la descompunerea problemei în subprobleme

Acest efect este determinat de limitările ființei umane în prelucrarea informațiilor. Se pare că suntem capabili la un moment dat să prelucrăm simultan și complet informații referitoare numai la aproximativ șapte obiecte, entități sau concepte distincte. Peste 7 ± 2 entități distincte ce trebuie folosite simultan, numărul de erori comise crește mult mai repede. Dacă problema este descompusă în subprobleme atunci numărul erorilor crește mai lent (linia punctată).

Tocmai această relație dintre elementele problemei și generarea erorilor justifică inegalitatea $C(P + Q) > C(P) + C(Q)$. Aceasta înseamnă că în cazul problemelor mari pentru a învinge complexitatea cu un cost mai redus trebuie să se recurgă la descompunerea problemei în module mai mici și 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' și P'' sunt două subprobleme independente prin a căror rezolvare se soluționează la un cost mai scăzut problema P . Dacă cele două probleme nu sunt chiar independente, atunci trebuie soluționată și interferența dintre ele.

Procesul de descompunere în componente mai simple, relativ independente, introduce noi surse de erori, mai ales legate de comunicarea dintre ele. Erorile introduse sunt în general mai simple și mai ușor de localizat.

În aceste condiții ne putem aștepta la o curbă a erorilor care crește mai lent în raport cu numărul de elemente ale problemei ce trebuie prelucrate simultan (linia punctată din fig.1.7).

Să considerăm următorul caz:

Trebuie proiectat un produs program cu 5000 linii (instrucțiuni). El poate fi realizat ca un singur modul cu un cost ridicat sau descompus în 10 module a 500 de linii fiecare, ș.a.m.d. În cazul extrem se poate realiza și din 5000 de module a câte o instrucțiune fiecare.

Natural, costul de realizare depinde de dimensiunea modulului și va fi cu atât mai mic cu cât dimensiunea este mai mică (curba 1 din fig.1.8).

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

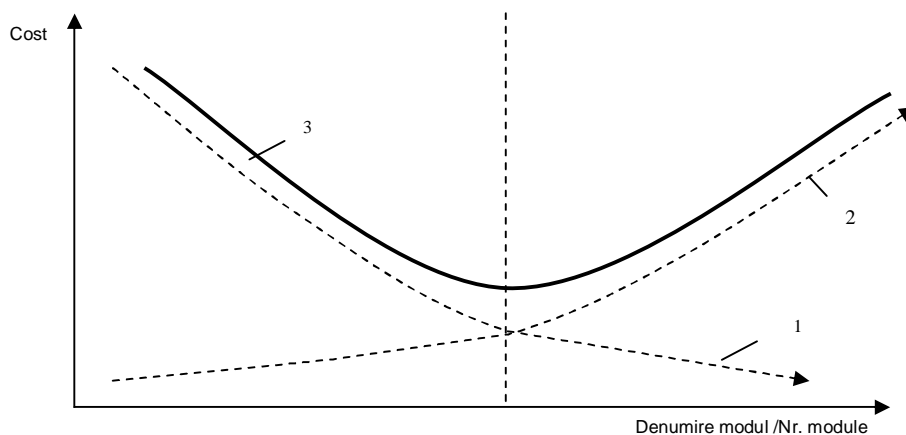


Fig.1.8. Relațiile cost-dimensiune modul și cost-număr module

În schimb, pe măsură ce produsul se descompune în mai multe componente, apar efecte noi datorită interferențelor, în număr mai mare, dintre module. Cu cât crește numărul modulelor, cu atât crește și costul realizării interfețelor dintre module (curba 2 din fig.1.8).

Costul realizării întregului program (deci și numărul erorilor comise) este suma dintre cele două tendințe opuse de realizare (curba 3 din fig.1.8).

Ceea ce se poate concluziona de aici este existența unui cost optim de realizare care determină în ultimă instanță o dimensiune optimă de modul și 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. O soluție de proiectare care adoptă ca bază principiul modularității își propune să găsească acest optim.

Termenul de inginerie software, apărut în 1968 și ales ca titlu pentru conferința despre criza software reunită la inițiativa comitetului științific NATO, sugera analogia cu disciplinele tradiționale și avea ca scop atenționarea și alertarea informaticienilor în privința caracterului rudimentar al tehnicilor folosite de ei în dezvoltarea software-ului. Dar în timp ce conceptele au progresat și au fost aplicate metodele cele mai eficace, instrumentele nu au evoluat pe măsură. Soluțiile la problemele software-ului expuse până acum nu sunt mutual exclusive, ci din contra se completează unele pe altele.

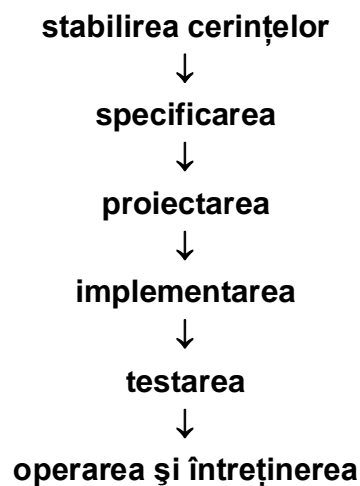
Ele sunt :

- 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 ;
- creșterea efortului pentru asigurarea că software-ul nu are erori .

Acestea constituie totodată și principalele obiective ale cărții de față, care vor fi tratate pe rând în continuare .

Una dintre ideile dominante în abordarea dezvoltării software-ului este dezvoltarea pe baza ciclului de viață sau modelul « waterfall ». În acest context se împarte dezvoltarea unui produs informatic în pași independenți, aflați într-o anumită succesiune.

Pașii succesivi sunt:



Specialiștii 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 pași ;
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.

Specificarea formală, verificarea, prototipizarea, precum și alte tehnici actuale se adresează numai unei clase de probleme luate în considerare în sistem, în ciclul de viață software.

Pe scară largă, proiectul va cuprinde un număr de activități legate între ele cum ar fi : analiza, specificarea, proiectarea, implementarea și altele.

În mod categoric, dacă vrem ca proiectele software să aibă succes atunci ele trebuie înainte bine planificate și efectiv controlate în fiecare etapă de execuție. Trebuie înlocuite metodele « ad-hoc » printr-o disciplină organizată. Unul dintre termenii care este utilizat foarte des astăzi în conexiune cu software-ul este calitatea acestuia. În general, orice produs care corespunde scopului pentru care a fost realizat este considerat un produs de calitate. În contextul realizării software-ului dacă un pachet de programe întrunește așteptările utilizatorilor poate fi considerat un produs de calitate.

Calitatea poate fi atinsă numai dacă există și sunt aplicate efectiv standarde, tehnici și proceduri care să funcționeze și să fie monitorizate. Aceste activități poartă numele generic de « asigurarea calității ».

Problema producerii unui software corect poate fi abordată prin utilizarea unor tehnici adecvate de specificare și verificare (formală și informală) . Dar corectitudinea este numai un aspect al calității. Utilizarea explicită a unei discipline de conducere a proiectului este factorul cheie în obținerea unei calități software ridicate.

1.8 Clasificarea sistemelor de programe

Clasificarea programelor presupune împărțirea acestora în diverse categorii în funcție de anumite criterii. Exactitatea acestor clasificări depinde în mare măsură de precizia cu care aceste criterii pot fi cuantificate și precizate. O astfel de clasificare în funcție de *dimensiunea* programelor împarte programele în: programe mari și programe mici. Limita dintre aceste clase este vagă.

Se consideră de obicei că programele mici sunt acelea realizate de o singură persoană, fără a fi neapărat descompuse în module, într-un interval de timp scurt. În această categorie intră programele realizate de începători în cursul procesului de instruire.

În categoria programelor mari intră în acest context toate programele ce nu se încadrează în limitele convenite pentru programele simple.

O altă schemă de clasificare a fost introdusă de E.Yourdan având, la bază ceea ce el numește *complexitatea tratării programelor*.

Pentru a face o ierarhizare după acest criteriu, Yourdan ia în considerare mai mulți parametri, cum ar fi:

- dimensiunea programelor exprimată în funcție de linii sursă;
- numărul de programatori ce participă la realizarea programelor;
- durata de timp afectată realizării programului;
- intensitatea interacțiunii cu alte programe sau sisteme.

Cerințe deosebite: fiabilitate ridicată, complexități suplimentare.

Yourdan distinge următoarele categorii de programe:

1) *Programe simple*. Această categorie cuprinde programele care:

- au mai puțin de 1000 linii sursă;
- sunt scrise de un singur programator în cel mult 6 luni;
- nu au interacțiuni cu alte programe sau sisteme.

2) *Programe de complexitate medie*

- au mai puțin de 10 000 linii sursă;
- sunt scrise de 1-5 programatori în cel mult 2 ani;
- au interacțiuni puține sau deloc cu alte sisteme;
- constau în general din 10-100 module;

Exemple: cea mai mare parte a aplicațiilor existente.

3) Programe complexe

- au mai puțin de 100 000 linii sursă ;
- sunt scrise de 5-20 programatori în 2-3 ani;
- sunt structurate în mai multe sisteme;
- interacționează cu alte sisteme;
- cuprind în general între 100-1000 module.

Exemple: compilatoarele pentru limbajele de nivel înalt, SGBD-uri, aplicații de timp real.

4) Programe deosebit de complexe:

- au între 100 000-1 milion de linii sursă;
- sunt scrise de o echipă formată din 100-1000 programatori pe parcursul mai multor ani;
- constau din 1000-10 000 module;
- interacțiuni complexe cu alte module ;
- necesită prelucrări de timp real, telecomunicații, multitasking.

Exemple: Sisteme de operare de tip OS/360, VS/370 dezvoltate de IBM, UNIX, sisteme informatice naționale, etc.

5) *Programe aproape imposibile*: Deși puține la număr prezintă următoarele caracteristici:

- au între 1-10 milioane de instrucțiuni;
- pentru realizare sunt necesare echipe de mai mult de 1000 programatori, pe perioada mai multor ani (10 ani sau mai mult);
- includ întotdeauna prelucrări de timp real;
- necesită teleprelucrare de date;
- sunt implicate în procese critice (ex.control de trafic aerian, apărare spațiu aerian);

Pentru un astfel de proiect în Australia s-a specificat un timp mediu între două căderi consecutive de 47 ani.

Deficiența majoră a celor două sisteme de clasificare prezentate mai sus constă în dificultățile mari întâmpinate în definirea diverselor clase de programe.

Există în literatura de specialitate un alt sistem de clasificare mai satisfăcător, care se bazează pe recunoașterea faptului că orice program este *un alt model al unui model teoretic pentru o abstractizare a unei porțiuni din lumea reală*.

Conform acestei clasificări programele sunt împărțite în 3 categorii, S, P și E. Deoarece programele considerate mari de clasificările anterioare intră, în general, în clasele P sau E, noua schemă de clasificare reprezintă o reluare pe alt plan a punctelor de vedere anterioare.

A) Programe S - Programe a căror funcție este definită complet printr-o specificație. Programarea pe bază de specificație este forma din care derivă cele mai avansate metodologii de programare. De precizat că toate programele mari sunt construite ca structuri de programe S.

Așa cum se remarcă din fig.1.9, specificația, ca o definiție formală a problemei, cuprinde toate datele care stau la baza procesului de creare a programului. Programul astfel obținut constituie în final o soluție a problemei. El este într-adevăr o soluție deci prin execuție pe calculator ieșirile sunt deduse din intrări, în conformitate cu specificația.

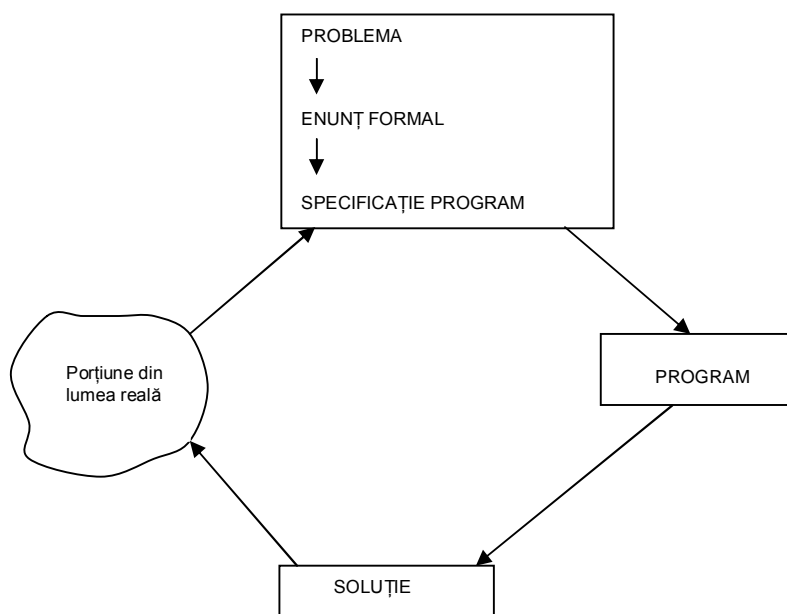


Fig.1.9. Programe S

În aceste condiții poate fi modificat pentru a-și îmbunătăți claritatea, pentru a-i reduce resursele necesare în timpul execuției, sau chiar pentru a crește încrederea în corectitudinea sa, însă toate modificările nu trebuie să-i afecteze corespondența între intrări/ieșiri pe care o definește și care este efectiv realizată în cursul execuției.

De fiecare dată când programul este modificat trebuie aratat că relația intrări-ieșiri nu s-a schimbat sau că noul program satisface o altă specificație, definind astfel o soluție a unei noi probleme.

B) Programe P – Există programe al căror enunț este un model al unei abstractizări a unei situații din lumea reală, conținând incertitudini, necunoaștere, criterii arbitrare, variabile continue, etc. Într-un anumit sens un astfel de enunț reflectă punctul de vedere personal al analistului și deci atât enunțul problemei, cât și soluția ei aproximează situația din lumea reală. Exemplu: jocul de șah, prognoza vremii, etc.

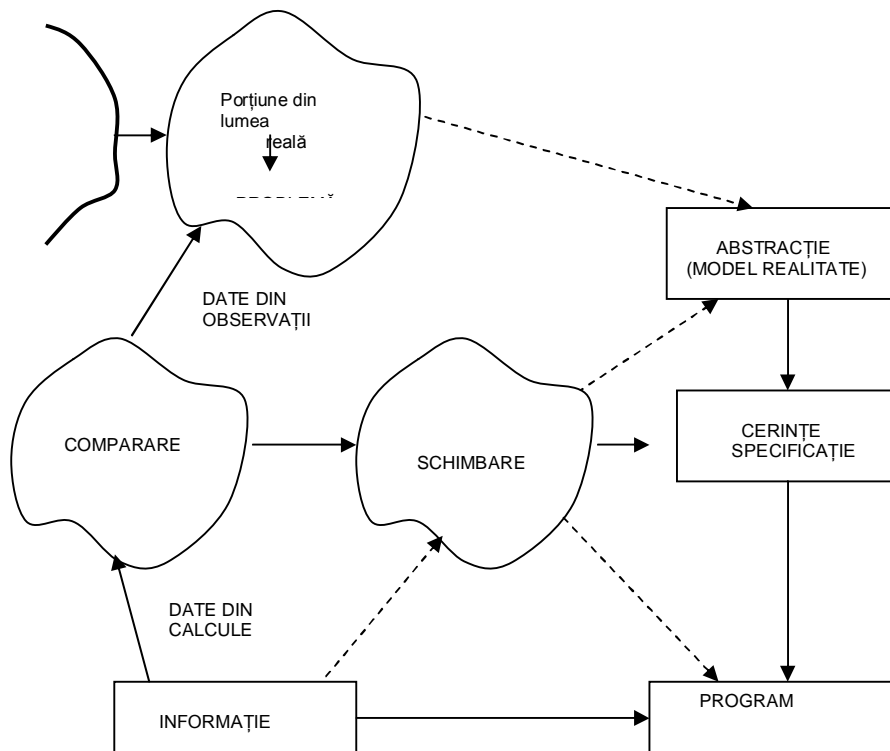


Fig.1.10. Programe P

Deși problema de rezolvat poate fi precis definită, acceptabilitatea unei soluții este determinată de contextul în care este folosită.

Soluția obținută este evaluată prin comparații cu mediul real și nu cu specificația. Aceasta este diferența esențială dintre programul S și P.

Deci dacă în cazul programelor S corectitudinea este legată, prin definiții, numai de specificații, în situația programelor P aceasta este strâns legată de valoarea și valabilitatea soluției obținute în contextul sau din lumea reală.

Diferențele dintre datele obținute din observații și din calcule pot cauza modificări în înțelegerea realității, în perceperea și formularea problemei, în model, în specificația și/sau realizarea programului.

Diferențele între observații și calcule nu sunt întotdeauna datorate funcționării defectuoase a programului sau imperfecțiunii modelului. Acestea sunt dificultăți care în timp pot fi depășite. Dar realitatea nu este fixă și ea se schimbă, implicând totodată, noi modificări ale programelor.

C) Programe E (fig.1.11) - Sunt programe care automatizează o activitate umană sau socială. Ele reprezintă o aplicare a calculatorului în lumea reală, din care cauză noile sisteme mai poartă numele și de *sisteme cu calculator inclus*.

Programele E sunt cele mai predispuse la modificări din cauza unei interacțiuni puternice cu mediul.

Exemplu: Sisteme de operare pentru calculator, gestiune de stocuri, rezervări de locuri, și sisteme de IA, sisteme expert. În toate cazurile comportarea sistemului, cerințele utilizatorilor și suportul cerut depind de experimentările directe la utilizatori.

Pe măsură ce acesta se familiarizează cu un sistem al cărui proiect și atribuții, depind cel puțin parțial, de propria lor atitudine, este posibil o modificare a comportamentului acestora, pentru a reduce sau a mări eficiența. În mod inevitabil aceasta implicare conduce la cerințe de modificare a sistemului.

Aceste modificări par a fi continue și nu ocazionale și sunt intrinseci naturii sistemelor de calcul și a modului în care sunt dezvoltate și folosite programele.

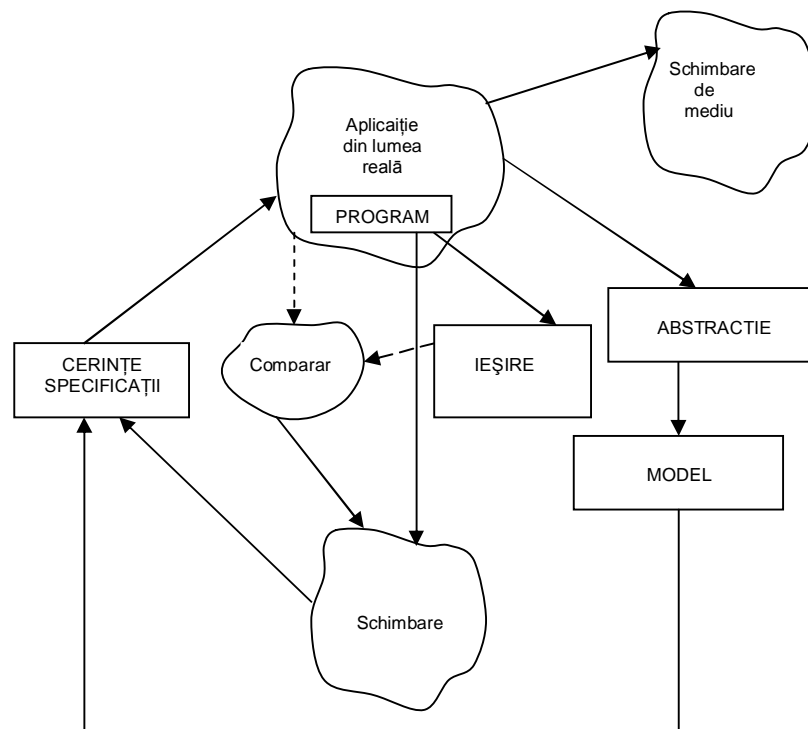


Fig.1.11. Programe S

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

Primul este de a descrie pachetul software și modul lui de utilizare. Această documentație este cunoscută sub numele de **documentație de utilizare** și este destinată utilizatorului sistemului, având un caracter mai puțin tehnic.

În momentul de față, documentația de utilizare este un instrument de marketing foarte important. O documentație bună, împreună cu o interfață bine proiectată și prietenoasă, măresc accesibilitatea produsului și prin aceasta sporesc vânzările. Producătorii care se respectă angajează de obicei personal calificat pentru elaborarea documentației sau furnizează versiunea preliminară a pachetului unor autori independenți, astfel ca la lansarea pe piață a produsului să apară și manualele necesare diferitelor categorii de utilizatori.

În general, documentația de utilizare ia forma unui manual care conține o prezentare a celor mai folosite caracteristici oferite de produsul respectiv (adesea sub forma unui ghid pas cu pas), un capitol cu instrucțiuni de instalare și o secțiune de referință în care toate funcțiile produsului sunt descrise detaliat.

Adesea, manualul este disponibil în formă tipărită, dar sunt și multe cazuri în care el este furnizat sub forma unui fișier stocat pe același mediu ca și produsul software. Această modalitate permite utilizatorului să vadă pe ecran porțiuni ale manualului chiar în timp ce folosește produsul. În acest caz, informațiile sunt împărțite în segmente de mici dimensiuni, numite pachete de asistență (help), care sunt incluse direct în sistemul software, în sensul că se poate cere acces chiar din cadrul programului sau uneori apar automat pe ecran dacă utilizatorul ezită prea mult între comenzi.

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** și are în mod inevitabil un caracter mult mai tehnic decât documentația de utilizare.

Cu ani în urmă, această documentație consta în programe sursă, în forma finală, la care se mai adăugau unele explicații schițate după încheierea procesului de dezvoltare abordare, pe care o vor recunoaște probabil mulți programatori începători. Această manieră este inacceptabilă astăzi, mai ales pentru sistemele de mari dimensiuni.

Astfel documentarea sistemului începe cu dezvoltarea specificațiilor inițiale și 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 și de relații între entități, dicționarul de date și 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, specialiștii încurajează limbajele de nivel înalt, bine proiectate, includerea în programe a

comentariilor și 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 și natura umană. Pe măsură ce dezvoltarea sistemului înaintază, este puțin probabil ca specificațiile, diagramele și schemele inițiale să rămână nemodificate. De cele mai multe ori vor apărea schimbări datorate unor probleme care nu au fost prevăzute de la început (de aceea și tehnicile care utilizează prototipuri iau din ce în ce mai mult amploare). În acest caz, există tendința de a se efectua direct modificările respective, fără a se mai reveni la documentele proiectate anterior. În această situație, documentele nu mai sunt conforme cu realitatea, iar documentația finală va fi incorectă.

Iată deci un alt argument în favoarea instrumentelor CASE, care ușurează mult redesenarea diagramelor și actualizarea dicționarelor de date. În plus, în contextul metodologiilor bazate pe prototipuri, care acceptă ideea că analiza și proiectarea sunt procese iterative în cadrul implementării, modificarea documentației devine regula și nu excepția. În acest mediu, probabilitatea de obținere a unei documentații finale corecte crește considerabil.

În încheiere, vom preciza din nou că exemplul cu actualizarea documentației este doar una dintre problemele întâmpinate de ingineria software, care trebuie să îmbine regulile abstracte ale științei cu înțelegerea naturii umane. Este suficient să spunem că în orice proiect care implică lucrul în echipă apar inevitabil conflicte interpersonale, ambiții și orgolii, ca să înțelegem că ingineria software este un domeniu vast, care include multe aspecte psihologice ce nu sunt legate direct de ceea ce numim informatică. Totuși, indiferent de natura relațiilor care există într-o echipă de proiectare a produselor software, documentația sistemului trebuie să însoțească produsul, să fie completă și în perfectă concordanță cu versiunea oferită utilizatorului. Este o cerință obligatorie din cele ce desemnează așa numitul "produs la cheie". Același lucru este valabil și pentru produsele care au suferit modificări, adăugări sau extensii.

1.10. Dreptul de proprietate și garanții

În închierea acestui capitol este bine să luăm în discuție și câteva aspecte juridice ale ingineriei software. Deși s-ar crede că principiile juridice generale sunt suficiente pentru rezolvarea disputelor apărute în industria software, lucrurile nu stau chiar așa. În particular, este necesară o metodă prin care investiția importantă făcută de o persoană fizică sau juridică pentru a dezvolta un produs software de calitate să fie protejată, iar persoana respectivă să-și poată acoperi cheltuielile și să obțină profit. În lipsa unei protecții a investiției, puțini vor fi cei care s-ar încumeta să producă software-ul de care societatea are nevoie.

Din păcate, problemele referitoare la dreptul de proprietate asupra produselor software nu se încadrează foarte clar în legile existente privind protecția drepturilor de autor și a brevetelor. Deși aceste legi au fost elaborate tocmai pentru a permite fabricantului unui “produs” să-și facă public produsul, păstrându-și totodată drepturile de proprietate asupra lui, particularitățile produselor software au pus adesea instanțele în mare dificultate în eforturile lor de aplica și 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 redată aceste idei. Valoarea unei opere literare rezidă din stil, formă și nu neaparat din subiect.

Prin urmare, munca autorului (artistului) este protejată acordându-i-se acestuia dreptul de proprietate asupra unei formulări a unei idei și nu asupra ideii în sine. O altă persoană este deci liberă să exprime aceeași idee, cu condiția ca în cele două versiuni să nu apară “similitudini substanțiale”.

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, și mai puțin în reprezentarea acestora.

Prin urmare, aplicarea directă a legii drepturilor de autor va lăsa neprotejată tocmai investiția principală a producătorului de software, permițând unei firme concurente să utilizeze liber algoritmul descoperit de acesta, cu condiția ca reprezentarea să nu fie “substanțial similară”.

Într-un fel, putem spune că legile referitoare la drepturile de autor au fost concepute astfel încât să protejeze mai degrabă implementarea decât funcția, însă valoarea unui program constă adesea mai degrabă în funcție decât în formă.

De aceea, legea dreptului de autor este mai potrivită pentru protecția programelor care implementează algoritmi clasici, bine cunoscuți, decât pentru protecția investițiilor care conduc la descoperirea de noi algoritmi.

Dacă un algoritm este deja cunoscut, atunci valoarea programului constă în modul în care este exprimat respectivul algoritm; în schimb, pentru un algoritm nou și creativ, principala valoare a programului este chiar algoritmul, pe care din păcate legea dreptului de autor nu îl protejează.

Este paradoxal, cu cât eforturile de dezvoltare ale unui program sunt mai creative, cu atât legea dreptului de autor îl protejează mai puțin.

În domeniul drepturilor de proprietate asupra pachetelor software, utilizarea brevetelor se lovește 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 algoritmi intră și ei în această categorie, astfel că și în acest caz legea lasă neprotejată cea mai valoroasă componentă a programului – algoritmul. În plus, obținerea unui brevet este un proces costisitor și 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 și cele legate de brevete au rolul de a încuraja atât creșterea numărului de invenții, cât și schimbul liber de idei, în beneficiul societății.

Atunci când drepturile le sunt protejate, creatorii și inventatorii sunt mai tentați să își aducă realizările la cunoștința publicului.

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

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

Cu toate acestea, instanțele iau arareori în considerare aceste limitări de responsabilitate dacă se dovedește că producătorul a dat dovadă de neglijență. În aceste cazuri, problema care se pune este dacă producătorul a asigurat produsului un nivel de calitate compatibil cu natura sa.

Astfel, un nivel acceptabil pentru un procesor de text poate fi considerat neglijență dacă este vorba de un sistem de control al unui reactor nuclear. Prin urmare, cea mai bună metodă împotriva eventualelor reclamații este abordarea cu maximum de seriozitate a procesului de dezvoltare a sistemului produs.

1.11. Exerciții propuse

1. Care este propria dv. cerință atunci când dezvoltați o piesă software? De ce? Trebuie să vă reexaminați solicitarea?
2. Este software-ul scump ? Ce criteriu ați utilizat pentru a ajunge la concluzia dumneavoastră?
3. Este ușor de dezvoltat/programat un sistem software? Justificați-vă răspunsul.
4. Se știe că există diferențe enorme între programatori din punctul de vedere al productivității . De ce credeți că este această situație? Care este cauza diferențelor dintre diverși programatori?
5. Se consideră următoarele cazuri :
 - a. Un microcalculator care controlează o mașină de spălat;
 - b. Un sistem de control al unei stații de transformare a energiei electrice;

- c. Software pentru calcularea și tipărirea unui stat de plată;
- d. Un sistem de operare de interes general ;
- e. Un sistem care controlează și monitorizează un echipament medical;
- f. O rutină matematică de interes general;
- g. Un joc pe calculator.

Pentru fiecare dintre aceste situații analizați importanța diferitelor solicitări identificate în acest capitol. Puneți-le în ordine, pentru fiecare situație.

6. Care credeți că va fi costul hardware-ului și software-ului în fiecare dintre cazurile de mai înainte?
7. Ce credeți despre mentenanță? V-ar place să o faceți?
8. Dați un exemplu de program în care minimizarea timpului de execuție și memoria ocupată sunt contradictorii. Gândiți-vă la un exemplu în care sunt în armonie.
9. Analizați conflictele și consistențele dintre solicitări din ingineria software.
10. În completarea cerințelor descrise în acest capitol există și altele în care ingineria software ar avea succes?

2

Etapele de dezvoltare a sistemelor de programe

2.1. Ciclul de viață

Există patru faze fundamentale ale metodologiilor ingineriei software:

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

Deși aceste faze se referă în mod special la *ciclul de viață* al produsului software, ele pot fi aplicate și altor stadii de existență prin care trece un program de la „naștere” până la „moarte”: lansare, întreținere, ieșire din uz.

2.1.1. Faza de analiză

Această fază definește *cerințele* sistemului, independent de modul în care acestea vor fi îndeplinite. Acum se definește problema pe care clientul dorește să o rezolve. Rezultatul acestei faze este *documentul care conține specificarea cerințelor* și care trebuie să precizeze clar ce trebuie construit.

Documentul încearcă să redea cerințele din perspectiva clientului, definind scopurile și interacțiunile la un nivel descriptiv înalt, independent de detaliile de implementare, cum ar fi, de exemplu: formularea problemei, așteptările clientului sau criteriile pe care trebuie să le îndeplinească produsul.

Granița dintre descrierile de nivel înalt și cele de nivel scăzut nu este foarte bine trasată.

Uneori, dacă un detaliu tehnic important trebuie specificat, el va apărea în document. Totuși, aceasta trebuie să fie excepția și nu regula. Aceste excepții pot fi determinate de necesitatea menținerii compatibilității cu alte sisteme deja existente, sau a unor anumite opțiuni dorite de client, de exemplu utilizarea unui

anumit standard sau o constrângere asupra dimensiunilor imaginii aplicației, care poate fi destinată unei categorii speciale de utilizatori sau care va rula pe niște sisteme cu o serie de particularități (de exemplu, monitoare care nu suportă rezoluții mari, etc.).

Faza de analiză poate fi văzută ca o rafinare a detaliilor. Distincția dintre detaliile de nivel înalt și cele de nivel scăzut sunt puse mai bine în evidență de abordările top-down (unde se merge către detaliile de nivel scăzut) și bottom-up (care tind către detaliile de nivel înalt).

Documentul cerințelor, numit *specificarea cerințelor* poate fi realizat într-o manieră formală, bazată pe logică matematică, sau poate fi exprimat în limbaj natural. În mod tradițional, el descrie *obiectele* din sistem și *acțiunile* care pot fi realizate cu ajutorul obiectelor. Aici noțiunea de „obiect” nu trebuie confundată cu obiectul din programarea orientată obiect. Descrierea obiectelor și acțiunilor trebuie să fie generală și să nu depindă de o anumită tehnologie. Desigur, într-o abordare POO, descrierile vor lua forma obiectelor și metodelor, însă în alte abordări, obiectele pot fi de exemplu servicii care accesează baze de date.

În general, documentul cerințelor descrie *ontologia* proiectului, adică vocabularul de cuvinte cheie (în special construcții substantivale și verbale) care va fi utilizat pentru definirea protocolului specific aplicației. Descrierile acestea nu implică proiectarea arhitecturii aplicației, ci enumerarea părților componente și 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 și structuri de date.

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

- *Obiecte*: Documentul trebuie să definească mai întâi ontologia sistemului, care este bazată în mare parte pe construcții substantivale pentru identificarea pieselor, părților componente, constantelor, numelor și a relațiilor dintre acestea;
- *Acțiuni*: Documentul trebuie să definească de asemenea acțiunile pe care trebuie să le îndeplinească sistemul și care sunt sugerate în general de

- construcții verbale. Exemple de acțiuni sunt: metodele, funcțiile sau procedurile;
- *Stări*: Sunt definite ca mulțimi de setări și valori care disting sistemul între două ipostaze spațio-temporale. Fiecare sistem trece printr-o serie de schimbări de stare. Exemple de stări sunt: starea inițială, cea finală sau stările de eroare. Cele mai multe stări depind de domeniul problemei. Stările sunt asociate cu obiectele sistemului. Un eveniment declanșează o tranziție de stare care poate conduce la îndeplinirea unei acțiuni de către sistem;
 - *Scenarii tipice*: Un scenariu este o secvență de pași urmați pentru îndeplinirea unui scop. Când sistemul este terminat și aplicația este disponibilă, clientul trebuie să poată utiliza, într-o manieră cât mai facilă și clar specificată, toate scenariile tipice ale aplicației. Scenariile tipice trebuie să reprezinte majoritatea scenariilor de utilizare ale aplicației. Ponderea acestora variază de la un sistem la altul, dar 90% se consideră o proporție acceptabilă. Bineînțeles că un sistem cu un singur scenariu de utilizare este relativ simplu de obținut, pe când unul cu mii de scenarii posibile va fi mult mai dificil de analizat. Deseori este invocată regula 80/20: 80% din funcționalitatea sistemului se realizează cu 20% din efortul de muncă. Executarea restului minoritar de funcționalitate necesită marea majoritate a timpului de lucru;
 - *Scenarii atipice*: Un scenariu atipic trebuie să fie îndeplinit de sistem numai în cazuri speciale. Clientul poate să sperie, de exemplu, că o eroare neprevăzută este un eveniment atipic. Totuși, sistemul trebuie să gestioneze un număr cât mai mare de categorii de erori, prin tehnici stabilite, precum tratarea excepțiilor, monitorizarea proceselor etc.;
 - *Cerințe incomplete sau nemonotone*: O enumerare completă a cerințelor, pentru toate situațiile care pot apărea în condiții de lucru reale, nu este posibilă. În logica tradițională, o teorie este definită de o mulțime finită de axiome. Teoremele din teoria respectivă sunt propoziții adevărate. Dacă se adaugă ulterior noi axiome, teoremele existente rămân valide iar noile

teoreme dezvoltate sunt adăugate teoremelor stabilite. În logica nemonotonă, adăugarea de noi axiome poate invalida unele teoreme care au fost demonstrate anterior. O nouă teorie nu mai este o simplă extensie a teoriei vechi, ci o mulțime de teoreme noi, împreună cu o parte din teoremele vechi. Procesul de stabilire a cerințelor are o natură iterativă și nemonotonă. Mulțimea inițială de cerințe (axiomele) definește posibilitățile (teoremele) sistemului. Noile cerințe pot infirma soluțiile vechi. Pe măsură ce un sistem crește în dimensiuni și complexitate, stabilirea cerințelor devine din ce în ce mai dificilă, mai ales când procesul de colectare a cerințelor este distribuit, fiind realizat de indivizi cu specializări diferite.

2.1.2. Faza de proiectare

Pe baza cerințelor din faza de analiză, acum se stabilește *arhitectura* sistemului: componentele sistemului, interfețele și 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 și capturează semnificația detaliilor din documentul cerințelor;
- *Interfețele* ajută la îmbinarea componentelor. O interfață reprezintă granița dintre două componente, utilizată pentru comunicarea dintre acestea. Prin intermediul interfeței, componentele pot interacționa;
- *Comportamentul*, determinat de interfață, reprezintă răspunsul unei componente la stimulii acțiunilor altor componente.

Documentul de proiectare descrie planul de implementare a cerințelor. Se identifică detaliile privind limbajele de programare, mediile de dezvoltare, dimensiunea memoriei, platforma, algoritmi, structurile de date, definițiile globale de tip, interfețele etc.

În această fază trebuie indicate și *prioritățile critice* pentru implementare. Acestea sugerează sarcinile care, dacă nu sunt executate corect, conduc la eșecul sistemului. Totuși, chiar dacă prioritățile critice sunt îndeplinite, acest fapt

nu duce automat la succesul sistemului, însă crește nivelul de încredere că produsul va fi o reușită.

Folosind scenariile tipice și atipice, trebuie realizate compromisurile inerente între performanță și complexitatea implementării. *Analiza performanțelor* presupune studierea modului în care diferitele arhitecturi conduc la diferite caracteristici de performanță pentru fiecare scenariu tipic. În funcție de frecvența de utilizare a scenariilor, fiecare arhitectură va avea avantaje și dezavantaje. Un răspuns rapid la o acțiune a utilizatorului se realizează deseori pe baza unor costuri de resurse suplimentare: indecși, managementul cache-ului, calcule predictive etc. Dacă o acțiune este foarte frecventă, ea trebuie realizată corect și eficient. O acțiune mai rară trebuie de asemenea implementată corect, dar nu este evident care e nivelul de performanță necesar în acest caz. O situație în care o astfel de acțiune trebuie implementată cu performanțe maxime este închiderea de urgență a unui reactor nuclear.

Planul de implementare și planul de test, descrise mai jos, pot fi incluse de asemenea în fazele de implementare și respectiv testare. Însă unul din scopurile fazei de proiectare este stabilirea unui plan pentru terminarea sistemului, de aceea cele două planuri au fost incluse în paragraful curent.

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

Planul de test definește testele necesare pentru stabilirea calității sistemului. Dacă produsul trece toate testele din planul de test, este declarat terminat. Cu cât testele sunt mai amănunțite, cu atât este mai mare încrederea în sistem și deci crește calitatea sa. Un anumit test va verifica doar o porțiune 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.

De exemplu, să considerăm BIOS-ul (Basic Input/Output System) construit de IBM la începutul anilor '80 în strânsă legătură cu sistemul de operare

DOS (Disk Operating System) al firmei Microsoft. Din rațiuni de performanță, BIOS-ul a fost plasat într-un chip ROM, și deci patch-urile pentru eventualele erori erau imposibile. Astfel, au fost necesare teste cu acoperire de 100%. Codul propriu-zis al BIOS-ului era destul de mic, câteva mii de linii. Însă deoarece BIOS-ul are o natură asincronă, testul a presupus mai întâi crearea unui mediu asincron care să aducă sistemul în starea dorită și apoi trebuia generat un eveniment care să declanșeze un test. Foarte repede, setul de test a devenit mult mai mare decât BIOS-ul. A apărut astfel problema testării însuși a mediului de teste! În final, o acoperire de 100% a fost atinsă, dar cu un cost foarte ridicat. O soluție mai ieftină a fost înscricționarea BIOS-ului într-o combinație dintre EPROM (Electronic Programmable Read Only Memory) și ROM. Cea mai mare parte a produsului era plasat în ROM, iar patch-urile erau plasate în EPROM. Aceasta a fost abordarea adoptată de Apple pentru Macintosh.

În general, este suficient ca testele să cuprindă scenariile tipice și atipice, fără să verifice întregul sistem, cu absolut toate firele de execuție. Acesta poate conține ramificații interne, erori sau întreruperi care conduc la fire de execuție netestate. Majoritatea sistemelor sunt pline de bug-uri nedescoperite. De obicei, clientul participă în mod logic la testarea sistemului și 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 știe exact ce trebuie să construiască, chiar dacă rămâne loc pentru inovații și flexibilitate.

De exemplu, o componentă poate fi proiectată mai restrâns, special pentru un anumit sistem, sau mai general, pentru a satisface o direcție de reutilizare.

Echipa trebuie să gestioneze problemele legate de calitate, performanță, biblioteci și 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 și chiar înainte ca procesul de dezvoltare ulterioară a produsului să poată continua;
- *Erori necritice*: Sunt cunoscute, dar prezența lor nu afectează în mod semnificativ calitatea observată a sistemului. De obicei aceste erori sunt listate în notele de lansare și au modalități de ocolire bine cunoscute;
- *Erori necunoscute*: Există întotdeauna o probabilitate mare ca sistemul să conțină 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 învățat însă acest lucru și produc sisteme cu funcționalitate extinsă, dar cu o calitate scăzută. Totuși, e mai simplu să-i explici clientului de ce lipsește o anumită funcție decât să-i explici de ce produsul nu este performant. Un client satisfăcut de calitatea produsului va rămâne loial firmei și va aștepta noile funcții 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-și poate descoperi foarte ușor propriile greșeli. O persoană nouă care privește codul poate descoperi greșeli evidente care scapă celui care citește și recitește materialul de multe ori. Din păcate, această practică poate determina o atitudine indiferentă față de calitate în echipa de implementare.

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

Testele de regresie (engl. „regression test”) sunt colecții de programe care testează una sau mai multe trăsături ale sistemului. Rezultatele testelor sunt

adunate și dacă există erori, eroarea este corectată. Un test de regresie valid generează rezultate verificate, numite „standardul de aur”.

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

Testele sunt colectate, împreună cu rezultatele standardelor de aur, într-un pachet de test de regresie. 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 așteptărilor curente. Pachetul de test este rulat din nou și 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 și dezvoltarea continuă. Acest mecanism detectează situațiile când starea curentă de dezvoltare a produsului invalidează o stare existentă. Astfel, se previne regresia sistemului într-o stare de eroare anterioară.

Există patru puncte de interes în testele de regresie pentru asigurarea calității.

Testarea internă tratează implementarea de nivel scăzut. Fiecare funcție 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 unităților testează o unitate ca un întreg. Aici se testează interacțiunea mai multor funcții, dar numai în cadrul unei singure unități. Testarea este determinată de arhitectură. De multe ori sunt necesare așa-numitele „schele”, adică programe special construite pentru stabilirea mediului de test. Numai când mediul este realizat se poate executa o evaluare corectă. Programul schelă stabilește stări și valori pentru structurile de date și asigură funcții externe fictive. De obicei, programul schelă nu are aceeași calitate ca produsul software testat și 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 interfeței sunt vizibile pentru test.

Testarea internă și a unităților poate fi automatizată cu ajutorul instrumentelor de acoperire (engl. „coverage tools”), care analizează codul sursă și generează un test pentru fiecare alternativă a firelor de execuție. Depinde de programator combinarea acestor teste în cazuri semnificative care să valideze rezultatele fiecărui fir de execuție. De obicei, instrumentul de acoperire este utilizat într-un mod oarecum diferit: el urmărește liniile de cod executate într-un test și apoi raportează procentul din cod executat în cadrul testului. Dacă acoperirea este mare și liniile sursă netestate nu prezintă mare importanță pentru calitatea generală a sistemului, atunci nu mai sunt necesare teste suplimentare.

Testarea aplicației testează aplicația ca întreg și este determinată de scenariile echipei de analiză. Aplicația trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziția clientului. Spre deosebire de testarea internă și a unităților, care se face prin program, testarea aplicației se face de obicei cu scripturi care rulează sistemul cu o serie de parametri și colectează rezultatele. În trecut, aceste scripturi erau create manual. În prezent, există instrumente care automatizează și acest proces. Majoritatea aplicațiilor din zilele noastre au interfețe grafice (GUI).

Testarea interfeței grafice pentru asigurarea calității poate pune anumite probleme. Cele mai multe interfețe, dacă nu chiar toate, au bucle de evenimente, care conțin cozi de mesaje de la mouse, tastatură, ferestre etc. și care asociate cu fiecare eveniment sunt coordonatele ecran. Testarea interfeței presupune deci memorarea tuturor acestor informații și elaborarea unei modalități prin care mesajele să fie trimise din nou aplicației, la un moment ulterior.

Testarea la stres determină calitatea aplicației în mediul său de execuție. Ideea este crearea unui mediu mai solicitant decât cel în care aplicația va rula în mod obișnuit. Aceasta este cea mai dificilă și complexă categorie de teste. Sistemul este supus unor cerințe din ce în ce mai numeroase, până când acesta cade. Apoi produsul este reparat și testul de stres se repetă până când se atinge un nivel de stres mai ridicat decât nivelul așteptat de pe stația clientului. Deseori apar aici conflicte între teste. Fiecare test funcționează corect atunci când este

făcut separat. Când două teste sunt rulate în paralel, unul sau ambele teste pot eșua. Cauza este de obicei managementul incorect al accesului la resurse critice. Mai apar și probleme de memorie, când un test își alocă memorie și apoi nu o mai eliberează. Testul pare să funcționeze corect, însă după ce este rulat de mai multe ori, memoria disponibilă se reduce iar sistemul cade.

Concluzii

Problematika domeniului ingineriei software este diversă și se amplifică pe măsura creșterii continue a complexității sistemelor software.

Toate eforturile au fost și sunt direcționate către satisfacerea cât mai completă a cerințelor utilizatorilor, a creșterii calității sistemelor și a scăderii costurilor de producere.

2.2. Cerințe – Specificații

În mod logic, prima etapă în realizarea unui produs software constă în stabilirea cu precizie a ceea ce utilizatorul dorește de la sistem. Dominanta în această etapă o constituie comunicarea dintre beneficiar (numit utilizator) și inginerul software. Inginerul care se ocupă de stabilirea cerințelor utilizatorului, ceea ce de fapt reprezintă analiza sistemului, se va numi simplu *analist*. Totul începe odată cu ideea utilizatorului de a avea un sistem nou. El colaborează cu analistul pentru definirea cerințelor și specificațiilor de proiectare ale sistemului. Utilizatorul poate să aibă o idee foarte vagă despre ceea ce dorește sau, din contra, el poate să știe foarte exact.

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

2.2.1. Noțiunea de "cerință"

Taskul de analiză și definire a cerințelor nu este nou pentru un sistem și în particular pentru un sistem software. Cerința reprezintă *ceea ce dorește* de fapt utilizatorul, fără a se preciza de fapt cum se realizează acel lucru. Una dintre marile controverse din Computer Science se enunță astfel:

« Este sau nu este necesar să se specifice și *cum* realizează sistemul o anumită solicitare». Aceasta este relația dintre specificare și 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 funcție de varianta aleasă, știe cum să-și orienteze investigațiile și, de asemenea, ce restricții are. Implementarea unui sistem poate reprezenta o parte importantă din specificațiile unui nou sistem. Mai mult, implementarea și specificarea par a fi intercorelate.

Mai sunt și alte considerente și rațiuni la care analistul trebuie să se gândească în timpul analizei. În primul rând, trebuie să verifice dacă o anumită solicitare este tehnic posibilă. În al doilea rând, este vital să ia în considerare implementarea pentru a putea estima costul și data de livrare a sistemului. O altă cerință pentru specificații este dacă acestea pot să constituie un ghid privind modul în care sistemul răspunde dorințelor utilizatorilor.

Câteodată, specificațiile suferă de deficiențe majore, cum ar fi: sunt incomplete, nu există nici o mențiune despre cost sau termen de livrare.

Să luăm acum în discuție specificațiile solicitărilor pentru o piesă simplă de software:

« Să se scrie un program Pascal care să realizeze un repertor de telefon personal. Programul trebuie să implementeze funcțiile de căutare a unui număr dat și de introducere a unui număr nou. De asemenea, el trebuie să realizeze o interfață prietenoasă cu utilizatorul ».

Pentru a decide dacă o specificație este bună, se formulează întrebările:

- Oare spune CE trebuie să facă sistemul și nu CUM ?
- Este clară ?
- Este suficient de detaliată ?
- Este completă ?

În privința primei întrebări se stipulează deja în enunț că, programul va fi scris în limbajul Pascal, care semnifică de fapt CUM se va implementa aplicația. În al doilea rând, specificațiile nu dau detalii asupra naturii celor două funcții - ele

sunt, în totalitate, vagi. În al treilea rând, cerința cu privire la interfața prietenoasă cu utilizatorul este și ea extrem de vagă.

În concluzie, este ușor să scrii specificații pentru piese software, dar este dificil să scrii specificații bune. Se va examina mai departe ghidul pentru scrierea specificațiilor produselor software.

2.2.2. Procesul de alegere a cerințelor

Activitatea de selectare a cerințelor presupune colaborarea și lucrul împreună atât al analistului cât și al utilizatorului. Se pot distinge trei feluri de activități în cadrul acestui proces de specificare a cerințelor, și anume:

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

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

Analiza este etapa în care efectiv inginerul software se gândește 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 existenței unor puncte de vedere diferite ale diverșilor utilizatori.

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

În procesul complicat al comunicării și negocierii aceste trei activități se pot repeta sporadic. De multe ori, pot exista situații de conflict, generate de exemplu de faptul că unii utilizatori influențați de filmele de "science fiction" văzute, au impresia că se poate face orice cu calculatorul, sau doresc un nivel foarte înalt de inteligență artificială.

Rezumând, rolul analistului este:

1. Să achiziționeze și să selecteze cererile de la utilizatori.
2. Să ajute la rezolvarea diferențelor 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 secțiunea următoare).
6. Să negocieze și în final să pună de acord diferitele opinii ale utilizatorilor pentru formularea specificațiilor.

2.2.4. Specificarea cerințelor

Sfârșitul procesului de analiză și selectare a cererilor îl constituie specificarea acestora. Este o "piesă vitală" din documentația sistemului, crucială în dezvoltarea ulterioară a oricărui sistem. Specificarea este documentul de referință prin care este asigurată dezvoltarea sistemului.

Cei trei factori importanți care sunt luați în considerație sunt:

- 1) nivelul de detaliere;
- 2) cui este adresat documentul;
- 3) notațiile utilizate.

Primul factor se referă la nevoia de a restrânge specificațiile, pe cât posibil numai la ceea ce "face sistemul" și nu la *cum* va realiza cerința respectivă.

A doua problemă este importantă deoarece specificațiile trebuie să fie înțelese de două categorii diferite de oameni: utilizatori și proiectanți.

Utilizatorii preferă o descriere netehnică, exprimată în limbaj natural. Din păcate acesta este excelent pentru scrisori și comunicare, dar impropriu pentru specificații precise, consistente și neambiguii.

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

Există câteva notații pentru scrierea specificațiilor:

- *informale* - scrise în limbaj natural, utilizate cât mai clar și cu cât mai multă grijă posibilă;
- *formale* - utilizând notații matematice, cu rigoare și consistență;
- *semiformale* - utilizând o combinație de limbaj natural cu diferite diagrame și notații tabelare.

Cele mai multe dintre aceste notații își au originea în metodele pentru proiectarea software-ului, care sunt de fapt metodele de implementare a software-ului. Aceste notații vor fi discutate mai târziu și includ pseudocodul, diagramele de flux și "diagramele de structură"

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

Varianta modernă este de a elabora două documente și anume:

1. Specificațiile cererilor scrise în primul rând pentru utilizatori, pentru a descrie punctul lor de vedere asupra sistemului, exprimat în limbaj natural. Acestea constituie substanța contractului dintre utilizatori și proiectanți.

2. O specificare tehnică, care este folosită în primul rând de proiectanți, exprimată într-o formă de notații formale și descriind numai o parte a informației în toate specificațiile cererilor.

În măsura în care aceste două documente sunt adoptate trebuie ca ele să fie compatibile. Considerând că specificațiile sunt scrise de obicei în limbaj natural, trebuie să se proiecteze structura generală a specificațiilor și să se identifice părțile componente.

O abordare care asigură o structură clară a specificațiilor este partiționarea. Programele sunt constituite în mod esențial, din combinații de date și acțiuni. În specificații, elementele corespunzătoare sunt solicitările funcționale și de date.

Una dintre disputele majore din "lumea calculatoarelor" este relativă la prioritatea uneia sau alteia dintre cele două elemente esențiale și anume: date sau funcții. În abordările de dezvoltare a sistemelor din ingineria software se tinde mai mult spre a plasa datele după acțiuni. Din contra, în abordările de tratare a informației se plasează mai întâi datele și apoi acțiunile.

Abordările actuale de dezvoltare a sistemelor software, cum ar fi cele orientate pe obiecte, acordă o importanță egală datelor și funcțiilor care tratează aceste date.

Totuși, formatul specificărilor tinde să reflecte metoda de dezvoltare a sistemului care va fi utilizată ulterior.

O lista de verificare a conținutului pentru specificațiile solicitărilor trebuie să arate astfel:

- solicitări funcționale;
- solicitări de date;
- restricții;
- ghid de aplicare.

Solicitări funcționale

Solicitările funcționale sunt esența specificațiilor cererilor. Ele indică sistemului "ceea ce trebuie să facă". Solicitățile funcționale sunt caracterizate de verbe și realizează acțiuni.

Exemple:

- Sistemul va afișa titlurile și toate cărțile scrise de același autor.
- Sistemul va afișa permanent temperaturile tuturor mașinilor.

Solicitări de date

Acestea au două componente:

1. Date de intrare sau ieșire pentru sistem, cum ar fi de exemplu dimensiunile ecranului;
2. Date care sunt memorate în sistem, de obicei în fișiere pe disc. De exemplu, informația despre cărțile dintr-o bibliotecă publică.

Restricțiile

Acestea au de regulă influențe asupra implementării sistemului. Exemple:

“Sistemul trebuie scris în limbajul de programare ADA”, “ Sistemul va răspunde utilizatorului în timp de o secundă”. Principalele restricții 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 informații despre 10000 angajați)
- 9). Nivelul de încărcare pentru terminale pe unitatea de timp
- 10). Fiabilitatea solicitărilor (ex. sistemul trebuie să aibă un anumit timp mediu între defecțiuni pentru o perioadă de șase luni).

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

Ghidul de aplicare

Acesta furnizează informații utile pentru implementare, în situația 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. “

Acum, după ce s-au studiat diferitele clasificări ale specificațiilor, trebuie examinate și deficiențele care pot să apară. Una dintre ele este imprecizia informației. De exemplu: “ Interfața va fi prietenoasă.” În altă situație sunt în contradicție unele cu altele, cum ar fi: “ Rezultatele vor fi memorate pe suport magnetic” , sau, “ Sistemul va răspunde în mai puțin de o secundă. “

Omisiunea sau incompletitudinea pot fi uneori dificil de identificat. O categorie tipică de specificații omise este aceea referitoare la tratarea datelor de intrare eronate, furnizate de utilizator. Câteodată o solicitare este neclară sau susceptibilă de mai multe interpretări și aceasta datorită, bineînțeles, limbajului natural de descriere.

În concluzie, realizarea cu succes a specificațiilor este o activitate necesară și care solicită implicarea cu competență a unui mare număr de persoane.

2.3. Concepte ale specificațiilor de programare

Orice program se exprimă în două forme:

- una fiind o mulțime obișnuită de comenzi și structuri de control caracteristice limbajului de programare utilizat, care exprimă CUM se rezolvă problema;
- a doua este o serie de propoziții (secvențe) într-un formalism matematic, care exprimă CE face programul.

Presupunând că una dintre aceste forme este o reprezentare satisfăcătoare a intențiilor programatorului, dacă se poate demonstra că ele exprimă același lucru, sau au aceeași semnificație, 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 consistența celor două descrieri ale aceleiași probleme.

Atunci când se solicită descrierea semnificației unei piese de software, de regulă se descrie interacțiunea programului cu mediul utilizatorilor. Exprimarea semnificației programului în termenii utilizatorilor prin descrierea execuțiilor sale posibile se numește **descriere operațională**. Această descriere nu este unică, o altă cale fiind descrierea în termenii entităților unui limbaj de programare.

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

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

Se știe că una dintre cele mai importante surse de erori în programe este generată de absența unei porțiuni din proiectul logic, deci un program trebuie exprimat (descriș) în așa fel încât să se reducă această sursă de erori.

2.3.1. Variabilele și stările unui program

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

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

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

Expresia condițiilor care trebuie satisfăcute, în spațiul stărilor, pentru o execuție corectă a programului se numește **precondiție**. Specificațiile stărilor la terminarea programului se numesc **postcondiții**.

Operațiile dintr-un program ne permit să reorganizăm diferitele reprezentări ale aceluiași concept. Un tip de date este o mulțime de valori și operații definite pe ea însăși.

Exemple:

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

>

"adunare" : $G \times G \rightarrow G$

+

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

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

În conformitate cu Dijkstra programatorii pot utiliza trei modalități de proiectare a programelor: enumerarea, inducția matematică și abstractizarea.

Instrumentul mental al enumerării este principala metodă folosită de programatori. Utilizăm gândirea enumerativă ori de câte ori vrem să instruiem pe cineva despre execuția mai multor activități.

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

În general, principiile inducției pot fi definite astfel:

1. **Baza inducției** este de a stabili că un anumit element p al unei mulțimi S satisface o relație R .
2. **Pasul inducției** 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 și pasul inducției sunt demonstrate, atunci orice element derivând din p printr-un număr de aplicații ale lui T , de asemenea satisface R .

Principiul inducției este fundamentarea teoretică a metodei de verificare propusă de Floyd. Importanța acestui concept de bază este aceea că nu cere o execuție mentală a unei secvențe de comandă, permițând proiectarea și verificarea ciclurilor care trebuie executate de mai multe ori, în funcție de valori particulare ale datelor.

Atunci când inducția matematică este aplicată asupra stărilor descrise de relația R , această metodă este un instrument efectiv pentru proiectarea secvențelor de iterații în general și executarea lor de ori câte ori.

În proiectarea programelor, așa cum am amintit deja, se utilizează și limbajul logicii predicatelor. Fiecare predicat definește un subset al spațiului stărilor pentru care el este adevărat.

Precondiția unui program este un predicat care definește un subset al stărilor acceptat ca intrare legitimă. Constanta predicat T, care este adevărată peste tot, definește toate stările posibile ale spațiului stărilor. Constanta predicat F, care este falsă peste tot, nu definește nici o stare în spațiul stărilor.

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

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

Dându-se un program S și o condiție r, vom găsi "cea mai slabă condiție", adică condiția care definește numai și numai acele stări inițiale care asigură terminarea în stările care satisfac r.

Această condiție, "cea mai slabă" este un predicat funcție atât de S cât și de r, indicat prin **$wp(S,r)$** .

Bineînțeles că se poate defini și problema duală. Dându-se o condiție p a unui program S, vom găsi "cea mai puternică" postcondiție, adică predicatul care definește numai și numai acele stări pentru care programul va satisface terminarea când este inițializat într-o stare care satisface p.

Această "cea mai puternică" postcondiție este o funcție de S și p definită ca **$Sp(S,p)$** .

Conceptele "cea mai slabă" condiție și "cea mai puternică" postcondiție 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 și conștient.

Calculul Dijkstra este direct îndreptat spre realizarea unei tehnici care, la sfârșitul 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 condițiile și postcondițiile problemei și 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 condiției și postcondiției. În acest fel întreaga structură a programului poate fi definită și se demonstrează corectitudinea lui prin utilizarea unui text mai scurt.

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

Problemele specifice limbajelor de programare, cum ar fi pointerii și transmisiile de parametri între module, limitează aplicabilitatea calculului Dijkstra. Totuși, exprimarea specificațiilor programelor în termenii limbajului calculului predicativ ajută la reducerea erorilor.

2.4. Specificarea formală

Obiectivul acestui capitol este de a înțelege și scrie specificațiile unui sistem software. S-a amintit deja că ieșirile din activitatea de analiză sunt constituite din specificații de proiectare, care conțin atât solicitări funcționale cât și solicitări de date, cu alte cuvinte se detaliază cu precizie ceea ce sistemul va executa având în vedere restricțiile practice de care proiectantul va ține cont. Ne vom referi la componenta funcțională a specificațiilor de proiectare ca la o *specificație software*.

Comportarea unui sistem este influențată de dezvoltarea lui în cele trei faze, și anume: specificarea, implementarea și verificarea. O abordare convențională este specificarea sistemului în limbaj natural (în limba română),

scrierea programului într-un limbaj de programare (ex. Pascal) și verificarea prin testare. S-a argumentat că o astfel de abordare este susceptibilă la tot felul de erori și în ultimă instanță generează software incorect.

Utilizând abordarea convențională se pune întrebarea: cum și de ce au fost scrise programe incorecte? Există trei cauze esențiale:

1. Specificarea este greșită - ea poate fi ambiguă, vagă, inconsistentă și/sau incompletă.
2. Programul este greșit - nu execută ceea ce există în specificații.
3. Verificarea este incompletă - testarea nu este exhaustivă, mai mult decât atât, nici nu poate fi exhaustivă .

Există următoarele aserțiuni 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 specificației - adică spune CUM poate fi rezolvată problema;
- 3). Verificarea este o *justificare* - spune DE CE realizarea satisface specificația.

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

Avantajele utilizării unei notații formale pot fi rezumate astfel:

- Specificațiile formale pot fi studiate matematic - cu alte cuvinte, o specificație poate fi judecată utilizând tehnicile matematice. De exemplu, diverse forme de inconsistență sau incompletitudine în specificații pot fi detectate automat.
- Specificațiile formale pot fi întreținute cu mai multă ușurință decât dacă ar fi scrise în limbaj natural. Aceasta face ca specificațiile să prezinte mai multă siguranță și să fie asigurat controlul posibilelor consecințe ale schimbărilor.
- Notația utilizată pentru exprimarea specificațiilor formale este extensibilă.

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

Și mai important este că în acest fel, avem posibilitatea potențială de a dovedi corectitudinea programului în raport cu specificațiile sale.

2.5. Exemplu de specificare formală

În mod curent, cele mai cunoscute limbaje de specificare formală sunt Z și VDM. Ambele utilizează o abordare pe bază de model, adică reprezintă tipurile de date și structurile necesare pentru a putea descrie problema folosind entități cum ar fi: mulțimi, funcții și propoziții.

Limbajul Z a fost prima dată introdus de Jean-Raymond Abriel în 1979 și dezvoltat în cadrul Programming Research Group la Universitatea Oxford. Partea "puternică" a lui Z este aceea că specificațiile structurate pot fi achiziționate folosind "schema de calcul" ; aceasta permite specificațiilor 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 inițiată în laboratoarele de cercetare IBM din Viena, în 1970. Această metodă se bazează pe notațiile semantice, o abordare a definirii limbajelor de programare făcută de Scott și Strachey. VDM reprezintă mai mult decât un limbaj semantic, el punând la dizpoziția celor care îl utilizează reguli și proceduri care vor fi urmărite în diferite stadii de dezvoltare a sistemului.

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

Pentru a putea înțelege mai bine exemplul următor, să reluăm câteva notații matematice cum ar fi: mulțimile, secvențele și funcțiile.

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

{Anton, Vasile, Ioana, Alice}

este o mulțime de nume. Se poate utiliza semnul "∈" pentru a indica apartenența unui element la o mulțime.

Astfel Anton ∈ {Anton, Vasile, Ioana, Alice}, dar

Sandu ∉ {Anton, Vasile, Ioana, Alice}.

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

[Anton, Vasile, Ioana, Alice]

este o secvență de nume. *Capătul (capul)* secvenței este Anton, iar *corpul* este [Vasile, Ioana, Alice]. Dacă S este o secvență, atunci elementele lui S marchează mulțimea de elemente din secvența S. Dacă S este o secvență a mulțimii, atunci elementele lui S = {Anton, Vasile, Ioana, Alice}. O secvență poate avea elemente care se repetă, în timp ce o mulțime, nu.

O *funcție* exprimă o relație între două elemente ale unei mulțimi. De exemplu, *director* poate asocia nume cu numere de telefon. Dacă vom utiliza *Nume* pentru a reprezenta mulțimea tuturor numelor posibile și *Număr* pentru a reprezenta mulțimea tuturor numerelor de telefon posibile, atunci putem descrie *director* astfel:

director : Nume → Număr

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

Submulțimea (subsetul) pe care este definită funcția se numește *domeniul de definiție al funcției*. De exemplu, să presupunem că *Nume* este mulțimea {Anton, Vasile, Ioana, Alice} și *Număr* este mulțimea {421563, 123456}. În acest caz *directorul* va fi definit explicit astfel:

director(Anton) = 421563

director(Alice) = 123456

Dacă Vasile și Ioana nu au telefon atunci *director (Vasile)* și *director (Ioana)* nu sunt definite. Domeniul funcției este mulțimea {Anton, Alice}, care este un subset al mulțimii *Nume*.

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

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

Anton \rightarrow 421563 și Alice \rightarrow 123456 se numesc "corespondențe" (maplets). *Director* este o funcție finită deoarece domeniul său este finit, cu alte cuvinte conține un număr finit de corespondențe.

Funcțiile exprimă mai mult decât o relație. Astfel, dacă luăm ca model funcția telefon, mai multe persoane pot avea același număr de telefon, dar o anumită persoană nu poate avea decât un singur număr de telefon.

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

O bancă are numai un ghișeu unde în mod normal clienții așteaptă la coadă. Fiecare client se identifică prin numărul de cont și sunt permise numai tranzacții de adăugare sau scoatere din cont. Clientului *i* se refuză eliberarea sumei cerute dacă contul său este vid. Dacă el dorește mai mult decât are în cont atunci va primi numai suma pe care o mai are în cont. După ce s-a efectuat tranzacția, clientul părăsește 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 funcție parțială care face corespondența între numerele de cont și balanța băncii:

$$\text{bancă} : \text{Cont} \rightarrow \text{Balanță}$$

Cont este mulțimea de numere de conturi posibile, iar *Balanță* este mulțimea tuturor balanțelor (disponibilului) băncii. Deci o funcție *bancă* poate arăta astfel :

$$\{a_1 \rightarrow 23, a_2 \rightarrow 87, a_3 \rightarrow 45\},$$

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

Coadă la ghișeul băncii poate fi modelată ca o secvență de numere de cont astfel:

coadă : *secvCont*

În acest fel $[a_2, a_5, a_1]$ reprezintă o coadă tipică. Vom adopta convenția 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 restricții suplimentare ori de câte ori se schimbă sau actualizează coada.

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

1. Intrările operației;
2. Condiția în care operația poate fi aplicată (*precondiția*);
3. Expresia care arată relația dintre starea sistemului înainte de operație și starea sistemului după operație (*post-condiția*).

Vom adopta convenția de a folosi numele obișnuit pentru a desemna starea inițială și *&nume*, pentru starea finală. Exemplu : *coadă* reprezintă starea cozii înainte de sosirea unui client, iar *&coadă*, după.

Să luăm acum specificația care introduce un client în coadă:

sosire (*client* : *Cont*)

precondiție

$$client \in dom_bancă$$

$$client \notin elem_coadă$$

post-condiție

$$\&coadă = adaug_coadă [client]$$

Numele operației este *sosire*. Ea are o intrare numită *client*, de tip *Cont*. Precondiția indică faptul că atunci când un client sosește la coadă el trebuie să aibă cont în bancă, altfel nu va fi acceptat în coada de așteptare. Post-condiția indică faptul că după operație coada se mărește cu o persoană. (Observație: *adaug* este un operator de concatenare a două secvențe).

De exemplu, să presupunem:

$$bancă = \{a_1 \rightarrow 23, a_2 \rightarrow 87, a_3 \rightarrow 45\}$$

$coadă = [a2]$

$client = [a1]$

atunci

$client \in \{a1, a2, a3\}$

$client \notin \{a2\}$

și

$\&coadă = [a2, a1]$

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

$credit (suma : Balanță)$

precondiția

$coadă \neq []$

post-condiția

$\&bancă = bancă \oplus \{cap\ coadă \rightarrow bancă(cap\ coadă) + suma\}$

$\&coadă = elimin\ coadă$

Operația *credit* are o intrare numită *sumă*, de tip *Balanță* și o condiție care precizează că lista (coada) nu trebuie să fie vidă, adică trebuie să fie măcar o persoană la coadă.

\oplus este un operator de scriere adițională, peste ceea ce era înainte. El crează o nouă funcție din alte două date ca operanzi. De exemplu, să presupunem că:

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

$coadă = [a2, a1]$

$suma = 10$

Atunci

$capul\ cozii = a2$

$bancă(capul\ cozii) = 87$

astfel încât

$\&bancă = \{a1 \rightarrow 23, a2 \rightarrow 87, a3 \rightarrow 45\} \oplus \{a2 \rightarrow 87+10\}$

$= \{a1 \rightarrow 23, a2 \rightarrow 97, a3 \rightarrow 45\}$

Corespondența care are a_2 în stânga a fost scrisă peste vechea corespondență, definind astfel o nouă funcție care modelează noua stare a băncii.

Ultima componentă a post-condiției arată că după efectuarea tranzacției, clientul din față părăsește coada :

$$\begin{aligned} \&coadă &= \text{elimin } [a_2 \ a_1] \\ &= [a_1] \end{aligned}$$

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

debit (*suma* : *Balanță*)

precondiție

$$coadă \neq []$$

$$bancă (\text{capul cozii}) \geq \text{suma}$$

post-condiția

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

$$\&coadă = \text{elimin } coadă$$

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

$$bancă = \{ a_1 \rightarrow 23, a_2 \rightarrow 87, a_3 \rightarrow 45 \}$$

$$coadă = [a_2, a_1]$$

$$\text{suma} = 10$$

$$\text{Capul cozii este } a_2 \text{ iar } \text{bancă} (\text{capul cozii}) = 87$$

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

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

Un *invariant* este o aserțiune despre componentele modelului matematic, pe care se vor baza apoi specificațiile noastre. Dacă invariantul se află înainte de

operațiile pe care le-am specificat, atunci el trebuie să apară și după ce operația este completă.

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

$$(\forall a \in \text{dom_bancă}) \text{bancă}(a) \geq 0$$

$$\text{elem_coadă} \subseteq \text{dom_bancă}$$

$$\# \text{coadă} \leq 10$$

Acest invariant stipulează că :

1. Oricare ar fi a din domeniul băncii, valoarea lui $\text{bancă}(a)$ este mai mare sau egală cu zero. Aceasta înseamnă că toți clienții băncii au credit.
2. Mulțimea de persoane aflată în coadă este o submulțime 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 și creditul conservă acest invariant, dar *sosire* nu, de aceea nu a existat nici o restricție cu privire la lungimea cozii. Totuși *sosire* va conserva invariantul dacă vom adăuga un extra predicat :

$$\# \text{coadă} < 10$$

în precondiția sa.

Invariantii furnizează un instrument de valoare pentru asigurarea consistenței peste operațiile care se fac în cadrul specificațiilor.

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 înțelegerea 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 specificații.

- Să existe o dovadă că programul este corect prin prisma specificațiilor sale.

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

2.6. Exerciții

1) Care sunt informațiile necesare care trebuie colectate și înregistrate ca specificații software?

2) Explicați dificultățile limbajului natural pentru descrierea specificațiilor software?

3) Luați ca exemplu un sistem informatic mic (care doriți dv.). Identificați componentele funcționale și de date din sistem. Identificați problemele legate de specificațiile software cum ar fi: ambiguitate, inconsistență și informații vagi.

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

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

s2 - pentru fiecare carte informațiile standard sunt:

- titlul
- autorul
- codul de clasificare (cota cărții)
- anul apariției
- dacă este împrumutată
- data solicitării

s3 - calculatorul trebuie să memoreze informațiile pentru 1000 de cărți

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

(a) solicitarea de împrumut a unei cărți

(b) înapoierea unei cărți împrumutate

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

s5 - comenzile vor fi accesibile printr-o selecție 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 cărților, cu un cap de tabel corespunzător. Acest lucru trebuie să se facă în paralel cu altă comandă primită;

s8 - măsuri de securitate: sistemul va inițializa informațiile din bibliotecă numai dacă el conține zero cărți;

s9 - când o carte este ștearsă sistemul va afișa informațiile despre ea;

s10 - sistemul trebuie livrat la data D, trebuie să nu depășească costul C, să fie în întregime documentat și ușor de întreținut.

5. " Scrieți un program de sortare ". Este această modalitate, adecvată ca specificație ?

6. Un aspect necesar al specificațiilor este discuția cu potențialii utilizatori. Cei mai mulți dintre ei nu înțeleg notațiile matematice. Este acesta un dezavantaj? Ce aduc în plus notațiile formale pentru a face această muncă mai productivă?

7. Enumerați câteva cerințe non-funcționale din implementarea programului de împrumut la bancă.

8. Scrieți specificațiile pentru deschiderea și închiderea unui cont bancar.

9. Modificați 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 specificații, pentru programator sau pentru cel care implementează programul ?

3

Paradigmele de dezvoltare a sistemelor de programe

3.1. Etapele dezvoltării programelor

Când pornim la dezvoltarea unui program avem nevoie de:

- înțelegere clară a ceea ce se cere;
- un set de metode și instrumente de lucru;
- un plan de acțiune.

Planul de acțiune se numește *metodologie de dezvoltare*. Dezvoltarea unui anumit program constă într-un set de pași ce se fac pentru a-l realiza. Luând în considerare tipul pașilor 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 acțiune este numit model: el poate fi privit ca un șablon al dezvoltării de programe. În timpul dezvoltării programelor s-a constatat că există anumite tipuri de activități care trebuie făcute la un moment dat:

- *Analiza cerințelor*: Se stabilește ce anume vrea clientul ca programul să facă. Scopul este înregistrarea cerințelor într-o manieră cât mai clară și mai fidelă. Claritatea se referă la lipsa ambiguității iar fidelitatea la înregistrarea cât mai exactă (posibil cuvânt cu cuvânt);
- *Proiectarea arhitecturală*: Din motive de complexitate, programele mari nu pot fi concepute și implementate ca o singură bucată. Programul va trebui construit așadar din module sau componente. Proiectarea arhitecturală împarte sistemul într-un număr de module mai mici și mai simple, care pot fi abordate individual;

- *Proiectarea detaliată*: Se realizează proiectarea fiecărui modul al aplicației, 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 și testate individual, după care sunt integrate în sistemul final. Având în vedere că funcționarea corectă a componentelor individuale a fost testată, integrarea ar trebui să fie o formalitate. Din păcate, componentele nu pot fi testate exhaustiv, iar când acestea lucrează împreună pot să apară situații pe care o anumită componentă nu le-a întâlnit în procesul de testare sau conflicte între anumite componente (de exemplu, conflicte de partajare a resurselor). S-a constatat că atunci când se aplică acest model, timpul de testare explodează, proiectul devenind greu de controlat; aceasta justifică denumirea de „big-bang”. *Modelul incremental* propune crearea unui nucleu al aplicației și integrarea a câte o componentă la un moment dat, urmată imediat de testarea sistemului obținut. Astfel, se poate determina mai ușor 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 îndeplinește cerințele utilizatorului. Întrebarea la care răspundem este: construim produsul corect? Un exemplu de acceptare este testul de acceptare, în care produsul este prezentat clientului. Clientul spune dacă este mulțumit cu produsul sau dacă mai trebuie efectuate modificări;
- *Verificarea*: În procesul de verificare ne asigurăm că programul este stabil și că funcționează corect din punctul de vedere al dezvoltatorilor. Întrebarea la care răspundem este: construim corect produsul?
- *Întreținerea*: După ce programul este livrat clientului, mai devreme sau mai târziu sunt descoperite defecte sau erori ce trebuie reparate. De asemenea,

pot apărea schimbări în specificațiile utilizatorilor, care vor diverse îmbunătățiri. Întreținerea constă în gestionarea acestor probleme.

Se poate constata ușor că aceste activități sunt în strânsă legătură cu cele patru faze ale ingineriei programării: analiza, proiectarea, implementarea și testarea.

3.2. Paradigmele de dezvoltare software

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

- **3.2.1. Metodologii generice**
 - Metodologia secvențială
 - Metodologia ciclică
 - Metodologia hibridă ecluză
- **3.2.2. Metodologii concrete.**
 - 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

În acest paragraf, vor fi prezentate trei categorii importante de metodologii: secvențială, ciclică și hibridă. În metodologia secvențială (cascadă), cele patru faze urmează una alteia într-o modalitate serială. În metodologia ciclică (spirală),

fazele sunt dispuse în cicluri care își generează incremental contribuțiile la sistemul final. Metodologia hibridă (ecluză) combină progresul constant al metodologiei secvențiale cu incrementele iterative ale metodologiei ciclice.

3.2.1.1. Metodologia secvențială

În metodologia secvențială (fig.3.1), cunoscută și 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 tranziție de fază poate fi necesară o decizie managerială.

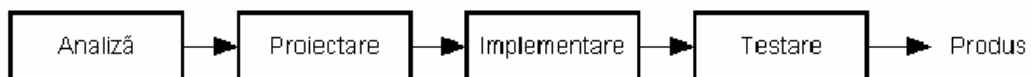


Fig. 3.1. Metodologia secvențială

Avantaje

Metodologia secvențială este potrivită când complexitatea sistemului este mică iar cerințele sunt statice. Ea spune că mai întâi trebuie să ne gândim ce trebuie construit, apoi să stabilim un plan de lucru și apoi să realizăm proiectul, ținând cont de standardele de calitate. De asemenea, se aliază metodelor de inginerie hardware. Forțează menținerea unei discipline de lucru care evită presiunea scrierii codului înainte de a cunoaște precis ce produs va trebui de fapt construit.

De multe ori, echipa de implementare se află în situația de a programa înainte de finalizarea analizei, ceea ce conduce inevitabil la descoperirea unor părți de cod inutile sau care contribuie foarte puțin (poate chiar și ineficient) la funcționalitatea produsului final. Totuși, acest cod devine un balast foarte costisitor: dificil de abandonat și greu de schimbat. Această metodologie forțează analiza și planificarea înaintea implementării, o practică foarte nimerită în multe situații.

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

Dezavantaje

Unul din principalele dezavantaje ale metodologiei secvențiale este faptul că acordă o foarte mare importanță fazei de analiză. Membrii echipei de analiză ar trebui să fie probabil clarvăzători ca să poată defini *toate* detaliile aplicației încă de la început. Greșelile nu sunt permise, deoarece nu există un proces de corectare a erorilor după lansarea cerințelor finale. Nu există nici feedback de la echipa de implementare în ceea ce privește complexitatea codului corespunzător unei anumite cerințe. O cerință simplu de formulat poate crește considerabil complexitatea implementării. În unele cazuri, este posibil să fie chiar imposibil de implementat cu tehnologia actuală. Dacă echipa de analiză ar ști că o cerință nu poate fi implementată, ei ar putea-o schimba cu o cerință diferită care să satisfacă cele mai multe dintre necesități și care să fie mai ușor 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 și trimise următoarei echipe cu foarte puțin feedback. Echipa de analiză nu poate avea toate informațiile privitoare la calitate, performanță și motivare.

Într-o industrie în continuă mișcare, metodologia secvențială 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 își schimbă cerințele după terminarea fazei de analiză? Acest lucru se întâmplă însă frecvent; după ce clientul vede prototipul produsului, el își poate schimba unele cerințe.

3.2.1.2. Metodologia ciclică

Metodologia ciclică (fig.3.2), cunoscută și sub numele de metodologia „spirală”, încearcă să rezolve unele din problemele metodologiei secvențiale. Și această metodologie are patru faze, însă în fiecare fază se consumă un timp mai scurt, după care urmează mai multe iterații prin toate fazele.

Ideea este de fapt următoarea: gândește un pic, planifică un pic, implementează un pic, testează un pic și apoi ia-o de la capăt. În mod ideal, fiecărei faze trebuie să i se acorde atenție și importanță egale.

Documentele de la fiecare fază își schimbă treptat structura și conținutul, la fiecare ciclu sau iterație. Pe măsură ce procesul înaintază, sunt generate din ce în ce mai multe detalii. În final, după câteva cicluri, sistemul este complet și gata de lansare. Procesul poate însă continua pentru lansarea mai multor versiuni ale produsului.

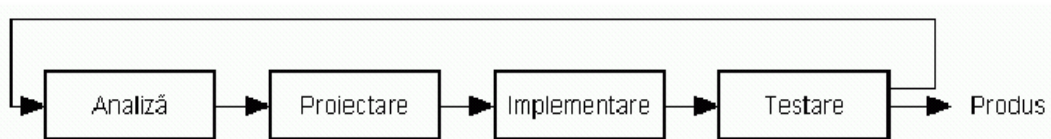


Fig.3.2. Metodologia ciclică

Avantaje

Metodologia ciclică se bazează pe ideea perfecționării incrementale ale metodologiei secvențiale. Permite feedback-ul de la fiecare echipă în ceea ce privește complexitatea cerințelor.

Există etape în care pot fi corectate eventualele greșeli privind cerințele. Clientul poate arunca o privire asupra rezultatului și poate oferi informații importante mai ales în faza dinaintea lansării produsului. Echipa de implementare poate trimite echipei de analiză informații privind performanțele și viabilitatea sistemului. Acesta se poate adapta mai bine progresului tehnologic: pe măsură ce apar noi soluții, ele pot fi încorporate în arhitectura produsului.

Dezavantaje

Metodologia ciclică nu are nici o modalitate de supraveghere care să controleze oscilațiile de la un ciclu la altul. În această situație, fiecare ciclu

produce un efort mai mare de muncă pentru ciclul următor, ceea ce încarcă orarul planificat și poate duce la eliminarea unor funcții sau la o calitate scăzută. Lungimea sau numărul de cicluri poate crește 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 responsabilității. Echipa de implementare poate primi sarcini la care ulterior se va renunța.

Echipa de proiectare nu are o viziune globală asupra produsului și deci nu poate realiza o arhitectură completă. Nu există termene limită precise. Ciclurile continuă fără o condiție clară de terminare. Echipa de implementare poate fi pusă în situația nedorită în care arhitectura și cerințele sistemului sunt în permanență schimbare.

3.2.1.3. Metodologia hibridă ecluză

Metodologia ecluză (engl. „watersluice”), propusă de Ronald LeRoi Burbach (1998), separă aspectele cele mai importante ale procesului de dezvoltare a unui produs software de detaliile mai puțin semnificative și 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.

La început, într-un proces iterativ, fazele de analiză, proiectare, implementare și testare sunt împărțite în mai multe sarcini potențiale, fiecăruia atribuindu-i-se o prioritate care reflectă beneficiul îndeplinirii sarcinii respective pentru scopul final. La fiecare moment se execută sarcina cu prioritate maximă. În funcție 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 și stabilirea priorităților sunt mai bune, cu atât mai eficientă este metodologia.

Pe măsură ce sarcinile stabilite sunt îndeplinite, noi sarcini pot fi descoperite. Acestea sunt adăugate sarcinilor rămase nesoluționate și se reatribuie prioritățile. Procesul continuă până când produsul este gata de lansare.

Prioritățile se stabilesc pe baza unei *funcții de prioritate*, care depinde atât de domeniul problemei și cât și de normele firmei. Ea trebuie să realizeze un compromis între cantitate și calitate, între funcționalitate și constrângerile privind resursele, între așteptări și realitate. Toate funcțiile de prioritate ar trebuie să aibă ca prim scop lansarea produsului.

Pe lângă rolul evident de a stabili prioritățile și deci ordinea de execuție a sarcinilor de lucru, funcția mai trebuie să gestioneze sarcinile conflictuale și nemonotone. Ea trebuie să împartă aceste sarcini în grupuri consistente, să reglementeze selecția grupurilor consistente și apoi să dirijeze selecția sarcinilor din cadrul grupurilor. Pe măsură ce sistemul crește, funcția de prioritate trebuie să aleagă sarcini consistente cu partea deja constituită din sistem. O sarcină nemonotonă vine în contradicție cu sistemul realizat deja și trebuie eliminată dacă nu este absolut necesară pentru succesul sistemului.

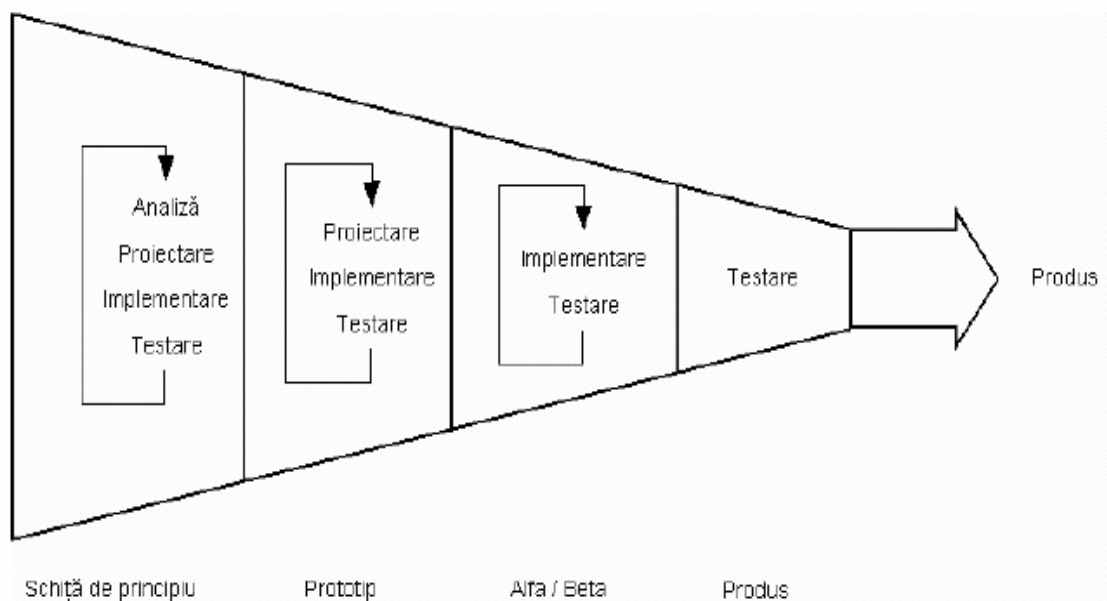


Fig. 3.3. Metodologia hibridă ecluză

Odată ce o componentă este terminată și acceptată de echipă, schimbările asupra sa sunt înghețate. 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 puține la număr, bine justificate și documentate.

Etapele principale ale metodei sunt: schița de principiu, prototipul, versiunile alfa/beta și produsul final. În prima etapă, *schița de principiu*, echipele lucrează simultan la toate fazele problemei. Echipa de analiză sugerează cerințele. Echipa de proiectare le discută și trimite sarcinile critice de implementare echipei de implementare. Echipa de testare pregătește și dezvoltă mediul de test în funcție de cerințe. Echipa de implementare se concentrează asupra sarcinilor critice, care în general sunt cele mai dificile. Această abordare contrastează cu practica curentă de realizare mai întâi a sarcinilor simple. Dacă nu sunt respectate cu strictețe etapele sistemele pot să eșueze. Odată ce componentele critice au fost realizate, sistemul este gata de a face tranziția către stadiul de prototip.

Unul din scopurile aceste etape este de a se convinge echipele că o soluție poate fi găsită și pusă în practică.

În cea de a doua etapă, de *prototip*, cerințele și documentul cerințelor sunt înghețate. Schimbările în cerințe sunt încă permise, însă ar trebuie să fie foarte rare și numai dacă sunt absolut necesare, deoarece modificările cerințelor în acest stadiu al proiectului sunt foarte costisitoare. Este posibilă totuși ajustarea arhitecturii, pe baza noilor opțiuni 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 aplicației. Unul din scopurile acestei etape este de a convinge persoanele din afara echipelor că o soluție este posibilă.

Acum produsul este gata pentru lansarea versiunilor *alfa și beta*. Arhitectura este înghețată, iar accentul cade pe implementare și asigurarea calității. Prima versiune lansată se numește în general *alfa*. Produsul este încă imatur; numai sarcinile critice au fost implementate la calitate ridicată. Numai un număr mic de clienți sunt în general dispuși să accepte o versiune alfa și să-și asume riscurile asociate. O a doua lansare reprezintă versiunea *beta*. Rolul său este de a

convinge clienții că aplicația va fi un produs adevărat și de aceea se adresează unui număr mai mare de clienți.

Când o parte suficient de mare din sistem a fost construită, poate fi lansat în sfârșit *produsul*. În această etapă, implementarea este înghețată și accentul cade pe asigurarea calității. Scopul este realizarea unui produs competitiv. În produsul final nu se acceptă erori critice.

Avantaje

Metodologia ecluză recunoaște faptul că oamenii fac greșeli și că nici o decizie nu trebuie să fie absolută. Echipele nu sunt blocate într-o serie de cerințe sau într-o arhitectură imobilă care se pot dovedi mai târziu inadecvate sau chiar greșite. Totuși, pentru respectarea termenelor limită, metodologia impune date de înghețare a unor faze. Există timp suficient pentru corectarea greșelilor 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 renunțe 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 responsabilități privind delimitarea etapelor și înghețarea succesivă a fazelor de dezvoltare. Ea presupune crearea unui mediu de lucru în care acceptarea responsabilității pentru o decizie care se dovedește mai târziu greșită să nu se repercuteze în mod negativ asupra individului. Se dorește de asemenea schimbarea atitudinii echipelor față de testare, care are loc încă de la început, și față 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 originală a lui Boehm, există o întoarcere, un pas înapoi interactiv între fiecare două etape. Astfel, metoda cascadă este de fapt o combinație de metodologie secvențială cu elemente ciclice. Totuși, în practica inginerescă, termenul „cascadă” este utilizat ca un nume generic pentru orice metodologie secvențială.

Acesta este modelul după care de obicei sistemele sunt dezvoltate în practică. De asemenea, reordonarea fazelor s-a dovedit a fi sub-optimală. Există o mare atracție pentru acest model datorită experienței, tradiției în aplicarea sa și succesului pe care l-a implicat. O sarcină complexă este împărțită în mai mulți pași mici, ce sunt mai ușor de administrat. Fiecare pas are ca rezultat un produs bine definit (documente de specificație, 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.

Dezavantajul principal al modelului în cascadă apare deoarece clientul obține o viziune practică asupra produsului doar în momentul terminării procesului de dezvoltare. De asemenea, modelul nu are suficientă putere descriptivă, în sensul că nu integrează activități ca managementul resurselor sau managementul configurației. Aceasta face dificilă coordonarea proiectului.

După cum am menționat la prezentarea metodologiei generice secvențiale, și modelul cascadă impune înghețarea specificațiilor foarte devreme în procesul de dezvoltare pentru a evita iterațiile frecvente (reîntoarcerile în fazele anterioare atunci când în faza curentă s-au detectat erori: în timpul analizei se descoperă erori de specificații, în timpul implementării se descoperă erori de specificații/proiectare etc., astfel încât procesul poate implica multiple secvențe de iterații ale activităților de dezvoltare).

Înghețarea prematură a cerințelor conduce la obținerea unui produs prost structurat și care nu execută ceea ce dorește utilizatorul. Conduce de asemenea la obținerea unei documentații neadecvate deoarece schimbările intervenite în iterațiile frecvente nu sunt actualizate în toate documentele produse.

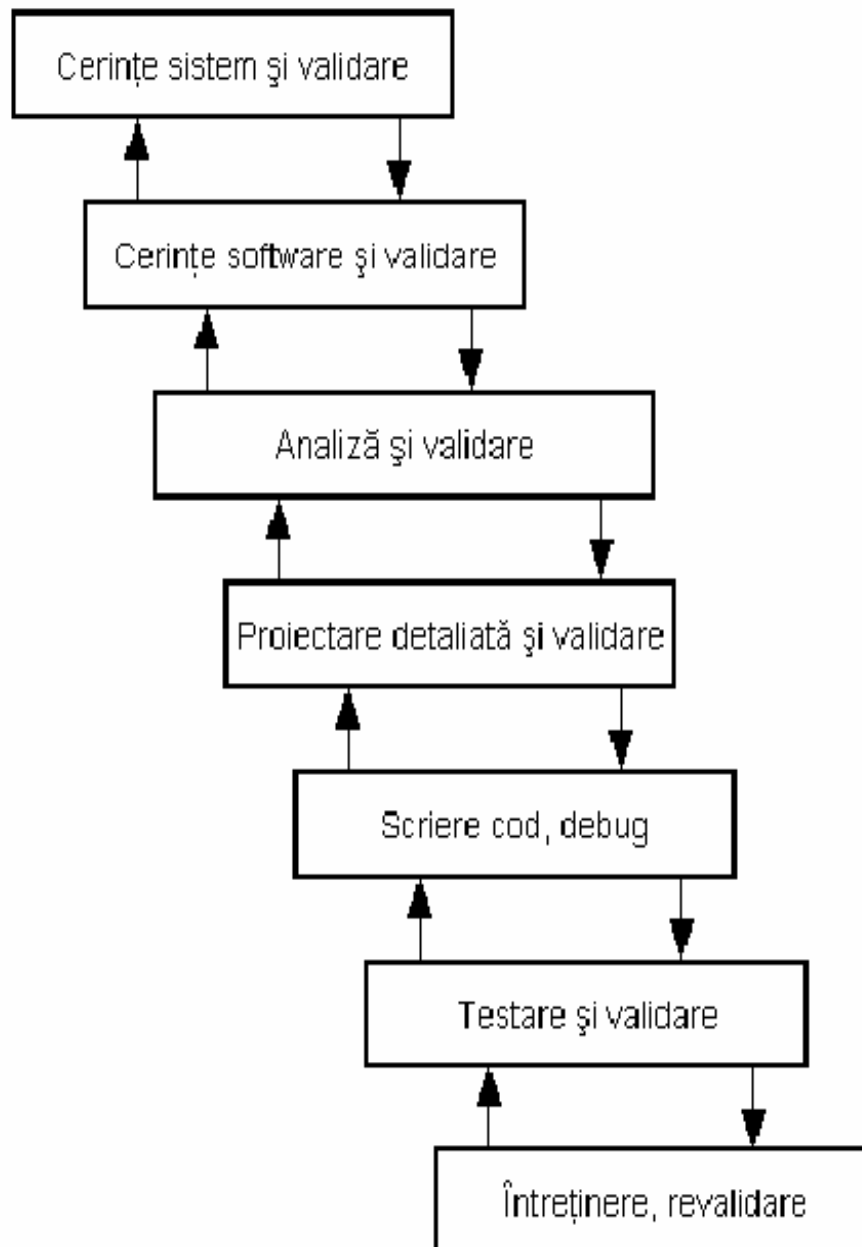


Fig. 3.4. Metodologia cascadă

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 definește următorii pași în dezvoltarea unui produs:

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

Modelul în spirală (fig. 3.5.) recunoaște că problema principală a dezvoltării programelor este riscul. Riscul nu mai este eliminat prin aserțiuni de genul: „în urma proiectării am obținut un model corect al sistemului”, ca în modelul cascadă. Aici riscul este acceptat, evaluat și se iau măsuri pentru contracararea efectelor sale negative.

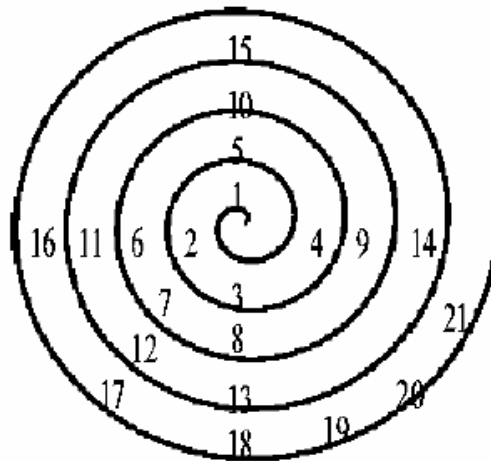


Fig. 3.5. Modelul spirală

Exemple de riscuri:

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

- ⇒ clientul schimbă cerințele;
 - ⇒ o firmă concurentă lansează un program rival pe piață;
 - ⇒ un dezvoltator/arhitect părăsește 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 conține o serie de activități comune:

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

În modelul spirală se consideră că fiecare pas din dezvoltare conține o serie de activități comune:

- pregătirea: se identifică obiectivele, alternativele, constrângerile;
- gestionarea riscului: analiza și rezolvarea situațiilor de risc;
- activități de dezvoltare specifice pasului curent (de exemplu analiza specificațiilor 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 soluția finală.

Deși este considerată ca un exemplu generic pentru metodologia ciclică, metoda are și elemente secvențiale, puse în evidență de evoluția constantă de la o etapă la alta.

Tabelul 3.1. Ciclurile metodologiei în spirală

<i>Ciclul 1 – Analiza preliminară:</i>
<ol style="list-style-type: none"> 1. Obiective, alternative, constrângeri 2. Analiza riscului și prototipul 3. Conceperea operațiilor 4. Cerințele și planul ciclului de viață 5. Obiective, alternative, constrângeri 6. Analiza riscului și prototipul
<i>Ciclul 2 – Analiza finală:</i>
<ol style="list-style-type: none"> 7. Simulare, modele, benchmark-uri 8. Cerințe software și acceptare 9. Plan de dezvoltare 10. Obiective, alternative, constrângeri 11. Analiza riscului și prototipul
<i>Ciclul 3 – Proiectarea:</i>
<ol style="list-style-type: none"> 12. Simulare, modele, benchmark-uri 13. Proiectarea produsului software, acceptare și verificare 14. Integrare și plan de test 15. Obiective, alternative, constrângeri 16. Analiza riscului și prototipul operațional
<i>Ciclul 4 – Implementarea și testarea:</i>
<ol style="list-style-type: none"> 17. Simulare, modele, benchmark-uri 18. Proiectare detaliată 19. Cod 20. Integrarea unităților și testarea acceptării 21. Lansarea produsului

3.2.2.4. Metodologia spirală WinWin

Această metodologie extinde spirala Boehm prin adăugarea unui pas de stabilire a priorității la începutul fiecărui ciclu din spirală și 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 și alternativele.

Punctele ancoră stabilesc trei scopuri intermediare. Primul punct ancoră, numit *obiectivul ciclului de viață*, precizează cazurile sigure de funcționare pentru întregul sistem, arătând că există cel puțin 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 viață și prototipul sistemului. Această primă ancoră spune de ce, ce, când, cine, unde, cum și estimează costul produsului. După executarea acestor operații, este disponibilă analiza de nivel înalt a sistemului.

Al doilea punct ancoră definește *arhitectura ciclului de viață*, iar al treilea – *capacitatea operațională inițială*, incluzând mediul software necesar, hardware-ul, documentația pentru client și instruirea acestuia.

Aceste puncte ancoră corespund etapelor majore din ciclul de viață al unui produs: dezvoltarea inițială, lansarea, funcționarea, întreținerea și ieșirea din funcțiune.

3.2.2.5. Prototipizarea

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

Această versiune nu reprezintă întregul sistem, însă este o parte a sistemului care cel puțin funcționează. Prototipul ajută clientul în a-și defini mai bine cerințele și prioritățile. Prin intermediul unui prototip, el poate înțelege ce este posibil și ce nu din punct de vedere tehnologic. Prototipul este de obicei produs cât mai repede; pe cale de consecință, stilul de programare este de obicei (cel puțin) neglijent. Însă scopul principal al prototipului este de a ajuta în fazele de analiză și proiectare și nu folosirea unui stil elegant.

Se disting două feluri de prototipuri:

- jetabil sau de aruncat (throw-away);
- evoluționar sau lent și recuperabil.

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

În cazul realizării unui prototip evoluționar, scopul este de a crea un schelet al aplicației care să poată implementa în primă fază o parte a cerințelor sistemului. Pe măsură ce aplicația este dezvoltată, noi caracteristici sunt adăugate scheletului existent. În contrast cu prototipul de aruncat, aici se investește un efort considerabil într-un design modular și extensibil, precum și în adoptarea unui stil elegant de programare.

Această metodă are următoarele avantaje:

- permite dezvoltatorilor să elimine lipsa de claritate a specificațiilor;
- oferă utilizatorilor șansa de a schimba specificațiile într-un mod ce nu afectează drastic durata de dezvoltare;
- întreținerea 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 clienți;
- clientul nu înțelege de ce produsul necesită timp suplimentar pentru dezvoltare, având în vedere că prototipul a fost realizat atât de repede;
- deoarece au în fiecare moment șansa de a face acest lucru, clienții schimbă foarte des specificațiile;
- poate fi nepopulară printre dezvoltatori, deoarece implică, în cazul prototipului jetabil, renunțarea 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 știe cu exactitate ce a dorit într-adevăr utilizatorul. Dezvoltarea evolutivă este similară abordării exploratorii, dar scopul acestei dezvoltări este de a stabili cerințele utilizatorului, mai ales atunci când este dificil de făcut acest lucru, sau când documentele de specificații existente sunt ambigue sau incomplete. Această tehnică este în esență o metodă de analiză a specificațiilor.

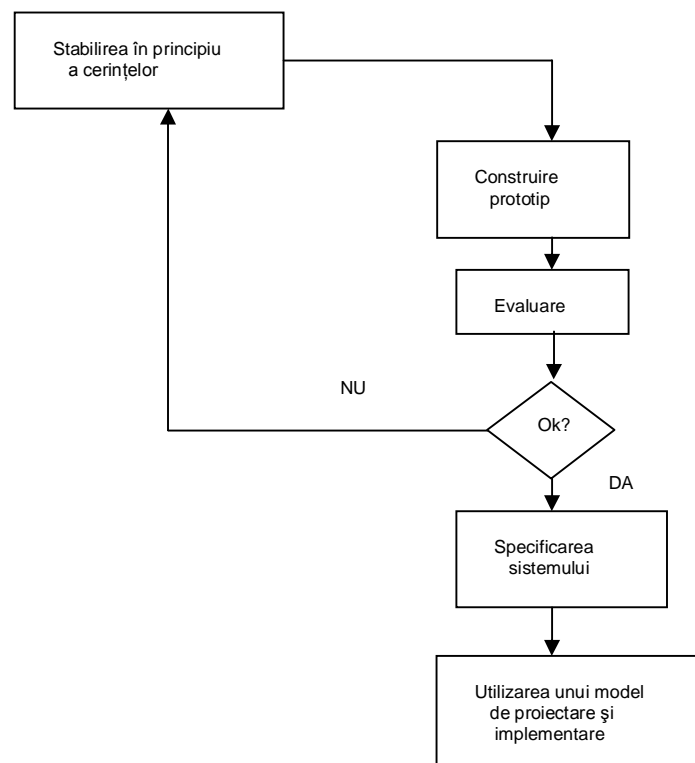


Fig.3.6. Modelul prototipului rapid

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

Când se construiește un prototip, proiectarea și implementarea se realizează în faze diferite de timp.

Cele două abordări pot fi combinate în mod util așa cum se arată în fig. 3.7. În acest caz, stadiul de analiză a cerințelor 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 esențial, un proces complet liniar. Datorită completitudinii specificațiilor utilizatorului, buclele de reacție (feedback) din stadiile precedente par a fi, din ce în ce mai puțin, necesare.

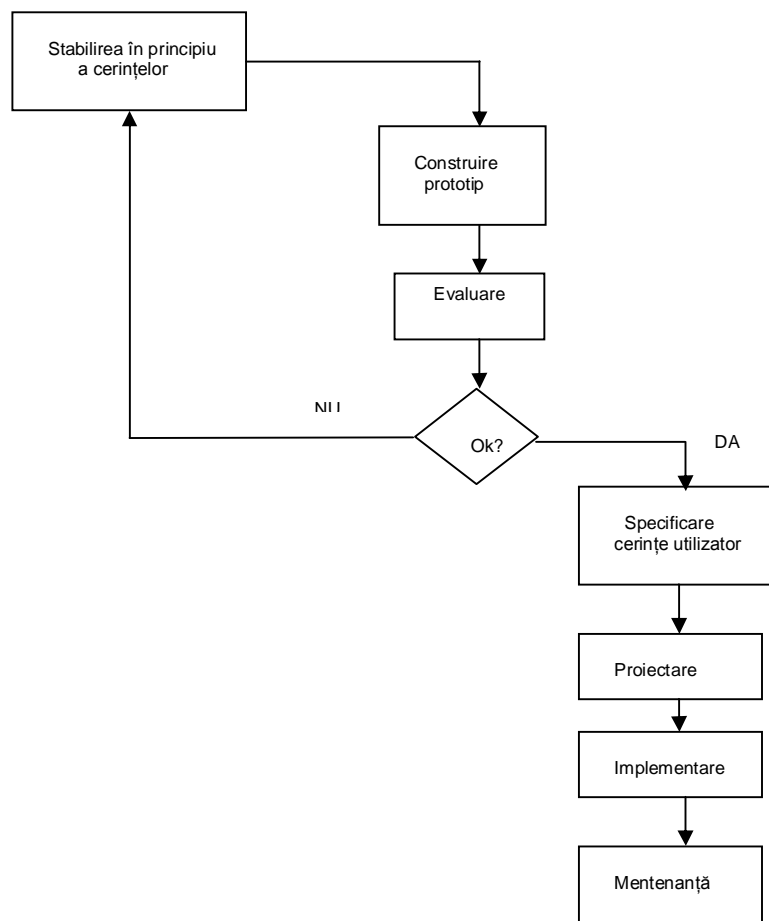


Fig.3.7. Modelul integrat "cascadă și prototipizare"

3.2.2.6. Metodologia Booch

Această metodologie asigură o dezvoltare orientată obiect în fazele de analiză și proiectare. Faza de analiză este împărțită în mai mulți pași. Primul pas este stabilirea cerințelor din perspectiva clientului, generând o descriere de nivel înalt a funcționării și structurii sistemului. Al doilea pas este analiza domeniului, realizată prin definirea claselor: atributele, metodele, moștenirea. Faza de analiză este terminată cu un pas de acceptare. Această fază iterează cei trei pași până când soluția este consistentă.

Și faza de proiectare este iterativă. Design-ul logic este transformat în design fizic, cu detalii privind firele de execuție, procesele, performanțele, tipurile de date, structurile de date etc. Se creează un prototip și apoi se testează. Faza iterează design-ul logic, cel fizic, prototipurile și testarea.

Metodologia Booch este secvențială în sensul că mai întâi este terminată analiza și apoi proiectarea. Ea este ciclică datorită iterațiilor din fiecare fază. Metoda se concentrează în special asupra acestor două faze, de analiză și proiectare, fără să insiste foarte mult asupra implementării și testării.

3.2.2.7. Metode formale

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

Avantaje:

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

Dezavantaje:

- specificarea formală este de obicei o barieră de comunicare între client și analist;

- necesită personal foarte calificat (deci mai scump);
- folosirea impecabilă a tehnicilor specificării formale nu implică neapărat obținerea de programe sigure, deoarece anumite aspecte critice pot fi omise din specificațiile inițiale.

3.2.2.7. Modelul V

În limba germană se numește “V-Modell” și a fost dezvoltat pentru reglementarea dezvoltării de software în administrația federală germană. Acest model evidențiază testarea pe tot parcursul ciclului de dezvoltare (Fig. 3.7.).

Trecerea la faza următoare a dezvoltării software-ului se face numai după ce toate produsele din faza curentă au trecut testele de verificare și validare

Procesul de verificare și validare are scopul detectării cât mai multor erori în ciclul de dezvoltare

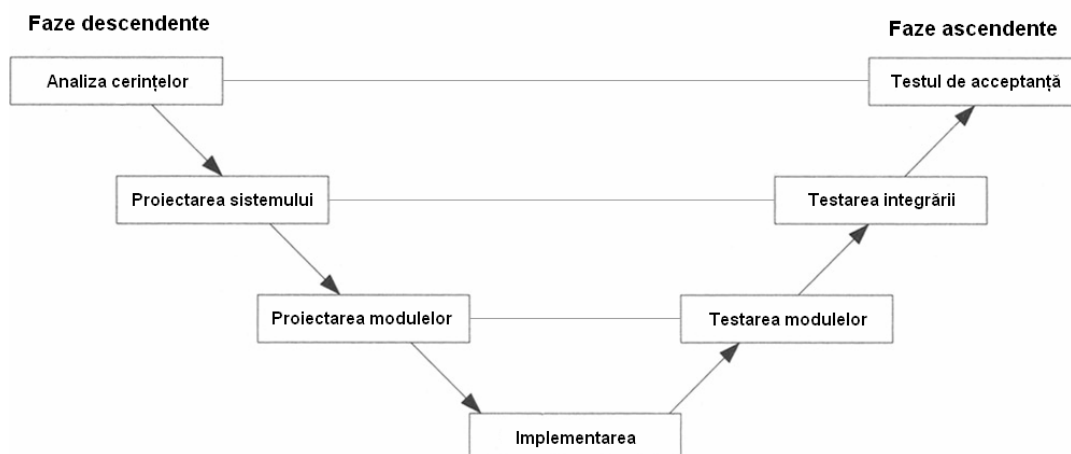


Fig. 3.7. Modelul V

3.2.2.8. 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 aplicații

Fiind o tehnologie nouă (și extremă) are atât adepți cât și critici. XP consideră că dezvoltarea programelor nu înseamnă ierarhii, responsabilități și termene limită, așa cum se află acestea pe masa administratorului, ci înseamnă colaborarea oamenilor din care este formată echipa. Aceștia sunt încurajați să își afirme personalitatea, să ofere și să primească cunoștințe și să devină programatori străluciți.

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 ședințelor și a rapoartelor de activitate. XP ne amintește cu respect ca fișierele PowerPoint nu se pot compila.

De altfel, inspirarea proceselor de dezvoltare a programelor din ingineria construcțiilor 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 și abia apoi trece la execuție, toate acestea într-un mod secvențial și previzibil. Dar dezvoltarea de programe nu seamănă cu așa ceva, oricât am vrea să credem asta. Dacă inginerului constructor respectiv i s-ar schimba cerințele de rezistență și i s-ar muta malurile chiar când a terminat de construit jumătate de pod, putem fi siguri că acel inginer și-ar schimba modul de lucru. Din păcate însă, nu știm (încă) cum. Inițiatorii XP definesc următoarele două cartele, 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 cerințe și de a evalua iterațiile sistemului
- Livrarea incrementală
 - Programul este dezvoltat incremental, clientul indicând cerințele care trebuie incluse la fiecare iterație
- Oamenii nu procesul

- Abilitățile echipei de dezvoltare trebuie recunoscute și exploatate. Echipa trebuie lăsată să-și contureze propriile modalități de lucru, fără a i se da rețete
- Acceptarea schimbării
 - Echipa trebuie să se aștepte ca cerințele să se schimbe iar proiectarea sistemului trebuie făcută astfel încât să se adapteze ușor la aceste schimbări
- Menținerea simplității
 - Concentrare pe simplitate atât în programele dezvoltate cât și în procesul de dezvoltare. Oricând este posibil, trebuie eliminată complexitatea din sistem

Problemele metodelor agile

- Este dificilă menținerea interesului clienților implicați în proces
- Membrii echipei pot fi incapabili să se adapteze la implicarea intensă caracteristică metodelor agile
- Când există mai mulți factori de decizie, este dificilă prioritizarea schimbărilor
- Menținerea simplității necesită lucru suplimentar
- Contractele pot fi o problemă în cazul dezvoltării iterative

Carta drepturilor dezvoltatorului:

- Ai dreptul să știi ceea ce se cere, prin cerințe clare, cu declarații clare de prioritate;
- Ai dreptul să spui cât îți va lua să implementezi fiecare cerință, și să îți revizuești estimările în funcție de experiență;
- Ai dreptul să îți accepți responsabilitățile, în loc ca acestea să-ți fie asignate;
- Ai dreptul să faci treabă de calitate în orice moment;
- Ai dreptul la liniște, distracție și la muncă productivă și plăcută.

Carta drepturilor clientului:

- Ai dreptul la un plan general, să știi ce poate fi făcut, când, și la ce preț;

- Ai dreptul să vezi progresul într-un sistem care rulează și care se dovedește că funcționează trecând teste repetabile pe care le specificești tu;
- Ai dreptul să te răzgândești, să înlocuiești funcționalități și să schimbi prioritățile;
- Ai dreptul să fii informat de schimbările în estimări, suficient de devreme pentru a putea reduce cerințele astfel ca munca să se termine la data prestabilită. Poți chiar să te oprești la un moment dat și să rămâi cu un sistem folositor care să reflecte investiția până la acea dată.

Aceste afirmații, deși par de la sine înțelese, conțin semnificații 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 maximum din cunoștințele sale;
- Scrierea de cod este activitatea cea mai importantă;
- Proiectul este în mintea tuturor programatorilor din echipă, nu în documentații, modele sau rapoarte;
- La orice moment, un reprezentant al clientului este disponibil pentru clarificarea cerințelor;
- 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 sfârșitul unei iterații (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 puțin metodologia orientată obiect care a devenit în momentul

de față, suverană și despre care vom vorbi în continuare. Mai întâi au fost descrise metodologiile generice: secvențială, ciclică și hibridă, cu avantajele și dezavantajele fiecăreia. Apoi s-au amintit câteva metode concrete de dezvoltate: modelul cascadă, modelul spirală, WinWin, prototipizarea, metodologia Booch, metodele formale și așa-numita „programare extremă”.

3.2.2.9. Metodologia Open Source

Metodologia Open Source înseamnă “sursă la vedere”. Este o abordare recentă, apărută ca urmare a dezvoltării mijloacelor de comunicație: FTP, e-mail, grupuri de discuție. 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 licențe open-source (gen GNU+ sistem de operare Open Source gen Unix).

Critici

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

O iterație tipică pentru o astfel de abordare este:

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

3.2.2.10. Reverse Engineering - Inginerie inversă

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

3.2.2.11. Metodologia de dezvoltare Offshore

Offshore Înseamnă în limba engleză , “în larg”. Companiile consideră ca profitabil outsourcing-ul pentru funcțiile care pot fi dezvoltate mai ieftin în altă țară

Venituri din outsourcing IT în 2004 au fost:

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

Motivarea externalizării (outsourcing) este:

- Reducerea costurilor;
- Creșterea eficienței;
- 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.

Etapetele metodologiei sunt:

1. Inițierea proiectului (local)
2. Analiza cerințelor (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 funcționale, așa cum au fost ele concepute de Yourdan, în sensul că focalizează abordarea pe identificarea obiectelor din domeniul aplicației, “potrivind” apoi procedurile în jurul acestor obiecte. La o eventuală modificare a cerințelor, nu va mai fi necesar să fie schimbată toată structura obiectelor.

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

Ce este un obiect?

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

Un obiect trebuie să aibă caracteristicile următoare:

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

- **Clasificare** : obiectele cu aceleași atribute și operații se grupează în clase. Fiecare obiect cu aceleași atribute și operații poate fi considerat ca o instanță a unei clase. Exemple: fereastră, triunghi, listă.

- **Polimorfism** : aceeași operație (cu același nume) poate să aibă comportament diferit în clase diferite. Exemple: *a muta* o fereastră, *a muta* un triunghi. Operația *a muta* înseamnă lucruri diferite, depinzând de obiectul asupra căruia se aplică. Implementarea concretă a unei operații într-o clasă se numește metodă.

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

Modele de lucru în OMT

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

OMT-ul folosește trei modele de bază și anume: modelul obiectelor, modelul dinamic, modelul funcțional.

Modelul obiectelor:

- descrie structura statică a obiectelor din sistem și relațiile 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* și ale cărui arce sunt *relațiile* dintre obiecte.

Modelul dinamic:

- descrie acele aspecte ale sistemului care se schimbă în timp;
- specifică și implementează partea de control a sistemului;
- descrie când se modifică sistemul;
- este reprezentat cu ajutorul diagramelor de stare.

Modelul funcțional:

- 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 și 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 aplicației, 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 cerințelor, analistul va concepe un model cu obiecte din domeniul aplicației și 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 și o strategie de alocare a resurselor. De asemenea, va trebui să stabilească împărțirea sistemului mare în subsisteme.

- Proiectarea obiectelor

Proiectantul obiectelor va adăuga detalii de implementare modelului obținut la analiză, în concordanță cu strategia aleasă în etapa de proiectare a sistemului. Accentul se va pune acum pe algoritmi și structurile de date folosite pentru a implementa clasele obținute în etapa de analiză.

- Implementarea

În cele din urmă, clasele și relațiile dintre clase obținute 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 deși analiza unui sistem poate să se efectueze folosind noțiuni de obiecte și 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 interfață pentru X Window, numită Motif, care este

scrisă în C, deși a fost proiectată folosindu-se noțiuni de analiză orientată pe obiect.

Conceptele de bază

A. Abstractizarea

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

B. Încapsularea (ascunderea informației)

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 și operații, în scopul perfecționării unui algoritm sau eliminării unor erori. Încapsularea ne va împiedica să modificăm toate caracteristicile obiectului, iar aplicațiile care utilizează obiectul nu vor avea de suferit.

C. Împachetarea datelor și a comportamentului în același obiect

Aceasta se referă tocmai la faptul că un obiect conține atât structuri de date cât și operații, și permite să se știe întotdeauna cărei clase îi aparține o anumită metodă, chiar dacă există mai multe operații cu aceeași nume, în clase diferite.

D. Partajarea

Tehnicile orientate pe obiect promovează partajarea la diverse niveluri. Partajarea poate însemna transmiterea acelorași structuri de date și respectiv, operații de-a lungul unor clase, dintr-o ierarhie de clase. Acest lucru are un efect benefic asupra economisirii spațiului. 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 funcționale (sau procedurale) accentul în analiza sistemului cade pe descompunerea funcțională a acestuia, deci pe ceea ce trebuie să facă

sistemul (în ultimă instanță pe construirea diagramelor de flux), în tehnicile OOP accentul cade pe înțelegerea sistemului din punct de vedere al descompunerii acestuia în entități (obiecte) și în stabilirea relațiilor 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 și care sunt relațiile 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 și are ca scop construirea unui model precis, concis și 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, așa cum este ea formulată de potențialii utilizatori. Analistul va construi un model, care poate fi incomplet, pe baza unor cerințe incomplete. De aceea, modelul trebuie apoi rafinat.

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

- scopul problemei;
- necesitățile;
- contextul aplicației;
- diverse presupuneri;
- performanțele necesare;

În partea de proiectare și implementare se vor urmări:

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

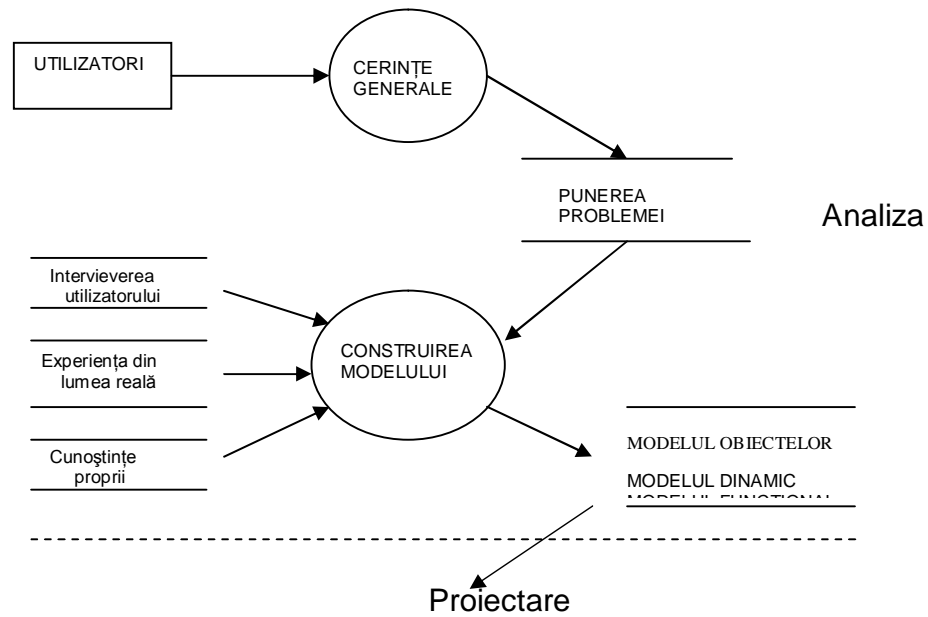


Fig.3.8. Procesul de analiza – vedere generală

Concluzii

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

4

UML- Limbaj unificat de modelare

4.1. Introducere în UML

Limbajul unificat de modelare, numit prescurtat UML se dorește a fi un instrument care să ofere o metodologie unificată de modelare, analiză și proiectare a sistemelor informatice. El oferă o arhitectură de sistem pentru analiza și proiectarea orientată obiect a sistemelor software, incluzând și documentarea acestora, dar și pentru modelarea altor sisteme non-software, cum ar fi modelarea afacerilor, de exemplu.

Metodele orientate pe obiect au evoluat începând cu anii 1960. Multe dintre limbajele orientate pe obiect au câștigat popularitate în anii 80' și proiectanții de sistem au dorit și o metodologie de proiectare orientată pe obiect. Comunitatea programatorilor era însă încă puternic dependentă de metodele de proiectare structurate, rezultând astfel, din combinarea celor două tipuri de metode de programare, modele inconsistente.

Totul a pornit de la Grupul de Management pe Obiect (OMG- Object Management Group), un consorțiu de peste 800 de membri și companii americane (fondat în 1989) care produc și distribuie aplicații realizate în tehnologie orientată obiect. Scopurile principale sunt reutilizarea, portabilitatea și interoperabilitatea software-lui orientat pe obiect.

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

UML este succesorul cel mai bun al metodelor de analiză și proiectare orientate obiect (OOA&A) care au apărut între 1980-1990 cum sunt OODA (Booch), OMT (Rumbaugh) și OOSE (Jacobson). Prima sa versiune a apărut în Ianuarie 1997 și a fost dezvoltat de cei trei autori amintiți mai înainte, adică: Grady Booch, Jim Rumbaugh și Ivar Jacobson.

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

CE ESTE UML ?

Sunt multe definiții și nici una unanim acceptată sau standardizată. Dar toate au în comun următoarele:

UML este un limbaj grafic pentru vizualizarea, specificarea, construirea și documentarea componentelor unui sistem software.

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

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

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

1. **Meta-metamodelul** – infrastructura pentru o arhitectură de modele;
2. **Metamodelul** – o instanță a unui meta-metamodel și definește semantica necesară pentru reprezentarea modelelor aplicației;
3. **Modelul** – o instanță a unui metamodel;
4. **Obiecte utilizator** – o instanță a unui model, utilizate pentru descrierea unui domeniu specific de informație.

Metamodelul UML este un model logic și are în componența 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 detaliile de implementare. Implementările care utilizează metamodelul logic trebuie să se conformeze semanticilor sale și să fie capabil să importe și să exporte obiecte.

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

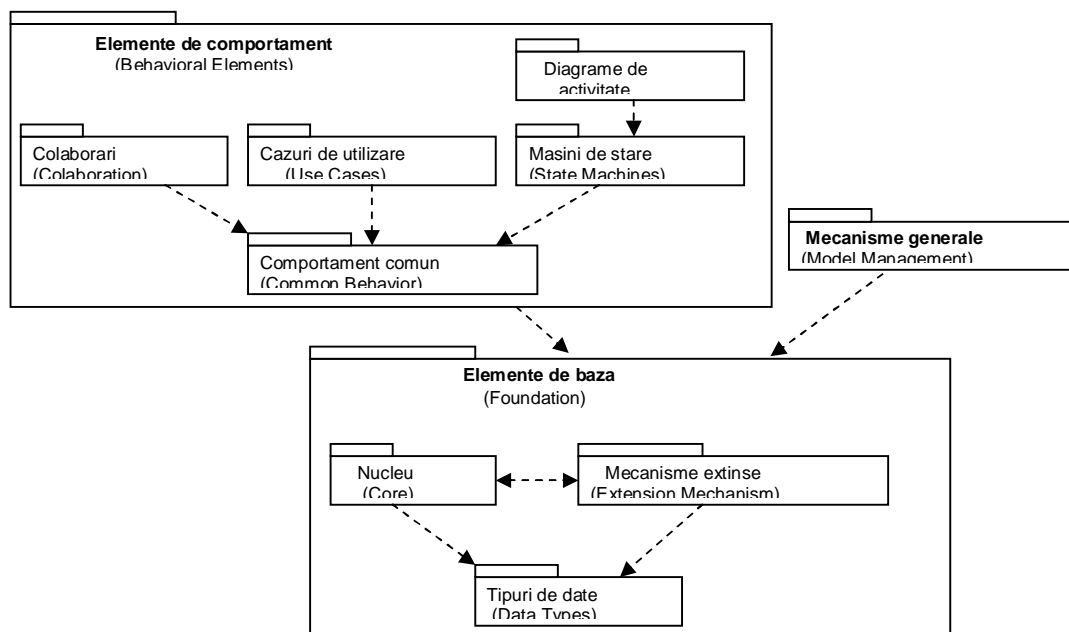


Fig. 4.1. Structura UML

Componentele de bază ale UML sunt elementele model. Un **model** reprezintă o colecție de *obiecte* (clase, pachete, actori, cazuri de utilizare, componente și noduri), *relații* (de asociere, dependență, generalizări) și *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 și clasele lor, interfețele, atributele și relațiile și *modele de comportament (dinamice)*, care accentuează comportamentul obiectelor din sistem, incluzând și metodele, interacțiunea, colaborările și starea lor istorică.

Principalele obiective ale UML-ului sunt:

- Să pună la dispoziția utilizatorului un limbaj vizual pentru dezvoltarea și 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 înțelegerea limbajului de modelare.

4.2. Diagrame și concepte UML

UML definește 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 interacțiuni (Interaction Diagrams);
 - 3.3.4. Diagrama secvențială (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 aplicație (de dezvoltare) (Deployment Diagram).

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

4.2.1. Diagrama de clase (Class Diagram)

O diagrama a claselor este un graf ale cărui noduri sunt clasele din sistem și ale cărui arce sunt relațiile dintre clase. Diagrama claselor este probabil cea mai importantă diagramă UML. O astfel de diagramă poate conține interfețe, pachete, legături, și chiar instanțe, cum ar fi obiecte.

Clasele sunt tipuri abstracte de date. În cadrul programului se lucrează cu instanțieri ale claselor definite, numite *obiecte*.

Clasele de obiecte sunt categorii de obiecte cu **atribute** (structuri de date) și **operații** (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 existență , nu prin atribute);
- **comportament** (felul în care un obiect acționează și reacționează în termenii comunicării prin mesaje și schimbării ale stării).

Scopul diagramei de clase este de a prezenta structura de clase din cadrul unui model. În aplicațiile orientate pe obiect clasele au atribute, operații și relații cu alte clase.

Elementul fundamental al diagramei claselor este un dreptunghi care reprezintă o clasă, așa cum este ilustrată în figura 3.2.

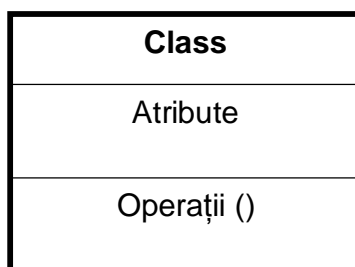










Fig. 4.2. Reprezentarea unei clase din diagrama de clase UML

Dreptunghiul care reprezintă clasa este împărțit în trei compartimente cu următoarele semnificații:

- Compartimentul de sus este cel mai important și conține numele clasei;
- Compartimentul din mijloc conține o listă de atribute;
- Compartimentul de jos conține o listă de operații.

În multe diagrame compartimentele de mijloc și de jos sunt omise. Chiar și atunci când sunt prezente, ele nu arată toate atributele și operațiile. Scopul este de a arata doar acele atribute și operații care sunt esențiale pentru diagrama respectivă.

Diagrama claselor poate conține următoarele elemente model:

-  **Pachete.** Pachetele sunt utilizate pentru a structura modelul.
-  **Dependențe între pachete.** Acestea arată că respectivele clase dintr-un pachet utilizează clasele pachetului de care depind.
-  **Colaborări între obiecte.**
-  **Interfețe.** Interfețele nu conțin atribute, ci doar operații.
-  **Clase.** Clasele reprezintă cel mai important concept al programării orientată pe obiect și al UML.
-  **Relații de moștenire.** Acestea se pot regăsi între interfețe sau între clase, dar niciodată între o interfață și o clasă.
-  **Relații de implementare.** Pot exista numai între interfețe și clase.
-  **Relații de asociere.** Asocierile sunt relații între clase.

În construirea claselor care vor sta la baza proiectării programului trebuie clarificate entitățile care vor evolua în sistem și operațiile asociate. Trebuie de asemenea definite reponsabilitățile fiecărei clase în implementare și colaborările între clasele modelate grafic prin legături de diverse tipuri și cu diferite multiplicități. Legăturile sunt simbolul relațiilor între clase.

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

Circle

```

ItsRadius:double
ItsCenter:Point

```

```

area():double
circumference():double
setCenter(Point)
setRadius(double)

```

Fig. 4.3. Reprezentarea clasei *Circle*

Fiecare instanță de tip *Circle* pare că are o instanță de tip *Point*. Aceasta este o relație cunoscută ca *relație de compoziție* și figurată în UML ca în fig. 4.4.

**Fig. 4.4.** Relație de compoziție

Rombul negru reprezintă compoziția. El este plasat în dreptul cercului deoarece cercul este compus din puncte. Punctul nu trebuie să știe nimic despre cerc. Săgeata din partea cealaltă denotă faptul că relația poate fi parcursă numai într-un sens. În UML se presupune că relațiile sunt implicit bidirecționale cu excepția când se termină printr-o săgeată așa cum am prezentat mai sus. Dacă s-ar omite săgeata ar însemna că Punctul trebuie să cunoască Cercul, ceea ce la nivel de cod ar însemna să de include `# include "Circle.h"` în `Point.h`.

Relațiile de compoziție sunt mai puternice decât cele de conținut sau agregare. Agregarea este întregul sau o parte din relație. În cazul prezentat *Circle* este întregul iar *Point* parte a *Circle*.

Relația de compoziție indică de asemenea că timpul de viață al *Point* depinde de *Circle*. Aceasta înseamnă că, dacă Cercul este distrus va fi distrus și 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 relația de compoziție ca o variabilă membră. Se poate folosi la fel de bine un pointer care la sfârșit v-a fi șters de destructorul Cercului.

Clasificatori

Un *clasificator* este un element care descrie caracteristicile de comportament și structurale ale sistemului. Clasificatorii acceptați de UML sunt de mai multe tipuri și includ: clase, tipuri de date, interfețe, componente, semnale, noduri, cazuri de utilizare și subsisteme. Un clasificator declară un set de caracteristici care includ atribute, metode și operații. Numele unui clasificator este unic și reprezintă o metaclassă abstractă.

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

- <<metaclassă>> - precizează că instanțele clasificatorului sunt clase;
- <<powertype>> - specifică un clasificator ale cărui obiecte sunt descendenții unui anumit părinte;
- <<process>> – un clasificator care reprezintă un flux de control cu o interfață puternică;
- <<thread>> – un clasificator care reprezintă un flux de control;
- <<utility>> – specifică un clasificator care nu are instanțe;

Un *atribut* descrie un domeniu de valori pe care le pot lua instanțele unui clasificator. El poate avea o valoare inițială și una dintre următoarele proprietăți

care se referă la posibilitățile de modificare a valorii, după ce obiectul a fost creat:

- *changeable* (modificabil)- nu se impune nici o restricție asupra valorii atributului;
- *addOnly* – valabil numai pentru atributele cu ordin de multiplicitate mai mare ca unu; o valoare odată adăugată nu mai poate fi ștearsă sau modificată;
- *frozen* – valoarea atributului nu poate fi modificată după inițializarea obiectului.

O **operație** este un serviciu care poate fi solicitat de către un obiect. Are aceeași semnificație ca și în cazul metodelor OMT prezentate în capitolul 2. O operație în UML poate avea una dintre următoarele proprietăți 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 secvențial;
- *guarded* – se pot produce simultan apeluri multiple către o instanță dar numai unul are permisiunea să înceapă, celelalte fiind blocate până la finalizarea operației.
- *concurrent* - se pot produce simultan apeluri multiple către o instanță, fiecare dintre ele putând fi realizate în paralel.

Operațiile pot avea parametri care sunt utilizați pentru specificare și fiecare include un nume, un tip și direcția comunicării. Direcția se poate indica astfel:

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

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

O diagramă tipică de clase este prezentată în figura 3.5.

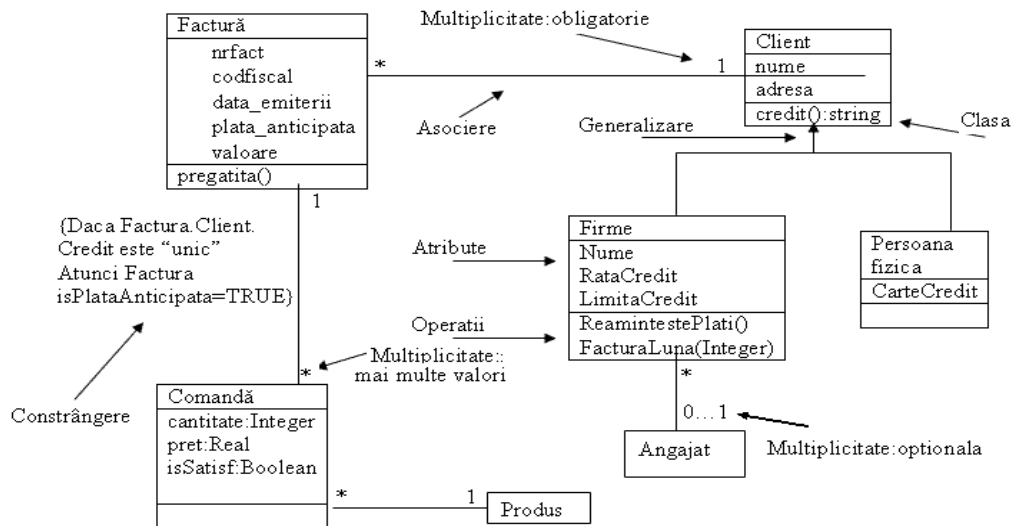


Fig. 4.5. Diagramă de clase

4.2.1.1. Diagrama de obiecte

Această diagramă evidențiază un set de obiecte și relațiile 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 și este o instanță a unei diagrame de clase. Ea arată o instanță a unei stări a unui sistem la un moment dat și conține: obiecte, legături, eventual note și 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 și asupra căruia pot opera alte obiecte. Un agent lucrează în numele unui actor sau altui agent.

- *Legătura* reprezintă o instanță a unei relații de asociere și definește o conexiune între instanțe. O legătură trebuie să aibă un furnizor (desemnează elementul care nu este afectat de o modificare) și un client. Un element furnizor poate participa în mai multe relații de legătură către diferiți clienți. Un element client poate participa numai într-o singură relație de legătură cu un furnizor. O legătură este o dependență unde furnizorul este un model și clientul reprezintă instanțierea modelului care îndeplinește substituirea parametrilor modelului;
- *Note* pentru formularea constrângerilor și a altor comentarii sau alte explicații referitoare la clasificatorul utilizat;

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

Modelarea structurii obiectelor presupune:

- Identificarea mecanismului de modelat (o anumită funcție sau comportamentul unei părți a sistemului);
- Pentru fiecare mecanism, se identifică clasele, interfețele și 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 responsabilități de nivel înalt pentru fluxul lucrării;
- Identificarea condițiilor stărilor inițiale și postcondițiilor stărilor finale;
- Specificarea activităților și acțiunilor începând cu starea inițială;
- Evidențierea tranzațiilor care conectează aceste activități și acțiuni;
- Evidențierea obiectelor importante implicate în fluxul de lucrări, cu evidențierea schimbării valorilor obiectelor.

Modelarea operațiilor presupune:

- Colectarea abstracțiilor implicate în operații (parametri, attribute ale claselor);
- Identificarea condițiilor stării inițiale și postcondițiilor stării finale;
- Specificarea activităților și acțiunilor începând cu starea inițială;



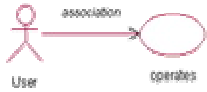
- Folosirea ramificării dacă este necesar;
- Folosirea bifurcării și reunirii pentru specificarea fluxurilor de control paralele.

4.2.2. Diagrama cazurilor de utilizare

Diagrama prezintă funcționalitatea sistemului din punctul de vedere al interacțiunilor externe și este utilizată în etapa de analiză. Elementele UML conținute într-o astfel de diagramă sunt: cazuri de utilizare, actori, relații de utilizare (includere), relații de extindere, relații de generalizare și de asociere. Elementele din această diagramă sunt utilizate în primul rând pentru a defini comportamentul sistemului, a subsistemelor, a unei entități, fără a specifica structura internă. Diagrama cazurilor de utilizare prezintă relația dintre actori, cazuri de utilizare și sistem.

Diagrama este un grafic care conține actori, un set de cazuri de utilizare, posibile interfețe și relațiile dintre acestea.

1. Componentele diagramei cazurilor de utilizare

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

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



Analist



Profesor



Student



Sistem Contabil

Este o entitate internă sistemului (utilizator uman, dispozitiv fizic, un alt sistem), legată de acesta printr-un schimb de informație. Definește un set al rolurilor pe care utilizatorii unei entități le pot avea în cadrul interacțiunii cu aceasta.

2. Cazuri de utilizare

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



Acces informatii



Afiseaza comanda



Afiseaza cererea

3. Relații *Uses* și *Extends*

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

Relațiile 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 față de acesta. Exemplul tipic pentru o astfel de diagramă este prezentat în figura 3.6. și se referă la modelul cumpărării de produse online. Săgețile pline din figuri reprezintă cazuri “copii” ().

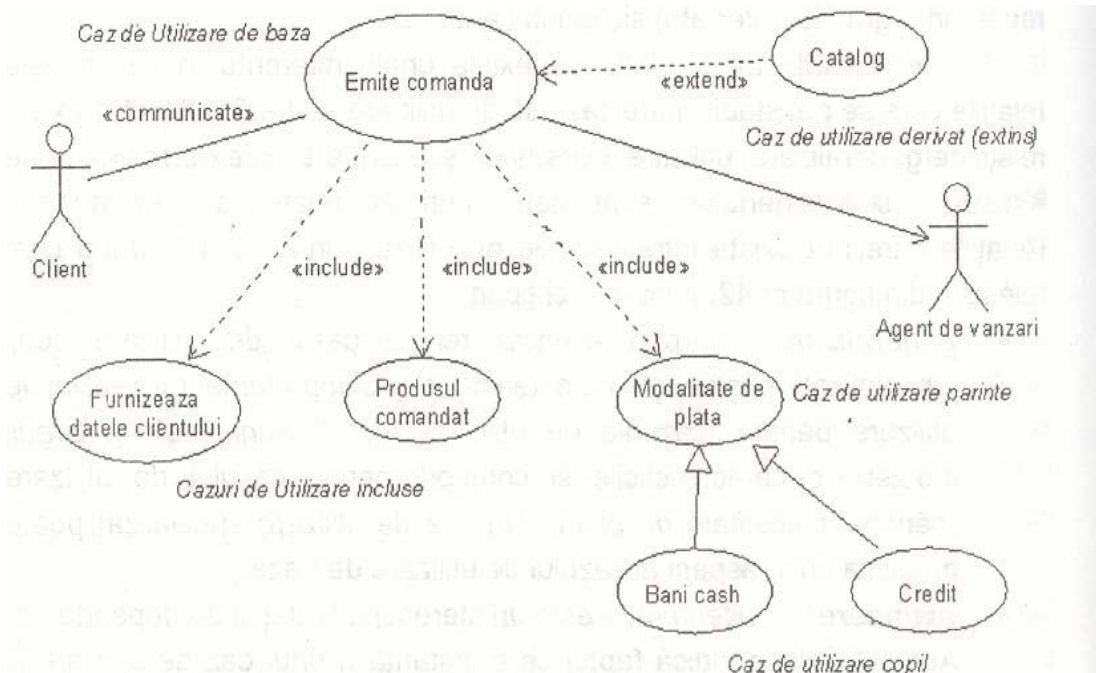


Fig. 4.6. Diagrama cazurilor de utilizare pentru comerț on-line

Un alt exemplu de diagramă a cazurilor de utilizare este prezentat în figura 4.7. și se referă la un sistem de evidență a studenților.

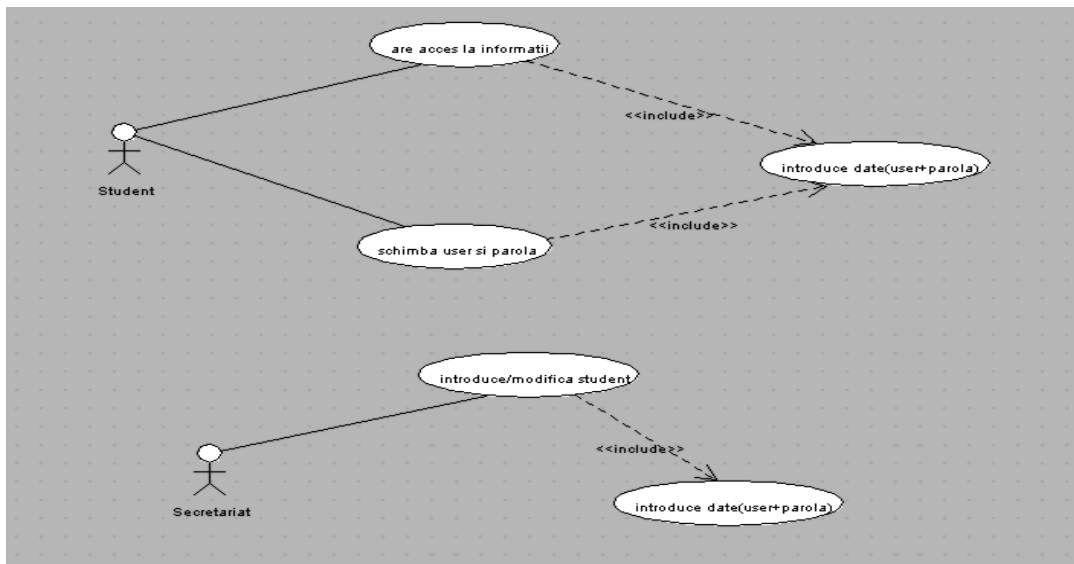


Fig. 4.7. Diagrama cazurilor de utilizare pentru un sistem de evidență a studenților

Un alt exemplu de diagramă este ilustrat în figura 3.8. Este prezentat un caz de utilizare al unui sistem destinat urmării activității unei agenții de turism. Utilizatorul (actor în sistem) se autentifică și poate deveni client al agenției. De asemenea un ofertant de servicii turistice se poate adresa agenției și devine utilizator, dar de alt tip. Operatorul (angajat al agenției de turism), actor și el în sistem, poate vizualiza clienții existenți, ofertele și poate face rezervări.

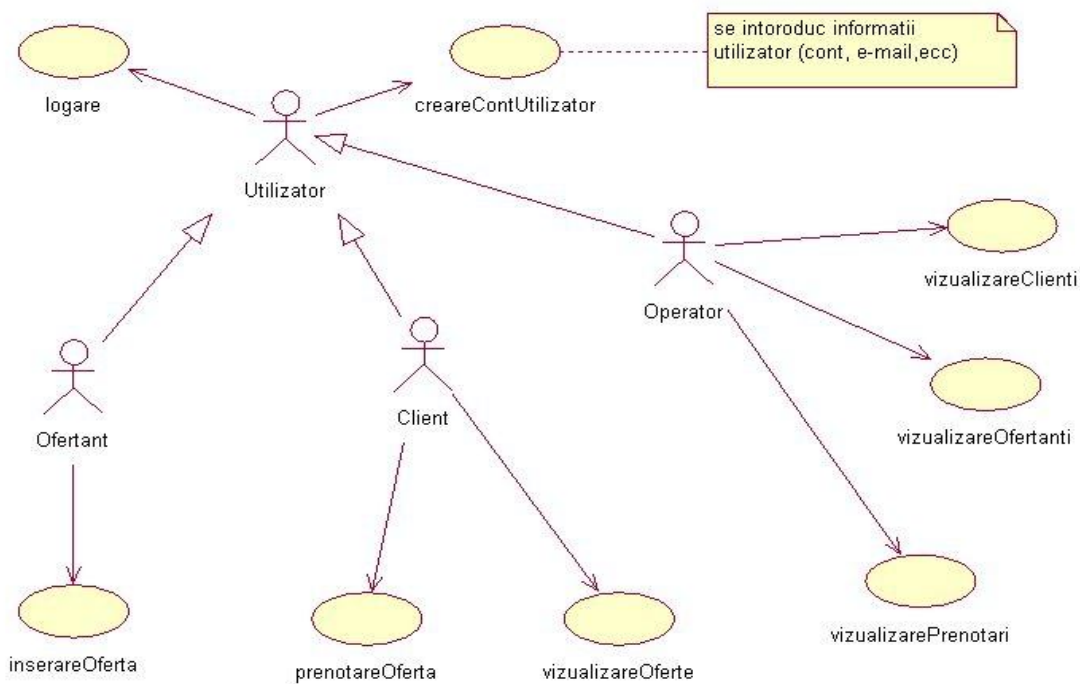


Fig. 4.8. Diagrama cazurilor de utilizare pentru o agenție de turism

4.2.3. Diagrame de comportament

4.2.3.1. Diagrama de stare

La fel ca și în abordarea OMT, prezentată în capitolul 3, diagrama de stare modelează comportamentul unui singur obiect (instanță 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 conține următoarele elemente: stări, mașini de stări, tranziții, evenimente, acțiuni și bare de sincronizare. O *mașină de stări* este o succesiune de stări prin care trece un

obiect, pe durata sa de viață ca răspuns la evenimente. O mașină 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 activitățile ce au loc în obiect.

O stare reprezintă o situație din viața unui obiect, putând satisface anumite condiții, realizând activități sau așteptând producerea unor evenimente.

O stare poate fi:

- Inițială, sau de început – arată momentul de inițiere a mașinii de stare sau al unei substări și se reprezintă în diagramă printr-un cerc înnegrit;
- Intermediară – o stare prin care trece mașina de stare;
- Finală – mașina de stare a fost executată. Se reprezintă prin două cercuri concentrice, cercul din interior fiind înnegrit.
- Compusă – are în componență mai multe substări disjuncte. Se reprezintă printr-un dreptunghi împărțit pe orizontală în două zone.
- Concurențială. Se reprezintă printr-un dreptunghi împărțit pe orizontală în trei zone.

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

Tranziția reprezintă trecerea de la o stare la alta dintr-o diagramă 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 și reprezintă mașina de stare pentru un obiect *Produx*. Diagrama de stare compusă pentru același obiect este ilustrată în fig.3.10.

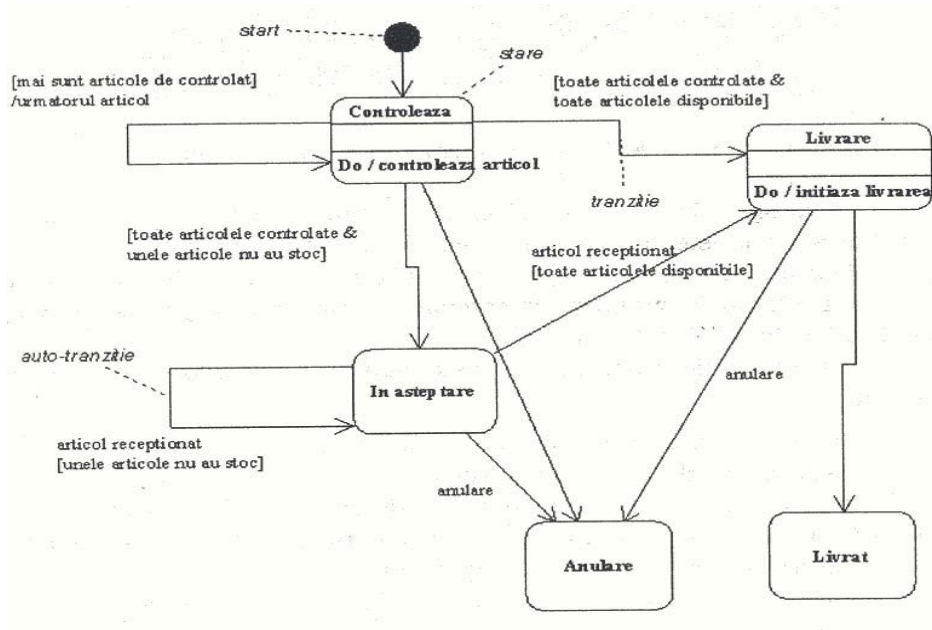


Fig. 4.9. Diagramă de stare

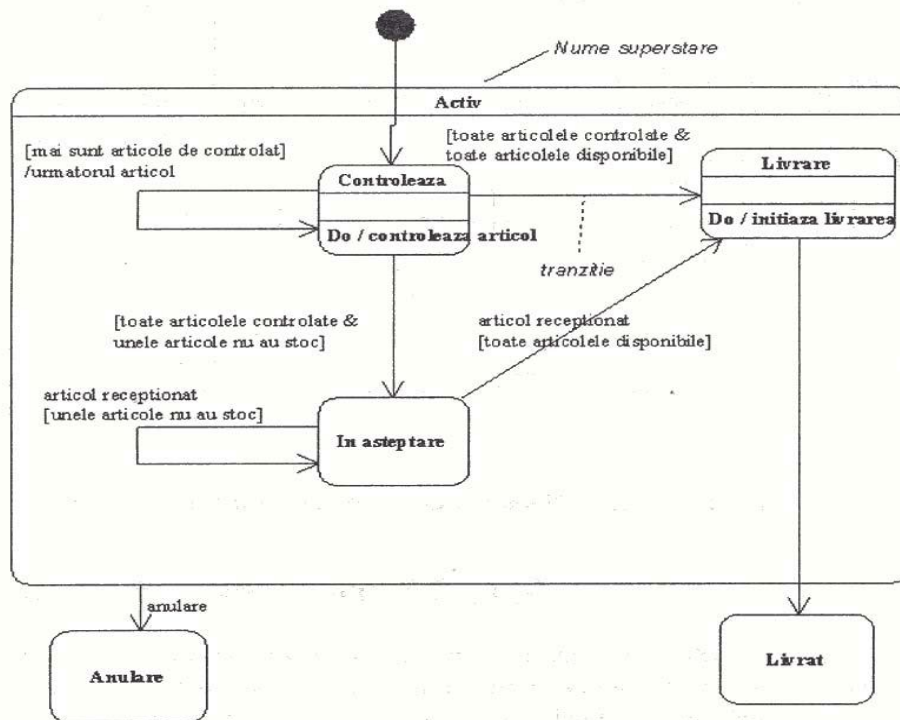


Fig.4.10. Diagrama de stare compusă

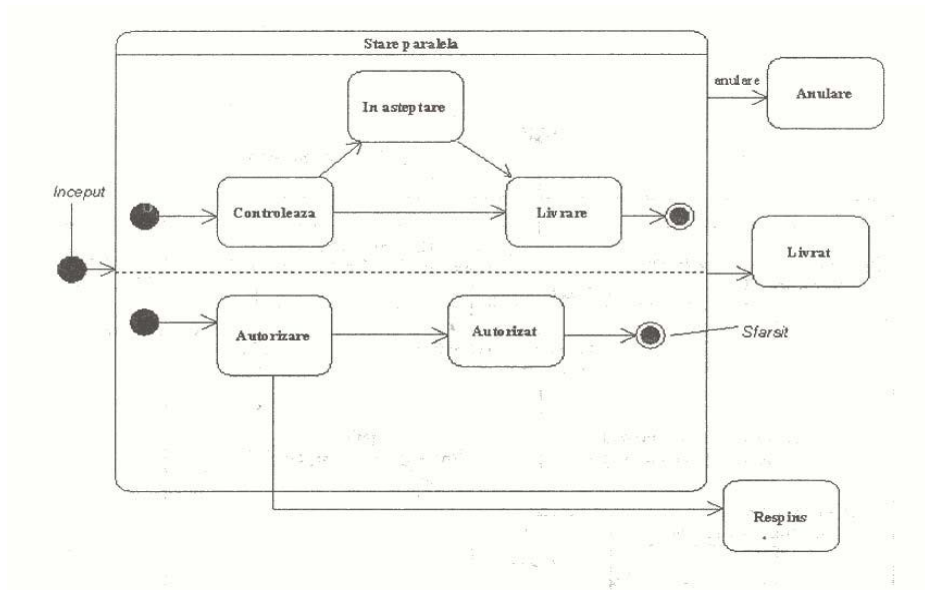


Fig. 4.11. Diagrama de stare concurențială

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

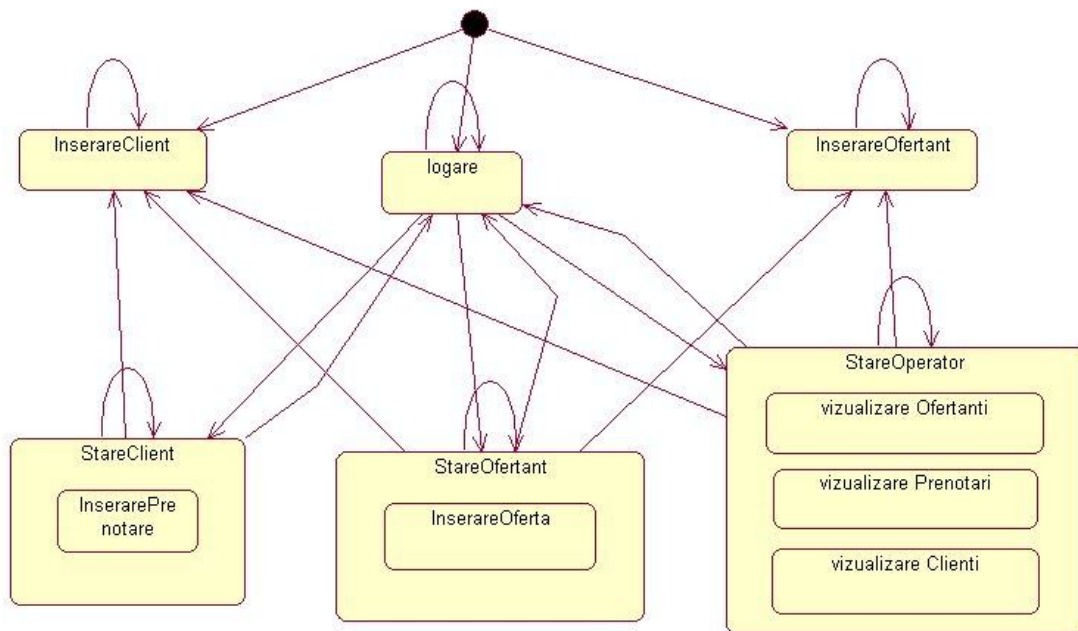


Fig. 4.12. Diagrama de stare compusă pentru agenția de turism

4.2.3.2. Diagrama de activitate

O diagramă de activitate este un caz particular al diagramelor de stare UML, care definește un proces ce evoluează de-a lungul acțiunilor sale. Această diagramă nu extinde semantica diagramelor UML dar definește forme prescurtate care se aplică modelării proceselor. Este utilizată mai ales atunci când este necesară evidențierea legăturii fiecărui serviciu sau a unui multor procese paralele. Diagrama evidențiază fluzul de control de la o activitate la alta.

O activitate rezultă dintr-o anumită acțiune (apelul unei operații, 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 mașini de stări, în care stările reprezintă performanța activităților sau subactivităților.

O stare de activități reprezintă o subactivitate care are o anumită durată și este constituită dintr-un set de acțiuni.

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

Diagrama de activitate conține următoarele elemente UML: *stări* (de activitate, de acțiune, de început, de sfârșit), *tranziții*, *obiecte*, *decizii*, *semnale* (recepționate sau transmise), *bare de sincronizare*, *culoare (swimlane)*. Simbolurile lor grafice sunt:

- Reprezintă starea inițială, începutul procesului;
- ⦿ Reprezintă starea finală sau sfârșitul procesului (nu este absolut necesară figurarea stării finale, dacă aceasta reiese clar din contextul activităților reprezentate).
- ▭ Starea de acțiune – stările prin care este posibil să treacă programul în funcție de ceea ce are de executat.
- ◇ Bloc de condiționare ce împarte secvența în mai multe alternative
- ↓
↕ Bloc de bifurcare - este similar conectorului logic "și". Firele de execuție care pleacă din acest element se pot desfășura paralel sau pe rând.

- 
 Bloc de reunire – element prin care se sincronizează firele de execuție.
- 
 Bloc de sincronizare. Este folosit în cazul subsecvențelor concurente pentru a sincroniza relația “producator-consumator” astfel încât consumatorul să utilizeze resursele disponibile la un moment dat.
- 
 Tranziție –menține stările active și elementele modelului împreună.
- 
 Reprezintă dependențe. Se pot trasa între oricare dintre elementele modelului.

Pentru o mai bună înțelegere a diagramelor de activități 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 cărți împrumutate. Pentru acest caz diagrama de activitate este prezentată în figura 4.13.

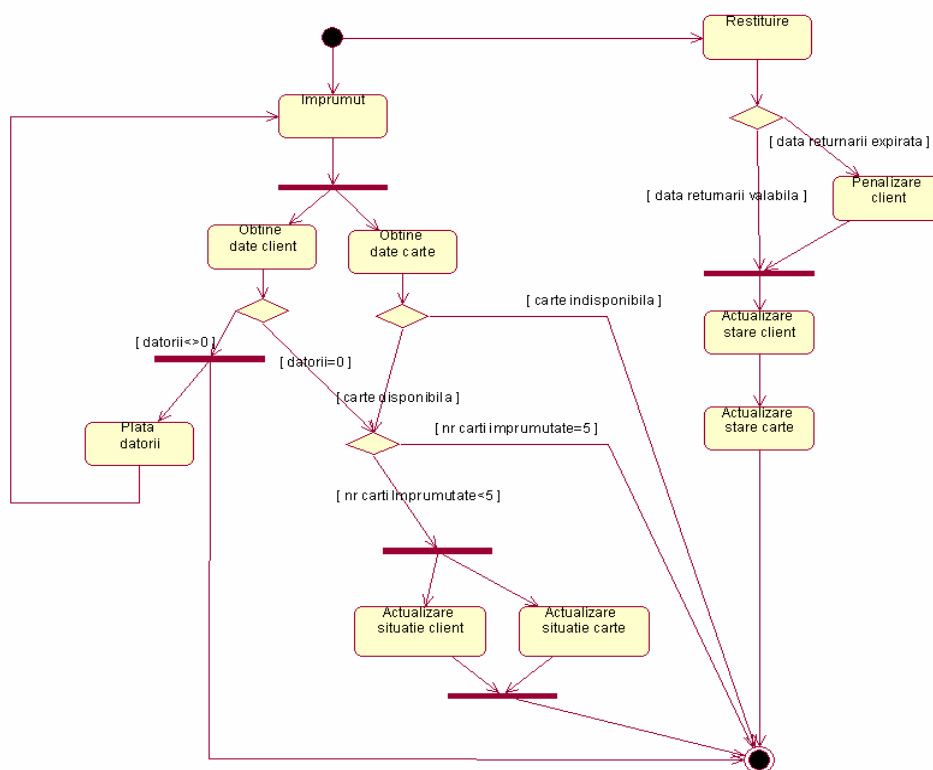


Fig. 4.13. Diagrama de activitate pentru sistemul “Biblioteca”

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

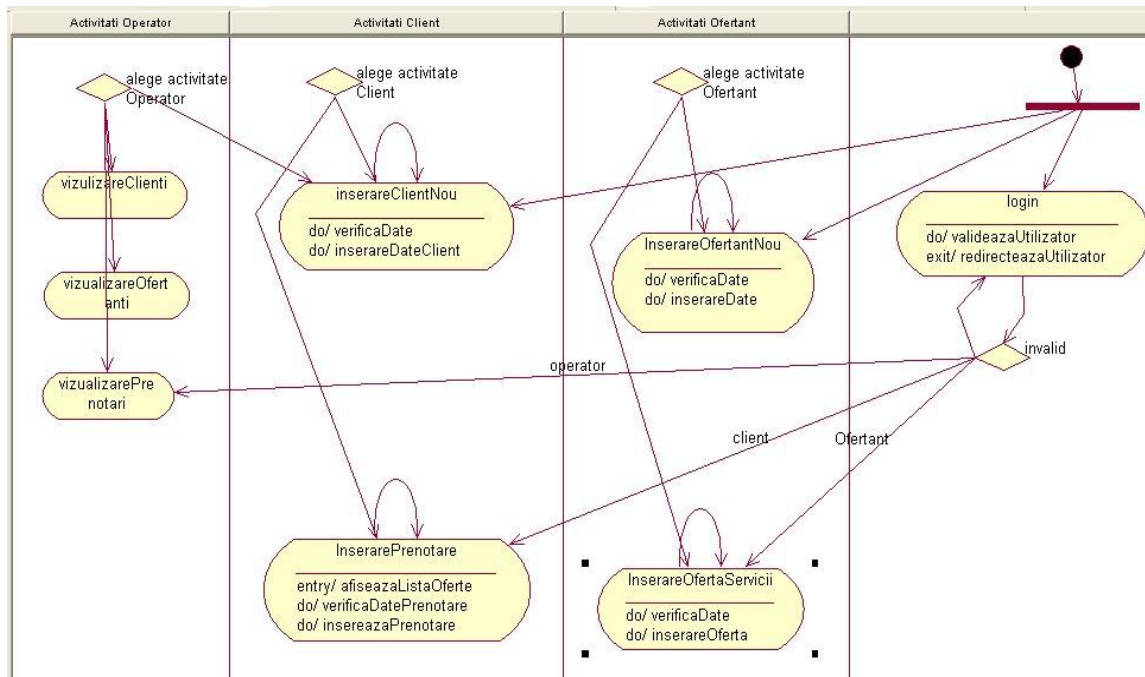


Fig. 4.14. Diagrama de activități pentru sistemul Agenția de turism

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 și de logare a acestuia în calitate de client.

Diagrama de activitate poate fi utilizată pentru:

1. Modelarea fluxului de activitate, activități așa cum sunt văzute de actorii din sistem. Aceasta presupune:
 - selectarea obiectelor care au responsabilități de nivel înalt pentru fluxul de activitate;
 - identificarea precondițiilor stărilor inițiale și postcondițiilor stărilor finale;
 - specificarea activităților și acțiunilor începând cu starea inițială;
 - evidențierea tranzițiilor care conectează aceste activități și acțiuni;

- precizarea obiectelor importante implicate în fluxul de activitate, cu evidențierea schimbării valorilor.

2. Modelarea operațiilor:

- colectarea abstracțiilor implicate în operații (parametri, atributele claselor);

- identificarea precondițiilor stărilor inițiale și postcondițiilor stărilor finale;

- specificarea activităților și acțiunilor începând cu starea inițială;

- folosirea ramificărilor, dacă este necesar;

- folosirea bifurcării și reunirii pentru specificarea fluxurilor de control paralele.

4.2.3.3. Diagrame de interacțiuni

Diagramele de interacțiuni sunt modele care descriu modul în care grupuri de obiecte colaborează în același comportament. De obicei o diagramă de interacțiuni capturează comportamentul unui singur caz de utilizare. Diagrama prezintă un obiecte și mesaje care se schimbă între acele obiecte în cazul de utilizare respectiv.

Există două tipuri de diagrame de interacțiune: diagramele secvențiale și diagramele de colaborare.

4.2.3.4. Diagramele secvențiale

Diagramele secvențiale sau diagrame de secvență sunt reprezentări alternative pentru interacțiuni între obiecte. Ele reprezintă interacțiunile între obiecte din punct de vedere temporal, contextul obiectelor nefiind prezentat în mod explicit (ca în diagramele de colaborare), accentul concentrându-se pe exprimarea interacțiunilor.

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

Diagrama de secvențe, 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 secvențial. Diagrama de secvențe arată

ordonarea în timp secvențial a interacțiunilor între obiecte, iar în particular, aceasta arată obiectele care participă la o interacțiune și succesiunea mesajelor care sunt schimbate.

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

- Stabilirea contextului interacțiunii (sistem, subsistem, operație, clasă, un scenariu al unui caz de utilizare sau colaborare);
- Identificarea obiectelor care joacă rol în acțiune;
- Stabilirea pentru fiecare obiect a duratei de viață în timpul interacțiunii. Pentru obiectele create și distruse în timpul interacțiunii trebuie să se indice explicit, prin mesaj, acest lucru;
- Pentru fiecare mesaj începând cu primul (care inițiază acțiunea) și continuând cu celelalte în ordinea succesiunii, se prezintă proprietățile (parametrii);
- Timpul și spațiul cerut pentru fiecare mesaj;
- Precondiții sau postcondiții pentru fiecare mesaj.

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

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

Firele verticale întrerupte reprezintă durata de viață a unui obiect. Mesajele indicate pe săgețile 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.

Așa cum am precizat deja în interiorul unei diagrame secvențiale, obiectele sunt plasate la începutul diagramei. Dedesubtul fiecărui obiect se află o linie întreruptă, care este numită *linia de viață a obiectului*, și care reprezintă durata de viață a obiectului în timpul interacțiunii.

Fiecare mesaj este reprezentat de o săgeată plasată între liniile de viață ale celor două obiecte care interacționează. Ordinea în care aceste mesaje sunt transmise este de la începutul către sfârșitul diagramei. Fiecare mesaj are o etichetă cu numele mesajului; de asemenea se pot include argumente și unele informații de control și se pot folosi auto-delegațiile.

Auto-delegația este un mesaj pe care un obiect și-l transmite singur, reprezentat de o buclă întoarsă către linia de viață a obiectului.

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

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

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

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

În fig. 4.15 este prezentată diagrama secvențială pentru același sistem exemplificat și în diagrama cazurilor de utilizare și destinat unei agenții de turism. Diagrama a fost realizată în mediul Visual Paradigm. Dreptunghiurile verticale situate pe liniile de viață ale obiectelor reprezintă activarea metodelor. Simbolurile X prezente la sfârșitul liniilor de viață indică ștergerea obiectului. Săgețile întoarse reprezintă auto-delegația.

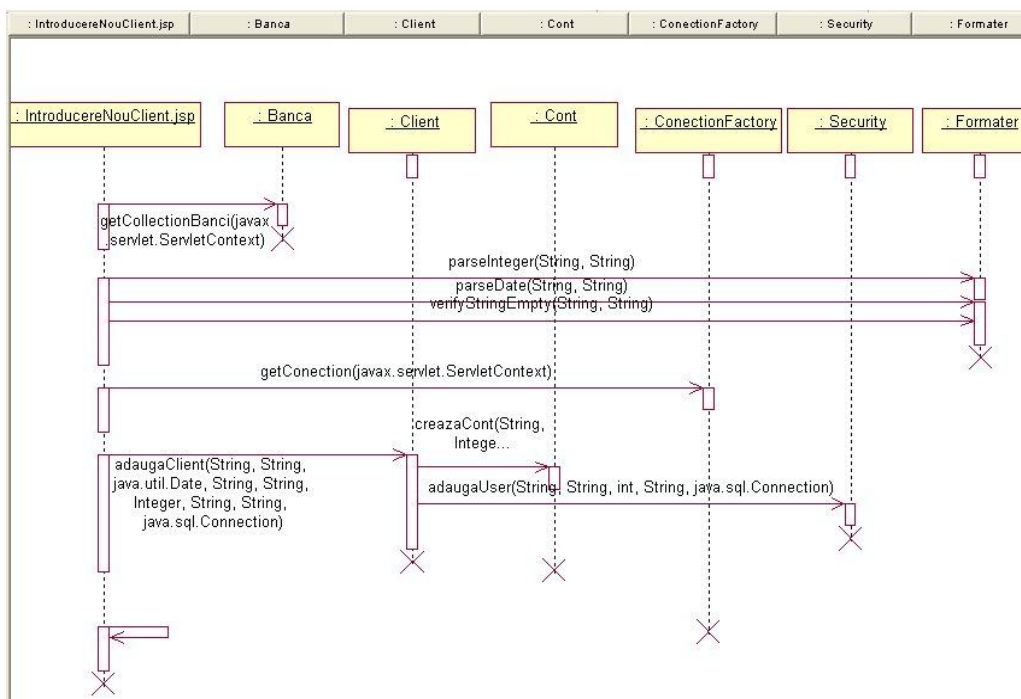


Fig. 4.15. Diagrama secvențială pentru o agenție de turism

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

4.2.3.5. Diagrame de colaborare

Diagrama de colaborare este un tip de diagramă de interacțiune, înrudită cu diagrama secvențială , cu diferența că, în acest caz, accentul cade pe interacțiunea (comunicarea prin schimb de mesaje) între diferitele obiecte implicate într-un caz de utilizare și nu pe succesiunea în timp a mesajelor .

Secvențialitatea acestora poate fi totuși modelată prin numerotare (nu prin dispunerea de-a lungul axei care simboliza durata de viața a unui obiect) .

Fiecare diagramă de colaborare realizează o vedere de ansamblu a legăturilor sau a relațiilor structurale ce se stabilesc între obiectele și entitățile 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 (instanțe ale claselor), legăturile (instanțe ale asocierilor definite între clase în diagrama claselor), și mesajele care pot fi asociate bidirecțional legăturilor.

Un obiect are: stare, comportament și identitate. Fiecare obiect din diagramă indică o instanță a clasei. Simbolul de obiect este similar cu cel al clasei exceptând faptul că numele este subliniat. Dacă se utilizează același nume pentru mai multe obiecte utilizate în aceeași diagramă, ele se presupun a reprezenta același obiect; pe de altă parte, fiecare simbol de obiect reprezintă în mod distinct un obiect.

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

Mesajele dintre obiecte reprezintă comunicarea dintre acestea și indică acțiunea în desfășurare. Mesajul se scrie orizontal pe o săgeată ce face legătura dintre două obiecte.

Un mesaj se poate reprezenta în trei moduri: mesajul singur, mesajul însoțit de numărul secvenței sau mesajul cu numărul secvenței și o etichetă.

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

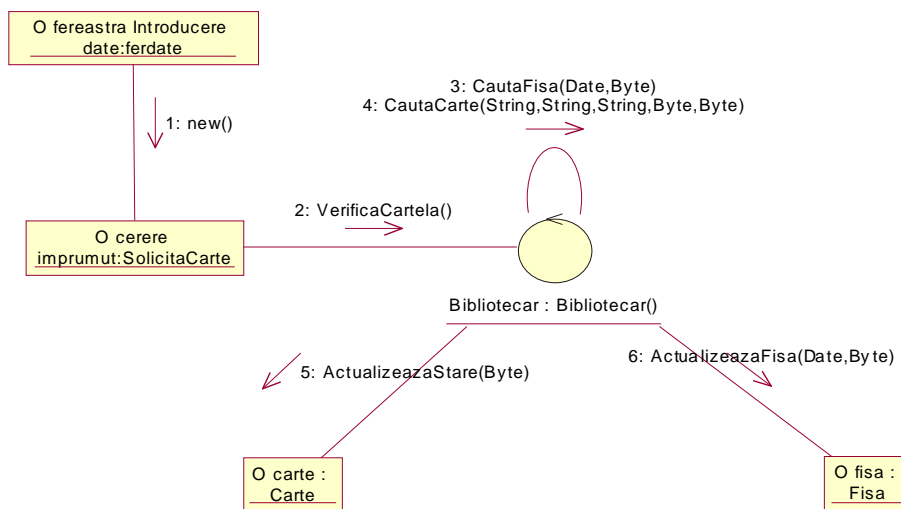


Fig. 4.16. Diagrama de colaborare pentru împrumut de carte dintr-o bibliotecă

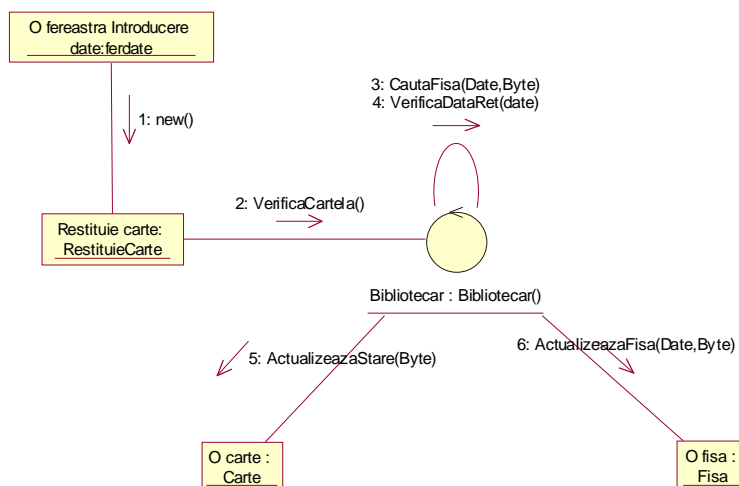


Fig. 4.17. Diagrama de colaborare pentru scenariu de restituire a unei cărți

Pentru o mai bună înțelegere și eventual o comparație între cele două tipuri de diagrame de interacțiuni (diagrama secvențială și diagrama de colaborare) figura 4.18. reprezintă diagrama de colaborare pentru sistemul destinat agenției de turism discutat anterior.

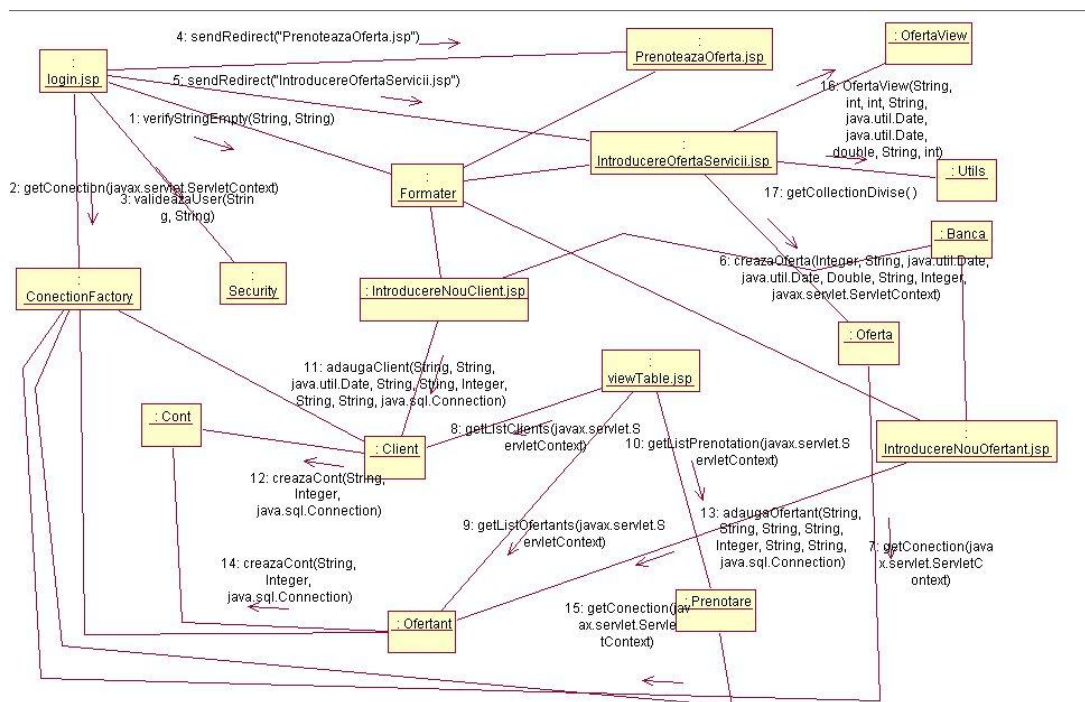


Fig. 4.18. Diagrama de colaborare pentru agenția de turism

4.2.4. Diagrame de implementare

Diagramele de implementare, așa cum le arată și numele, relevă aspecte ale implementării, incluzând cod sursă și execuția. Există două tipuri de astfel de diagrame și anume: diagrame de componente și diagrame de aplicație.

4.2.4.1. Diagrama de componente

Diagrama de componente este utilizată în modelarea aspectelor fizice ale sistemelor prezentând modul de organizare și relațiile de dependență între componente și obiecte. O astfel de diagramă are în componență următoarele elemente UML: componente, obiecte, interfețe și relații de dependență.

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 instanță a unei componente reprezintă o implementare run-time și poate fi utilizată pentru a arăta implementarea unit-urilor care au identitate în momentul execuției aplicației. Componentele unui sistem, incluzând programe, DLL, etc., pot fi amplasate în noduri.

Pentru a figura dependențele dintre diferite componente se utilizează o linie întreruptă de la o componentă la alta sau, de la o componentă la interfața altei componente. Alte elemente care pot fi incluse sunt: note, legături, note atașate.

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: *Clienți*, care este o bază de date, *Comenzi*- bază de date, *Evidență comenzi*- aplicație. De asemenea în diagramă există și un obiect, *Popescu Ion*, care este instanță a clasei *Client*. Relațiile dintre aceste elemente sunt relații de dependență.

Fig.4.20. prezintă o diagramă cu patru componente: *Comenzi*, *Terți* și *Produse*, care sunt fișiere și *Evidență terți*, care este aplicație.

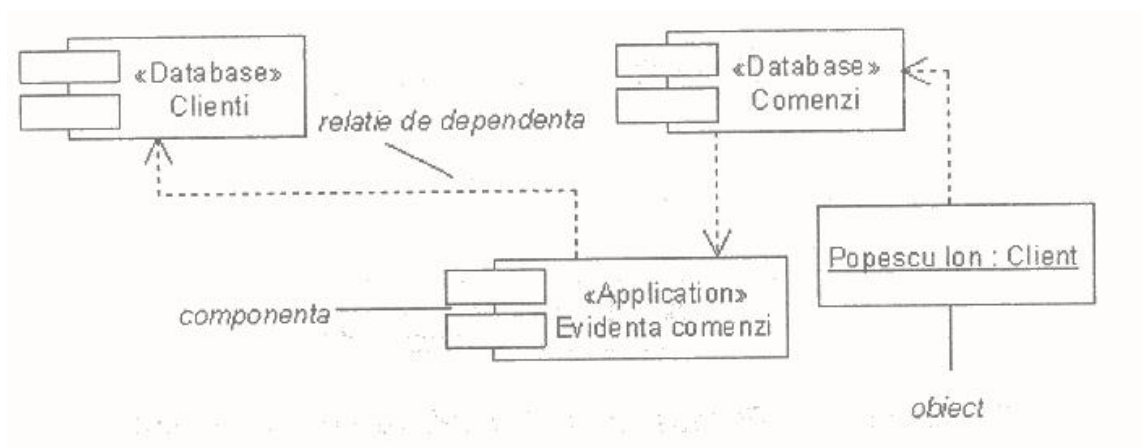


Fig. 4.19. Diagrama cu trei componente

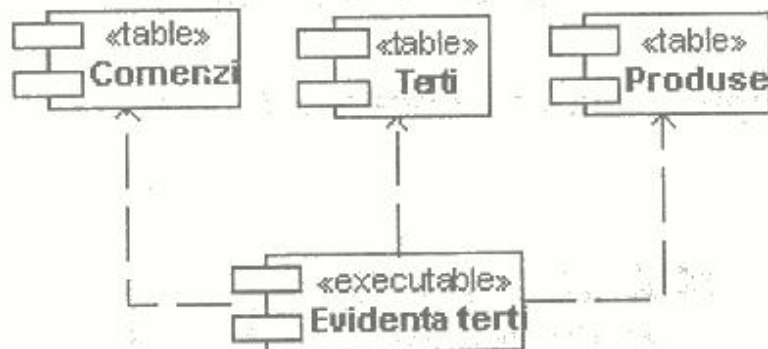


Fig. 4.20. Diagrama cu patru componente

4.2.4.2. Diagrama de aplicație (deployment diagram)

Diagramele de aplicație sau de desfășurare arată structura fizică a sistemului hardware și software, mai precis configurarea elementelor de procese run-time, a componentelor software, și a proceselor și obiectelor. Au în componență următoarele elemente UML: noduri, componente, obiecte, relații de dependență, relații de asociere (comunicație), precum și elemente ajutătoare: note, legături, etc.

Componentele au același rol și funcții ca în diagrama componentelor. Nodurile sunt obiecte fizice care există în timpul execuției aplicației și reprezintă de obicei resurse de prelucrare. Ele includ componente ale calculatoarelor, resurse umane sau resurse de procesare mecanică.

Diagramele de aplicație arată configurația elementelor de procesare în timpul execuției aplicației, precum și componenete software, procese și obiecte. Ele sunt utilizate pentru a modela viziunea asupra desfășurării statice a sistemului. Componentele care nu există ca entități la execuție (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 relații de comunicare. Nodurile pot conține și instanțe ale componentelor. Acest

lucru indică faptul că acele componente există sau rulează în acele noduri. La rândul lor componentele pot conține obiecte.

Componentele sunt conectate cu alte componente prin intermediul relațiilor de dependență (linii întrerupte în diagramă), sau prin interfețe. Aceste interfețe 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 relații de dependență.

O diagramă de aplicație, pentru o aplicația cu clienți și furnizori este prezentată în fig. 4.21.

Diagramele de aplicație se folosesc în următoarele cazuri:

- Modelarea sistemelor cu software implantat hardware. Diagramele se utilizează pentru a modela dispozitivele și procesele care compun sistemul.
- Modelarea sistemelor client-server. Un astfel de sistem este o arhitectură focalizată pe realizarea unei separări nete între interfața sistemului cu utilizatorul (de la client) și datele permanente ale sistemului (de pe server).
- Modelarea sistemelor complet distribuite.

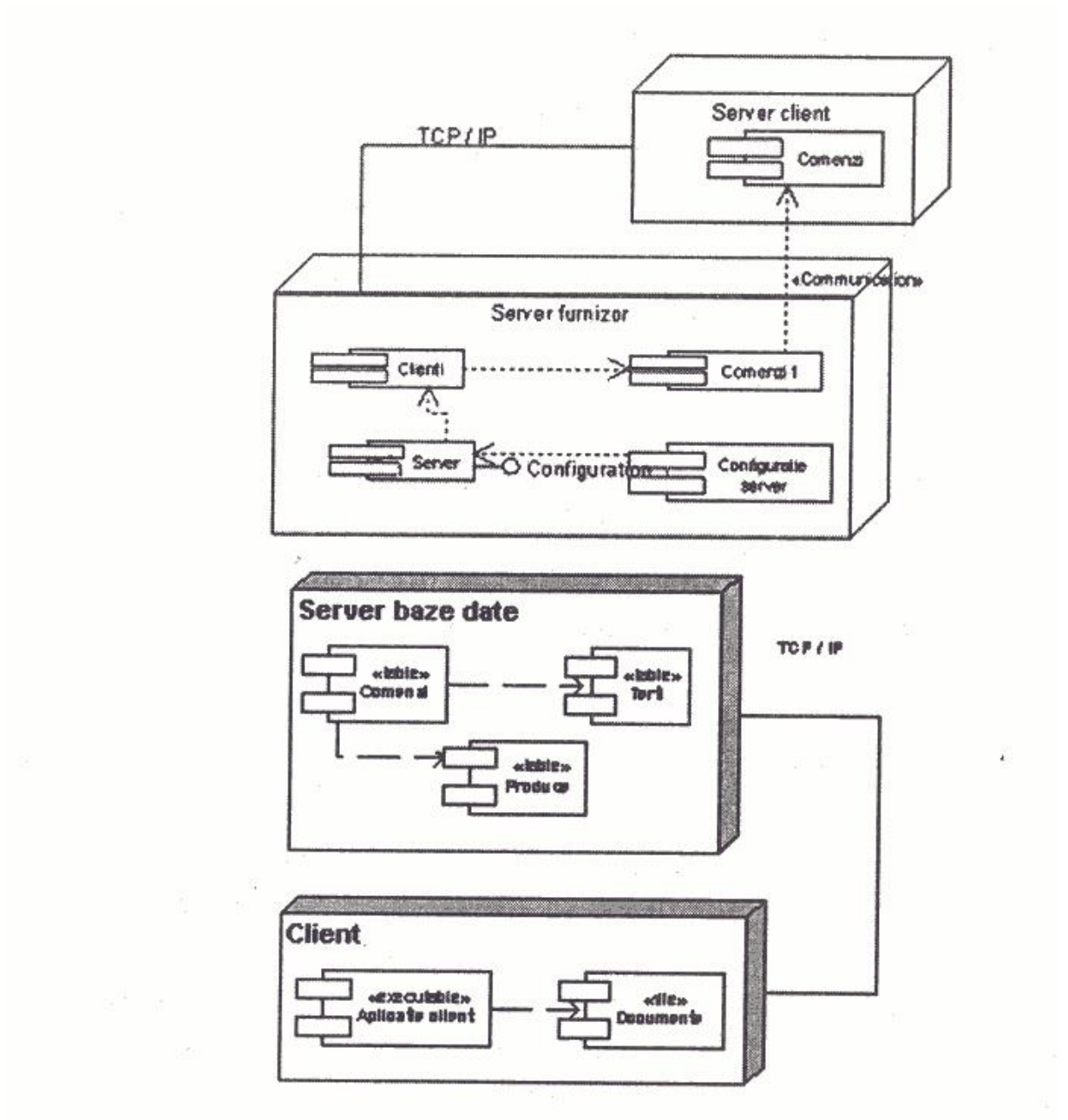


Fig. 4.21. Diagrama de aplicație

5

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 obținerii 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 față 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

Principiul Deschis – Închis (in engl. Open – Close Principle, OCP), a fost formulat de către Bertrand Meyer în cadrul unui articol în 1988, astfel:

*"Orice entitate software (clase, module, funcții, etc) ar trebui sa fie **deschisă** pentru **extindere**, și **închisă** pentru **modificare**." (Software entities should be open for extension, but closed for modification)*

Prin utilizarea principiului deschis – închis se evită fragilitatea, rigiditatea și imobilitatea piesei de soft, proiectându-se module care "să nu se modifice niciodată". În proiectarea sistemului de folosește **abstractizarea și polimorfismul**. Când specificațiile sistemului se modifică, comportamentul

modulelor se extinde prin adăugarea de cod nou, fără a interveni cu modificări în codul deja existent, care funcționează.

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

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

Deși la o prima vedere cele două cerințe par contradictorii, totuși 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 totuși 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 îndeplinește restricția 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 aplicații simple, în care atât clasa *Client* cât și clasa *Server* sunt două clase concrete. Clasa *Client* utilizează clasa *Server*. Dacă se dorește 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**.

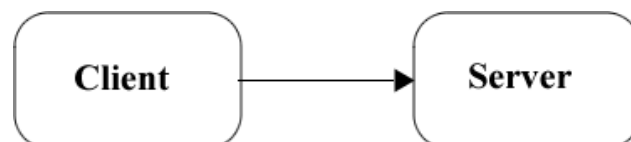


Fig. 5.1 Proiectare greșită Client-Server

Fig. 5.2 prezintă design-ul pentru aplicația **Server – Client** astfel încât să respecte principiul deschis – închis. În acest caz, se utilizează o clasă abstractă *AbstractServer*, din care se derivează clasa concretă *Server* și clasa *Client* depinde de această clasă abstractă. Obiectele *Client* vor folosi obiecte ale unei clasei derivate *ServerA*. Dacă se dorește 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.

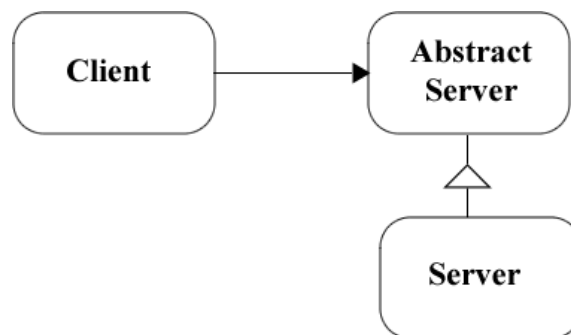


Fig. 5.2. Proiectare corectă Client-Server

Pentru obținerea unor noi comportamente necesare extinderii aplicației, 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%.” [8]. Nici un program nu poate fi închis pentru modificări în mod total, deoarece, oricât de închis ar fi un modul, întotdeauna pot apărea situații pentru care nu a fost închis. Având în vedere că, închiderea unui modul față 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 obține utilizând **abstractizarea**. Clasa abstractă furnizând metode care pot fi invocate dinamic și 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 locație separată, fie într-un alt fișier, fie într-un alt obiect, astfel că pe viitor modificările se vor face într-un număr minim de locații.

Reguli de proiectare folosite în OOD

Principiul deschis – închis este sursa unor reguli de proiectare folosite în OOD. Acestea se referă la variabile private și variabilele globale folosite în program:

Toate variabilele să fie private

Aceasta este dintre cele mai întâlnite convenții 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*, și nu *public* sau *protected*.

Privit din prisma OCP, motivația acestei convenții este următoarea: când variabilele dintr-o clasă se modifică, fiecare dintre clasele care depind de acestea ar trebui modificate. Deci, nici o funcție care depinde de o variabilă, nu poate fi închisă față 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, față 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;
- **consistență**: prin modificarea variabilelor din alte clase, de către diferiți utilizatori pot apărea situații de inconsistență, 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 față de celelalte module care modifică respectiva variabilă. Este posibil ca un modul să utilizeze variabila globală într-un mod neașteptat pentru celelalte module care o mai folosesc.

Designerul trebuie să evalueze cât din închiderea aplicație este "sacrificată" în favoarea variabilelor globale și să determine dacă avantajele oferite de utilizarea variabilelor globale compensează dezavantajele date de utilizarea lor.

Concluzii

În multe privințe acest principiu este considerat esența proiectării orientate pe obiecte. Avantajele care se obțin de pe urma folosirii acestuia sunt reutilizarea codului și mentenabilitatea software-ului. OCP afirmă ca un design bine gândit poate fi extins fără modificări, noile caracteristici ale sistemului adăugându-se prin completarea de cod nou, și nu prin modificarea celui existent.

Mecanismele pe care se sprijină acest principiu sunt abstractizarea și polimorfismul.

Faptul ca se programează într-un limbaj orientat pe obiecte nu duce automat la concordanță / conformitate cu OCP, ci este necesar ca designerul să aplice abstractizarea pe acele părți ale programului care sunt susceptibile de a fi modificate.

Crearea abstractizărilor și apoi derivarea claselor concrete din aceste abstractizări, duc la obținerea unor module deschise pentru extindere și închise pentru modificare. Mecanismul care asigura crearea acestor clase derivate este moștenirea, iar principiul substituției al lui Liskov este cel care ghidează designul acestor ierarhii de clase.

5.2. Principiul substituției Liskov

Principiul substituției al lui Liskov (Liskov Substitution Principle, LSP) a fost enunțat în literatura de specialitate astfel:

Funcțiile care utilizează pointeri sau referințe 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 (obținerea unui cod reutilizabil, prin extindere și nu prin modificare); deci LSP poate fi interpretat ca un mijloc de verificare a codului, din punctul de vedere al obținerii 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 referință către clasa de bază, și în același timp "știe", conform presupunerii de mai sus, despre toate clasele derivate din clasa de bază. O astfel de metodă încalcă OCP deoarece trebuie modificată ori de câte ori o nouă clasă derivată din clasa de bază este creată.

Exemple pentru ilustrarea LSP: Fie o clasă *Rectangle*:

```
public class Rectangle {
    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 și în Java, moștenirea se bazează pe modelul relațional ISA. Modelul relațional ISA descrie o relație strictă între clase, în care membrii unei clase formează o submulțime a unei alte clase. Modelul ISA (“is a”) pune în evidență că un obiect (*RombA* din Fig. 5.3.) care aparține unei subclase (clasa *Romb* din Fig. 5.3), aparține simultan tuturor super-claselor (deci *RombA* este o instanță a lui *Romb*, dar este în același timp instanță și a claselor *Patrulater* și *Poligon*)

Utilizarea modelului ISA este considerată ca fiind o tehnică de bază în Analiza Orientată pe Obiecte (Object Oriented Analysis, OOA).

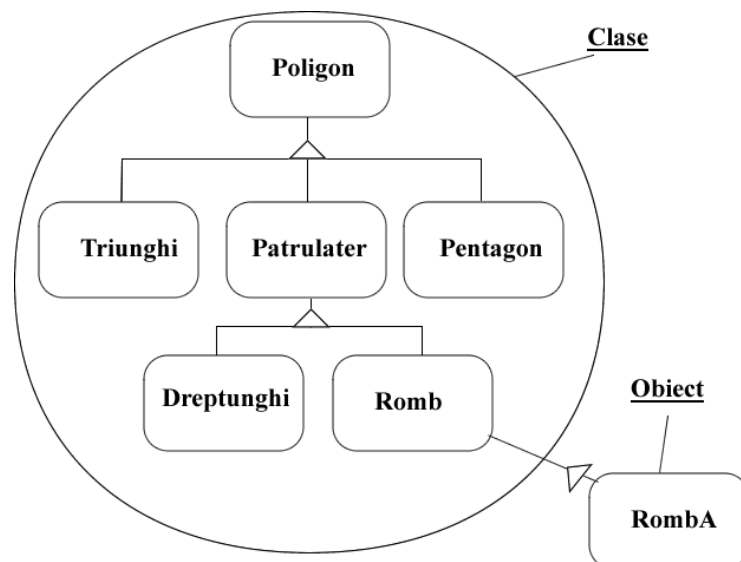


Fig. 5.3. Modelul relațional ISA

În continuarea exemplului de mai sus, să presupunem că la un moment dat, în decursul dezvoltării unei noi aplicații, proiectantul are nevoie să utilizeze și pătrate. Deoarece un pătrat este un dreptunghi cu toate laturile egale, putem modela clasa *Square* prin derivare din clasa *Rectangle*, 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, și cea mai importantă, este legată de variabilele *height* și *width*; pentru a defini un pătrat nu avem nevoie și de lățime și de înălțime (deci de amândouă), dar clasa *Square* le va moșteni pe amândouă. O problemă secundară este legată de folosirea eficientă a memoriei, mai ales dacă se utilizează sute de obiecte ale clasei *Square*.

Dar trecând peste acest aspect al eficienței memoriei utilizate, o problemă importantă este cea generată de existența celor două metode *setWidth* și *setHeight*. În primul rând, aceste metode sunt inadecvate pentru clasa *Square*, deoarece dimensiunile unui pătrat sunt egale. Iată deci o problemă de design, care totuși poate fi depășită dacă modificăm codul celor două metode astfel: când se setează "lățimea" unui obiect *Square*, "înălțimea" va fi ajustată în mod corespunzător, și reciproc. În acest mod un obiect *Square* își păstrează proprietățile matematice.

```
public class Square extends Rectangle{
    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 să fie închis pentru modificări.

Consistența unui model

În acest moment, există două clase, care în aparență funcționează corespunzător. *Square* și *Rectangle*, fiecare modelează corespunzător obiectele

matematice ale căror nume le poartă (pătrat și dreptunghi). În aparență, se poate spune că sistemul are un comportament consistent.

Se poate trage concluzia că modelul este auto-consistent și corect. Dar această concluzie ar fi o eroare **deoarece un model care este auto-consistent nu este în mod necesar consistent și cu toți utilizatorii săi.**

Să considerăm următoarea funcție:

```
void g(Rectangle r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert ((r.getWidth() * r.getHeight()) == 20) ;
}
```

În mod evident aceasta metodă funcționează corespunzător pentru *Rectangle*, dar pentru *Square* generează o aserțiune falsă. Aceasta este o problemă gravă. Desigur, programatorul care a scris această metodă a făcut o presupunere îndreptățită și anume că modificarea lățimii unui dreptunghi lasă înălțimea acestuia neschimbată. Funcția *g(Rectangle)* este un exemplu de funcție care acceptă referințe către *Rectangle* dar care nu funcționează corespunzător pentru obiectele din clasa *Square*. Acest tip de funcții încalcă principiul substituției 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 validității lui. Validitatea unui model se exprimă în termenii clienților săi.

Spre exemplu, în cazul de mai sus când s-a examinat versiunea finală a claselor *Square* și *Rectangle*, în mod izolat, s-a ajuns la concluzia că sunt consistente și valide. Totuși, 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 greșit.

Deci, pentru aprecierea unui proiect, analiza trebuie să fie făcută într-un context și nu în mod izolat. Design-ul trebuie văzut din perspectiva utilizatorilor acestuia; ei lucrează pe baza unor presupuneri rezonabile asupra modului de funcționare a modelului.

Trebuie analizat de ce modelul claselor *Square* și *Rectangle*, care părea corect, nu a funcționat. Nu este *pătratul* un *dreptunghi*? Nu a funcționat relația modelului ISA, orice obiect al clasei *Square* aparține sau nu și clasei *Rectangle*?

Un pătrat poate fi un dreptunghi, dar din punct de vedere strict matematic. Un pătrat ca obiect al clasei *Square* nu este un obiect *Rectangle*, deoarece comportamentul unui obiect *Square* nu este consistent cu comportamentul unui obiect *Rectangle*, iar clasele tocmai acest aspect trebuie să-l modeleze: **comportamentul**.

LSP accentuează faptul că în OOD, modelul relațional ISA se ocupă de comportament. Prin comportament se înțelege **comportamentul public**, extrinsec, cel pe care se bazează clienții, și nu comportamentul privat, intrinsec al obiectului. Spre exemplu, autorul funcției *g()*, de mai sus, s-a bazat pe faptul că pentru un obiect al clasei *Rectangle*, lățimea și înălțimea variază independent. Această independență a celor două variabile este un comportament public extrinsec pe care, probabil, contează și alți programatori.

Pentru ca LSP să fie valabil, implicit OCP, **toate clasele derivate trebuie să respecte comportamentul pe care clienții îl așteaptă de la clasa de bază pe care o utilizează**.

Design prin contract

Există o strânsă relație între LSP și conceptul de “Design by contract”, așa cum a fost definit de Bertrand Meyer [1]. Utilizând această schemă, metodele claselor declară precondiții și postcondiții. O metodă se execută dacă precondițiile sunt adevărate. La terminarea metodei, aceasta garantează că postcondițiile vor fi adevărate.

Aplicând pe exemplul de mai sus, putem spune că postcondiția metodei *setWidth(double w)* în clasa *Rectangle* este:

```
assert((width == w) && (height == old.height))
```

Regula care se aplică precondițiilor și postcondițiilor la derivare, așa cum a formulat-o B. Meyer este următoarea:

Când se redefinește o metodă în clasa derivată, precondiția se înlocuiește prin una mai "slabă", iar postcondiția prin una mai "puternică".

Cu alte cuvinte, utilizarea unui obiect se face prin intermediul interfeței clasei de bază și utilizatorul cunoaște doar precondițiile și postcondițiile acestei clase. Deci, obiectele claselor derivate nu trebuie să impună precondiții mai puternice decât ale clasei de bază, ele trebuie să accepte orice precondiție pe care clasa de bază o acceptă. De asemenea, clasele derivate trebuie să se conformeze tuturor postcondițiilor clasei de bază, comportamentul și ieșirile lor nu trebuie să încalce nici una din constrângerile impuse de superclasă.

Postcondiția din metoda *setWidth(double w)* din clasa *Square* este mai puțin restrictivă decât cea din metoda *setWidth(double w)* din clasa *Rectangle*, deoarece nu se conformează celei din superclasa, mai exact nu respectă și condiția: *(height == old.height)*. Deci, metoda *setWidth(double w)* din clasa *Square* încalcă contractul clasei de bază.

Concluzii

Respectarea OCP are ca efect obținerea unor aplicații mai robuste, cu o mai bună mentenabilitate și reutilizabilitate a codului. Principiul substituției este o caracteristică importantă a acelor programe/aplicații care respectă OCP, deoarece tipurile derivate pot înlocui tipurile de bază astfel încât codul claselor de bază poate fi oricând reutilizat și codul claselor derivate oricând modificat. Substituția claselor de bază cu obiecte ale claselor derivate este posibilă, deoarece clasele derivate respectă comportamentul extrinsec al superclaselor. Obiectele subclasselor, 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.

Mecanismul prin care se implementează o structură ierarhică conformă cu OCP și LSP este prezentat în capitolul următor, Principiul Inversării Dependențelor.

5.3. Principiul Inversării Dependențelor

Structura unui program, rezultată în urma respectării OCP și LSP, este generalizată în principiul inversării dependențelor (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 niște abstractizări.

B. Abstractizările nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstractizări.

Acest principiu este mecanismul de realizare a OCP, deoarece prin aplicarea DIP se creează o arhitectură a sistemului închisă pentru modificări (abstractizările de pe nivelul superior) și deschisă pentru extindere (clase concrete de pe nivelul inferior), deci modulele sunt reutilizabile și stabile.

Metodele tradiționale de dezvoltare a software-ului (programarea structurală) creează structuri în care modulele de pe nivelele superioare depind de cele de pe nivelele inferioare și abstractizările depind de detalii de implementare. Termenul de "inversare" din titulatura principiului, se folosește pentru a evidenția structura dependențelor inversată față de metodele procedurale tradiționale.

În general, modulele de pe nivelul superior conțin logica aplicației, ele dau o identitate aplicației, și totuși ele depind de modulele de pe nivelul inferior, fiind afectate direct de orice modificare în nivelul de care depind. Normal este ca nivelul superior să impună schimbări în nivelul inferior, modulele de pe nivelul superior să fie independente față 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, și nu invers.**

Fie exemplul din **Fig. 5.4** al unui program alcătuit din trei module. Modul de pe nivelul 1, *Client*, implementează logica programului, iar modulele de pe nivelul 2 realizează anumite operații, conform deciziilor luate de modulul *Client*. Așa cum se observă și din figură, modulul *Client* este dependent de modulele de pe nivelul 2. Dacă modulele de pe nivelul 2 mai pot fi reutilizate în aplicații în care să îndeplinească aceleași funcții, 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.

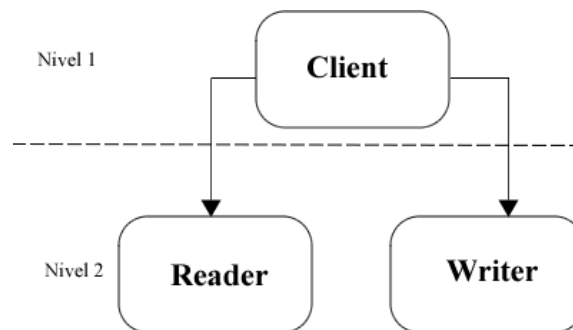


Fig. 5.4. Structura unui program alcătuit din trei module

Situația din exemplu de mai sus poate fi caracterizată prin observația că modulul de pe nivelul superior (logica programului) este dependent de modulele de pe nivelul inferior, module pe care le controlează. 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 aceeași natură. OOD ne pune la dispoziție un mecanism pentru a face acest lucru: **inversarea dependențelor**.

Dacă sistemului din Fig. 5.4 i se aplică principiul inversării dependențelor, conform căruia modulele de pe nivelele superioare nu depind de modulele de pe nivelele inferioare ci de niște abstractizări, iar abstractizările nu depind de "detalii", se obține diagrama de clase din Figura 5.5. Astfel, în această proiectare apar două clase abstracte "AbstractReader" și "AbstractWriter". Modulul *Client* depinde de aceste două abstractizări. În acest fel a fost înlăturată dependența

clasei Client față de clasele Reader și Writer. Dependența a fost inversată, în sensul că atât clasa Client cât și clasele Reader și Writer depind de cele două clase abstracte.

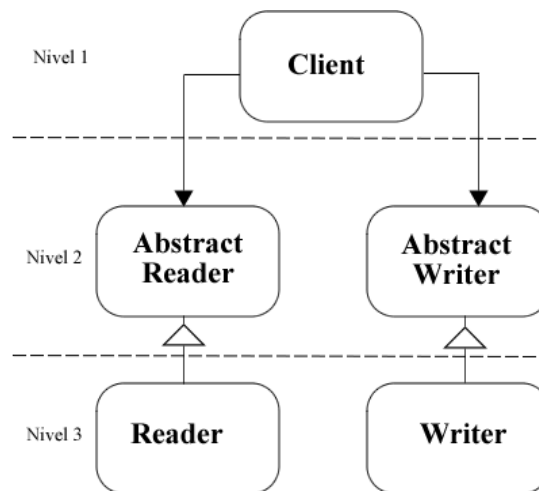


Fig. 5.5. Structura unui program alcătuit din trei module în urma aplicării DIP

Cu acest nou design, clasa Client poate fi reutilizată și în alte contexte, independent de clasele concrete Reader și Writer. Cu ușurință 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 dependențelor se obține o structură organizată pe nivele, cu modulele reutilizabile pe cele două nivele. Modulele de pe nivelul inferior sunt reutilizate în forma librăriilor. Dacă modulele de pe nivelul superior ar depinde de cele de pe nivelul inferior, reutilizarea lor în alt context este dificilă. Conformându-se DIP, dacă ele sunt independente, atunci reutilizarea este ușor de realizat.

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 interfeței sale."

O interpretare simplistă a acestei afirmații ar putea duce la realizarea unei arhitecturi în care dependențele ar fi "pasate" de la un nivel la altul, având în vedere că **dependența este tranzitivă**. (Fig. 5.6).

Nivelul Deciziilor depinde de **Nivelul Mecanismelor**, care depinde de **Nivelul Serviciilor**, deci **Nivelul Deciziilor** depinde de **Nivelul Serviciilor** (tranzitivitatea dependențelor).

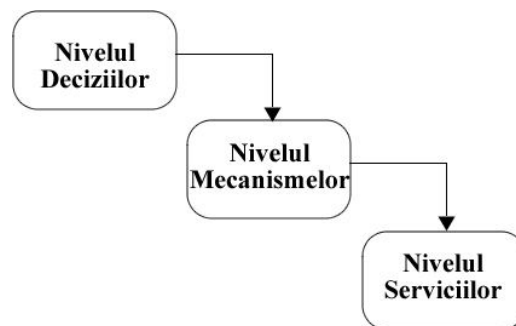


Fig. 5.6 Tranzitivitatea dependențelor

O structură corectă este cea în care nivelele inferioare oferă servicii prin intermediul interfețelor abstracte. (Fig. 5.7).

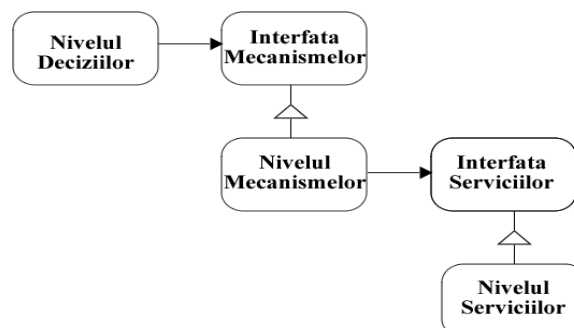


Fig. 5.7 Structură corectă în care nivelurile sunt independente unele față de altele și dependente doar de interfețe

Interfața reprezintă *totalitatea serviciilor pe care nivelul inferior le oferă nivelului superior*. Deci, toate nivelele sunt independente unele față de altele, și dependente doar de clasele abstracte. Nu se mai transferă dependențele de la un nivel la altul, și orice modificare ce se efectuează pe un nivel inferior nu afectează nivelul superior. Cu alte cuvinte, conform acestui model, **Nivelul Deciziilor** este complet independent de nivelele inferioare, putând fi reutilizat în orice alt context în care se definește un nivel inferior bazat pe interfața **Nivelului Mecanismelor**.

Deci, inversând dependențele, se creează o structură care are toate calitățile unui bun design: flexibilitate, durabilitate și mobilitate.

Comparație între o arhitectura procedurală și una OO

Comparația între o arhitectură procedurală și una orientată-obiect are ca scop să pună în evidență avantajele și dezavantajele fiecăreia. În **Fig. 5.8** este schițată arhitectura unui sistem procedural, în **Fig. 5.9** este schițată arhitectura unui sistem orientat pe obiecte.

Dezavantajele unei arhitecturi procedurale sunt:

- nu există o separare clară a nivelului decizional al aplicației de nivelul mecanismelor de implementare și de cel al detaliilor;
- orice modificare în modulele de pe nivelul inferior duce la modificări în modulele de pe nivelul superior.

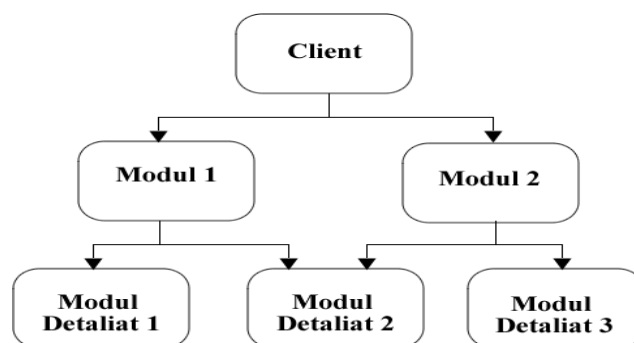


Fig. 5.8. O arhitectură a unui sistem procedural

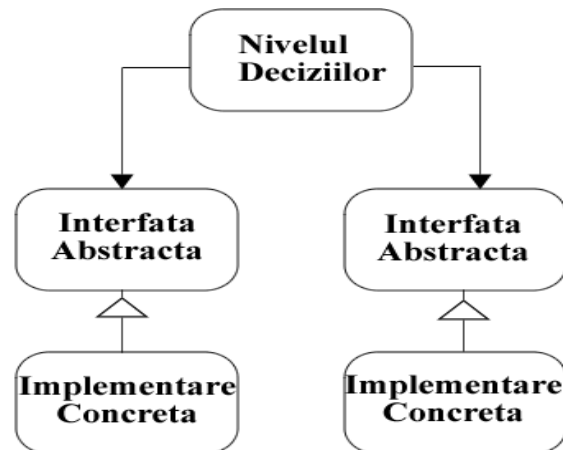


Fig. 5.9. O arhitectură a unui sistem OO

Aceste dezavantaje sunt înlăturate în cadrul unei arhitecturi OO, care respectă principiile OCP, LSP și DIP:

- prin organizarea pe nivele se separă clar nivelul decizional de cel al implementării detaliilor,
- prin inversarea dependențelor, obținându-se o arhitectură alcătuită din piese de software flexibile, mobile și ușor 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 PID, s-a ajuns la câteva rezultate practice. Aplicarea acestora de către proiectanții de software, contribuie la facilitarea muncii lor și asigură obținerea unui design de calitate.

Utilizarea interfețelor (claselor abstracte) pentru a evita legăturile directe între clasele concrete (Fig.5.10)

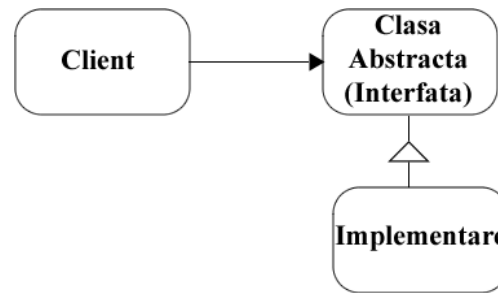


Fig. 5.10. O arhitectură care utilizează interfețele (clase abstracte) pentru a evita legăturile directe între clasele concrete

Utilizarea interfețelor contribuie la obținerea unui **cod stabil** pentru clasa concretă Client. O clasă abstractă este mai puțin probabil să fie modificată, pe de altă parte o clasă abstractă este mai ușor de extins/modificat. Totuși, trebuie avut în vedere că a modifica o abstractizare contravine OCP, care se bazează pe stabilitatea abstracțiunii.

Regulă de bază - Evitarea dependențelor tranzitive prin utilizarea interfețelor!

Concluzii

DIP este sursa multor beneficii ale tehnologiei orientate pe obiect; prin aplicarea lui se obțin module reutilizabile. Este foarte important pentru construcția codului, ca acesta să fie rezistent, robust și elastic la modificări. Deoarece abstractizările și detaliile aplicației sunt izolate unele de altele, codul este mult mai ușor de întreținut (mentenabilitate crescută).

Cheia din principiul inversării dependențelor este utilizarea abstractizării, la fel ca și în OCP. Detaliile de implementare sunt izolate de unele de altele, între ele fiind interpușe 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 și dependența acestora doar de abstractizări contribuie la obținerea unor clase care pot fi utilizate, cu ușurință, în alte aplicații.

Privit din perspectiva principiilor prezentate în secțiunile anterioare, se poate spune că OCP este **scopul**, DIP asigură **mecanismul** de îndeplinire a scopului, iar LSP este modalitatea de **verificare** a mecanismului.

Care este scopul unui design conform OCP ?

Obținerea unor entități software care să fie deschise pentru extindere dar închise la modificări, în acest timp păstrându-se calitățile unui bun design: flexibilitate, mobilitate și reutilizabilitate. DIP specifică modalitatea de atingerea acestui scop: într-o ierarhie de clase, clasa din care se derivează nu cunoaște nici una dintre subclasele sale și modulele care implementează detaliile depind de abstractizări, și nu invers. Iar prin LSP se asigură o măsură a calității moștenirii, verificându-se ca prin moștenire să se obțină subclase care au aceleași proprietăți ca și 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 dependențelor stabile

Așa cum s-a arătat în cadrul subcapitolului 1.2. *Probleme ale software-ului*, **interdependența** reprezintă una dintre cauzele pentru care un design este rigid, imobil și dificil de reutilizat. Totuși, interdependența este necesară dacă modulele implicate în design "colaborează". Ca urmare există tipuri de **dependență utilă** și tipuri de **dependență indesezirabilă**.

În acest paragraf se propune un model de design în care dependențele sunt toate utile și se descrie un set de indicatori pentru măsurarea conformității designului cu modelul propus.

Acești indicatori măsoară stabilitatea software-ului. **Stabilitatea este însăși esența designului software**. Când se proiectează un sistem software, se dorește ca acesta să fie stabil atunci când este modificat. Acesta este și scopul OCP: obținerea unor sisteme stabile. În acest capitol vom analiza modalitatea în care conceptul de stabilitate se răsfrânge asupra relațiilor dintre pachete, în cadrul structurilor aplicațiilor de mari dimensiuni.

Stabilitate și Dependență. Dependențe utile

Se consideră exemplul din *subcapitolul 5.3 (Principiul inversării dependențelor)*, 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 și 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 dependențe, se aplică DIP și se obține o structură (Fig. 5.5) în care modulul de pe nivelul superior depinde de niște abstractizări. Designul astfel obținut, este **robust, mentenabil și ușor de reutilizat**.

Dependențe utile

Totuși, nu toate dependențele au fost îndepărtate, ci doar cele care afectează calitățile programului. Dependențele rămase sunt **nevolatile**, pentru că obiectul/modulul/clasa față de care se manifestă dependența este puțin probabil să se modifice. Probabilitatea ca aceste clase abstracte, *AbstractReader* și *AbstractWriter*, să se modifice este foarte mică, spunem despre ele că au o **volatilitate scăzută**.

Deoarece clasa *Client* depinde de module nevolatile, este puțin predispusă la schimbări. Această situație ilustrează foarte bine principiul deschis – închis: clasa *Client* este **deschisă la extinderi**, deoarece putem crea noi versiuni de clase concrete derivate din *AbstractReader* și *AbstractWriter* pe care *Client* să le acționeze, și este **închisă pentru modificări** deoarece nu trebuie modificată pentru a face aceste extinderi.

Putem afirma în consecință, că o **dependență utilă** este o dependență față de un modul/clasă cu o volatilitate scăzută. Cu cât volatilitatea este mai scăzută, cu atât dependența este "mai bună". O dependență este indezirabilă când se manifestă față de un modul volatil. Cu cât modulul/clasa față de care se manifestă dependența este mai volatil, cu atât dependența 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 conțin 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 și de presiunea pieței și cererile clienților. Un modul este sau nu posibil să fie modificat, dacă conține sau nu ceva ce clientul dorește să schimbe. Acest tip de factori sunt greu de apreciat.

Există, totuși, un factor care influențează volatilitatea și care poate fi măsurat: **stabilitatea**. Stabilitatea, în sensul general acceptat, se definește ca fiind "**greu de modificat**".

Stabilitatea nu este o măsura a probabilității de a modifica un modul, ci a dificultății de a-l modifica.

Deci, modulele care sunt mai greu de modificat, sunt mai puțin volatile.

În exemplul amintit mai sus, clasele abstracte `AbstractReader` și `AbstractWriter` sunt stabile. Caracteristicile care fac aceste clase stabile, sunt:

- sunt **independente**, nu depind de "nimic" și "nimic" nu poate forța o schimbare a lor. *Se numesc clase **independente** acele clase care nu depind de nimic altceva.*
- de aceste clase depind alte clase (`Client`, `Reader` și `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 și clasele derivate. Deci, există motive serioase pentru care nu vom modifica cele două clase, îmbunătățindu-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 și responsabile**, deoarece asemenea clase nu au nici un motiv să se modifice, și au multe motive să nu se modifice.

Principiul dependențelor stabile

Într-un proiect (design), dependența dintre pachete ar trebui să fie în sensul creșterii stabilității pachetelor. Un pachet ar trebui să depindă de pachete mai stabile decât el.

Un design nu poate fi complet static, un anumit grad de volatilitate este necesar dacă se dorește realizarea mentenanței. Acest lucru se obține 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 așteptate și prevăzute.

Nici un pachet, despre care știm că va fi volatil, nu ar trebui să aibă între dependenți pachete greu de modificat. În caz contrar, și pachetul volatil va fi greu de modificat. În conformitate cu principiul dependențelor stabile (The Stable Dependencies Principle, SDP), pachetele care sunt proiectate să fie instabile (ușor de modificat) nu au ca dependenți pachete care au o stabilitate mai mare decât a lor (mai greu de modificat).

Indicatori ai stabilității

O modalitate de a măsura stabilitatea unui pachet este numărarea dependențelor care "intră" și "ies" din pachet. Aceasta ne va ajuta la calcularea **stabilității poziționale** a pachetului.

Se definesc următorii indicatori:

- **Ca : dependențe aferente:** Numărul claselor din afara pachetului care depind de clasele din acest pachet;
- **Ce : dependențe 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 C_a și C_e sunt calculați prin numărarea **claselor** din exteriorul pachetului în cauză care au relații de dependență cu clasele din interiorul pachetului.

Să considerăm următorul exemplu din **Fig. 5.11.**, în care săgețile punctate reprezintă dependențele între pachete. Relațiile dintre clasele pachetelor arată care este natura dependenței, modul cum este implementată aceasta (moștenire, agregare, asociere).

Calcularea stabilității pachetului din centrul diagramei.

Observăm ca există 4 clase exterioare pachetului care sunt în relații cu clasele interioare pachetului, deci $C_a=4$. Mai mult, există 3 clase exterioare pachetului central de care depind clasele interioare, deci $C_e=3$.

$$I = \frac{C_e}{C_a + C_e} = \frac{3}{7}$$

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* și *dependent*.

Când $I=0$, pachetul curent nu depinde de nici un alt pachet, dar de el depind alte pachete. Este *responsabil* și *independent*. Un astfel de pachet are un grad maxim de stabilitate. Din cauza pachetelor dependente, pachetul curent este dificil de modificat, și nu depinde de alte pachete care ar putea forța o modificare a acestuia.

Principiul dependențelor 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 dependenței.

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 situația ideală pentru un sistem de trei pachete.

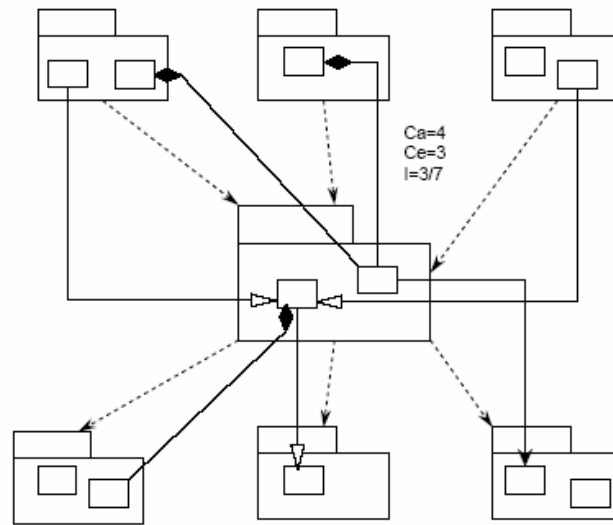


Fig. 5.11. Exemplu de calculare a gradului de stabilitate al unui pachet

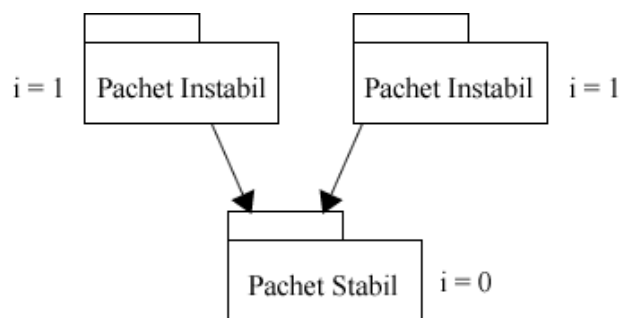


Fig. 5.12. Situația ideală din punct de vedere a stabilității pentru un sistem cu trei pachete

Pachetele care prezintă o probabilitate mai mare de a fi modificate sunt așezate pe un nivel superior, iar cele care sunt stabile sunt așezate la bază. Pentru acest mod de aranjare a pachetelor, orice săgeată direcționată în sus înseamnă o încălcare a principiului dependențelor stabile.

O parte din software-ul unei aplicații nu trebuie să se modifice foarte des, și anume logica de nivel înalt a aplicației, 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 conțină doar acele părți de software, despre care se știe că este probabil să fie modificate.

Dacă logica aplicației este plasată în pachete stabile, atunci codul sursă al acestor pachete va fi dificil de modificat. Acest fapt, ar putea face design-ul inflexibil. OCP oferă soluția pentru a face un pachet să fie flexibil și totuși să aibă un grad maxim de stabilitate ($I=0$),. Conform acestui principiu este posibil și 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 proporțională cu stabilitatea sa.

Principiul abstractizărilor stabile (The Stable Abstractions Principle, SAP) stabilește o **relație între stabilitate și abstractizare** afirmând că un pachet stabil ar trebui să fie și 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 ușurință modificat.

Dacă un pachet se dorește a fi stabil, ar trebui să fie alcătuit din clase abstracte astfel încât să poată fi extins. Pachetele stabile care pot fi extinse sunt și flexibile și nu constrâng design-ul.

Spre deosebire de **principiul inversării dependențelor** care este un principiu pentru **clase**, **principiile dependențelor stabile și 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 și numărul total de clase din pachet.

$$A = \frac{\text{NumarClaseAbstracte}}{\text{NumarTotalClase}}$$

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 conține numai clase abstracte.

Relația dintre stabilitate, I , și gradul de abstractizare, A (Fig. 5.13)

Pentru stabilirea relației dintre stabilitate, măsurată de indicatorul I , și 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 și o abstractizare maximă se găsesc în punctul $(0,1)$. Pachetele cu gradul de instabilitate cel mai mare și 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 și abstractizare, așa încât se admite că există un loc geometric al punctelor care definesc o poziție 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**.

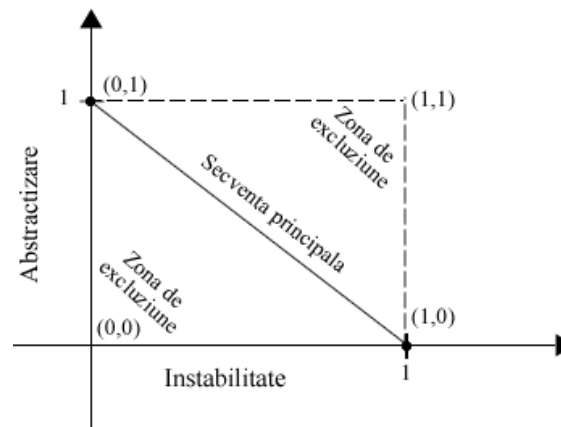


Fig. 5.13. Relația dintre stabilitate, I , și gradul de abstractizare, A . Zonele de excluziune.

Se consideră un pachet în zona determinată de $A=0$ și $I=0$, acesta va fi un **pachet stabil și concret**. Un astfel de pachet nu este de dorit deoarece este **rigid**, nu poate fi extins pentru că nu este abstract și 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 și I=1**. Un astfel de tip de pachet este de asemenea indezirabil (poate chiar imposibil) deoarece are grad maxim de abstractizare și totuși nici un alt pachet nu depinde de el. Și acest tip de pachet este rigid, pentru că abstractizările sunt imposibil de extins. Deci, un pachet în această zonă este fără sens. *Zona din jurul punctului (1,1) este o zonă de excluziune.*

Fie pachetul **din A=0.5 și I=0.5**. Acest pachet este parțial extensibil pentru că este parțial abstract; este parțial stabil deci extinderile nu duc la instabilitate maximă. Un astfel de pachet pare **echilibrat**, deoarece *stabilitatea și abstractizarea se compensează reciproc*. Deci, **zona în care A=I nu este o zonă de excluziune**. Pe linia dintre (0,1) și (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 și nici "prea instabil" pentru abstractizarea sa; are un număr "potrivit" de clase concrete și abstracte, proporțional cu dependențele aferente și eferente. Totuși 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.

Distanța față de Secvența Principală

Considerând că pe dreapta numită Secvența Principală, se găsesc pachetele care au caracteristicile ideale, putem crea un indicator care măsoară distanța pachetului curent față de cazul ideal.

$$D = \left| \frac{A + I - 1}{\sqrt{2}} \right|$$

unde :

- D = Distanța dintre punctul în care se găsește pachetul față de dreapta numită secvența principală;
- A = gradul de abstractizare;
- I = gradul de stabilitate.

Acest indicator are valori cuprinse în intervalul $[0, \sim 0.707]$. (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 și restructurat.

Concluzii

Indicatorii descriși mai sus măsoară conformitatea unui design față de un model de dependențe și abstractizări. Acești indicatori încearcă să măsoare calitatea unui design din anumite puncte de vedere, fără a avea pretenția de adevăr absolut. În funcție de dorințe de la un design, se pot defini și folosi diverși indicatori.

6

Motto:

*“Șabloanele de proiectare software te ajuta să
înveți mai mult din succesele altora, decât din
propriile eșecuri”*

[Mark Johnson]

Șabloane software de proiectare

6.1. Elementele unui șablon de proiectare software

În ingineria software, prin șablon de proiectare (în engleză “design pattern”, prescurtat-DP) se înțelege o soluție, un tipar care se aplică la un anumit tip de probleme. Un design pattern este o descriere sau un model pentru o soluție a problemei. Un șablon nu poate fi transformat direct în cod sursă. Șabloanele de proiectare în OOD (Object Oriented Design) arată relațiile și interacțiunile dintre clase sau obiecte, fără a detalia clasele și obiectele care sunt implicate.

Scopul proiectării cu șabloane este acela de a face un bun design orientat pe obiecte, și mai mult de atât, un **design reutilizabil** (ceea ce este mai important decât reutilizarea codului, dar adesea reutilizarea designului implică și reutilizarea codului).

Calitatea unui design software este direct proporțională cu experiența și cunoștințele anterioare ale celui care îl realizează. Un expert reutilizează soluții pe care le-a găsit în trecut și care deja și-au dovedit eficiența în probleme asemănătoare. Când un expert găsește o soluție pe care o consideră bună, o

refolosește. Șabloanele de proiectare rezolvă probleme specifice de design, făcând proiectul OO mai flexibil, elegant și reutilizabil. „Design patterns” ajută designerii de software propunând niște șabloane bazate pe experiența anterioară. Un designer care este familiarizat cu DP, le poate aplica imediat, pentru rezolvarea unor probleme specifice, fără a fi nevoit să le redescopere.

Șabloanele software sunt structuri care respectă principiilor de proiectare, deoarece sunt obținute ca urmare a aplicării acestor principii (șabloanele sunt rezultate directe ale principiilor). Cu alte cuvinte, prin utilizarea șabloanelor software în cursul proiectării unei arhitecturi OO, se obține garanția că sistemul respectă principiile prezentate în *capitolul 5* al acestei lucrări, având și calitățile unui ”bun design”: reutilizabil, flexibil și robust.

Lucrarea de referință pentru șabloanele 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 soluții recurente la probleme comune în designul software. În cartea GoF sunt propuse 23 de șabloane de proiectare într-o formă ușor de reutilizat; aceste șabloane încorporează experiența autorilor în rezolvarea problemelor de OOD.

Șabloanele din aceasta carte au scopul de a descrie obiectele, clasele și modul în care acestea comunică, scopul urmărit fiind rezolvarea unei probleme generală de design într-un context particular.

Un șablon are patru elemente esențiale:

1. **Nume:** se dorește ca numele să descrie în câteva cuvinte, o problemă de design pattern, soluția și efectele ei.
2. **Problema:** descrie situațiile în care se aplică șablonul. Explică atât problema cât și contextul acesteia.
3. **Soluția:** descrie elementele care fac parte din design, relațiile dintre ele, responsabilitățile fiecărui element și colaborările dintre acestea. Soluția nu descrie un proiect concret specific unei situații sau o implementare concretă, deoarece șablonul are un caracter general, referindu-se la o diversitate de contexte. DP furnizează o descriere abstractă a unei probleme și un

aranjament general al elementelor componente (clase și obiecte) care rezolvă problema descrisă.

4. **Efecte:** se referă la rezultatele aplicării șablonului. Acestea sunt foarte importante atunci când evaluăm alternativele și pentru a estima costurile și beneficiile aplicării șablonului. Efectele unui șablon se resimt și în flexibilitatea, extensibilitatea și portabilitatea unui sistem.

Șablonul identifică clasele și instanțele participante, rolul acestora, colaborarea și distribuirea responsabilităților între ele.

Tabel 6.1. Șabloanele de proiectare GoF

		SCOP (Ce face șablonul)		
Cui se aplică șablonul ?		Creățional	Structural	Comportamental
	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 șabloane :

Șabloanele creaționale

Abstract Factory – furnizează o interfață pentru crearea familiilor de obiecte dependente sau relaționate, fără a specifica clasele lor concrete.

Builder – separă construcția unui obiect complex de reprezentările sale astfel încât procese de construcție similare pot crea reprezentări diferite.

Factory Method – definește o interfață pentru crearea unui obiect, dar lasă subclasele să decidă care clasă va fi instanțiată.

Prototype – specifică tipurile de obiecte create folosind o interfață prototipizată și crează noi obiecte copiind acel prototip.

Singleton – asigură că o clasă are numai o instanță, dar furnizează un punct global pentru a o accesa.

Șabloanele structurale

Adapter – face conversia interfeței unei clase la interfața dorită de client. *Adapter* permite claselor să lucreze împreună, ceea ce altfel n-ar fi posibil din cauza incompatibilității interfețelor.

Bridge – decuplează o formă abstractă de implementare astfel încât cele două pot fi schimbate independent.

Composite – compune obiecte din arborele de structură care reprezintă părți sau întreaga ierarhie. *Composite* lasă clienții să trateze fiecare în parte și uniform obiectele și compunerea lor.

Decorator – atașează în mod dinamic responsabilități adiționale obiectelor. Furnizează subclaselor o alternativă flexibilă pentru extinderea funcționalităților.

Facade – furnizează interfață unificată cu un set de interfețe într-un subsistem. Definește o interfață de nivel înalt care determină ca subsistemul să fie mai ușor de utilizat.

Flyweight – folosește tehnica "sharing" (divizare în mod egal) pentru a lucra cu un număr mare de obiecte mici în mod eficient.

Proxy – furnizează un înlocuitor pentru alt obiect în scopul de a controla accesul la el.

Șabloane de comportament

Chain of Responsibility - permite cuplarea expeditorului unei cereri cu destinatarul ei dând mai mult decât unui singur obiect șansa de a utiliza acea cerere. Înlănțuie obiecte destinate și lasă cererea să treacă de-a lungul șirului atât timp cât un obiect o utilizează.

Command – încapsulează cererea ca un obiect lasând prin aceasta clienți parametrizați cu diferite cereri, coadă sau tabel de cereri și suportă operații care se pot detașa.

Interpreter – dându-se un limbaj, definind o reprezentare pentru gramatica sa, interpretează propoziții din acel limbaj.

Iterator – furnizează o cale de acces la elementele unui agregat de obiecte secvențiale fără a expune reprezentarea sa internă.

Mediator – definește un obiect care încapsulează toată mulțimea de obiecte cu care interacționează. *Mediator* promovează cuplajul slab păstrând obiectele pentru a putea fi referite în mod explicit și lasă să poată fi schimbată independent interacțiunea lor.

Memento – fără a fi violată încapsularea, capturează și externează starea internă a unui obiect, astfel încât, obiectul poate fi restaurat mai târziu în starea sa.

Observer - definește o dependență „unu-multi” între obiecte, astfel încât când un obiect își schimbă starea toate obiectele din dependența sa sunt actualizate în mod automat.

State – permite unui obiect să-și modifice comportamentul atunci când se schimbă starea sa internă. Obiectul va părea că-și schimbă clasa.

Strategy – definește o familie de algoritmi, încapsulați fiecare și-i face interșanjabili. *Strategy* lasă algoritmul să poată fi schimbat în mod independent de clienții care îl utilizează.

Template Method – lasă subclasele să-și redefinească câțiva pași din algoritm fără a modifica structura algoritmului.

Visitor – permite să definești o nouă operație 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 și alte șabloane mai mult sau mai puțin cunoscute, unele noi (tratând noi tipuri de probleme, apărute ca urmare a dezvoltării limbajelor OO), altele sunt variații ale șabloanelor amintite mai sus (aplicații ale șabloanelor GoF pe cazuri particulare).

6.2. Cum rezolvă șabloanele problemele de proiectare

Șabloanele de proiectare rezolvă o serie întregă de probleme curente cu care se confruntă proiectanții de sisteme orientate pe obiecte, în diferite moduri. Programele orientate pe obiecte sunt constituite, evident din obiecte. Un obiect „împachetează”, assemblează la un loc, atât datele cât și procedurile care operează cu acele date. Procedurile se numesc *metode* sau *operații*. Un obiect execută o operație în momentul în care primește o *cerere* (sau *un mesaj*) de la un client.

Cererile constituie singura modalitate de a determina un obiect să execute o operație. Operațiile constituie singura cale de a schimba datele interne ale obiectului. Din cauza acestor restricții 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 luați în considerație mulți factori caum ar fi: încapsulare, granularitate, dependență, flexibilitate, performanță, evoluție, reutilizare, și mulți alții. Ei tot influențează descompunerea și câteodată sunt în contradicție.

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*, șablonul 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 și mâine! Abstractizările care decurg din proiectare sunt cheia realizării unui proiect flexibil.

Șabloanele de proiectare ajută la identificarea celor mai puțin evidente abstractizări și 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. Șablonul *Strategy* () descrie cum se implementează familii interșanjabile de algoritmi. Șablonul *State* () reprezintă fiecare stare a unei entități 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 și dimensiune. Se poate reprezenta orice ca obiect, plecând de la hardware și până la întreaga aplicație. Cum decidem care poate fi un obiect? Șabloanele se adresează exact acestei cerințe. Șablonul *Facade* descrie cum putem reprezenta subsisteme complete ca obiecte, *Flyweight* descrie cum vor fi suportate un număr uriaș de obiecte de o granularitate mai fină. Alte șabloane descriu căi specifice de descompunere a unui obiect în altele mai mici.

6.3. Cum se selectează un șablon

Atunci când sunt mai multe șabloane la dispoziție 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 selecție a celui mai potrivit șablon pentru o anumită problemă.

- Se trec în revistă șabloanele de proiectare care sunt disponibile, se studiază ce oferă fiecare și se alege acelea care se potrivesc cel mai bine problemei care trebuie proiectată.
- Se studiază relațiile dintre șabloane (fig. 6.1) și se alege grupul cel mai potrivit.
- Se studiază scopurile, asemanările și deosebirile dintre șabloane.

- Se parcurge lista (Tabel 6.2.) cu acele aspecte ale șabloanelor care pot fi modificate în mod independent, modificări care corespund scopului proiectului propus.

Dacă proiectantul nu are experiență în proiectarea orientată pe obiect se poate începe cu cele mai simple și comune șabloane de proiectare: Abstract Factory, Factory Method, Adapter, Observer, Composite, Strategy, Decorator, Template Method.

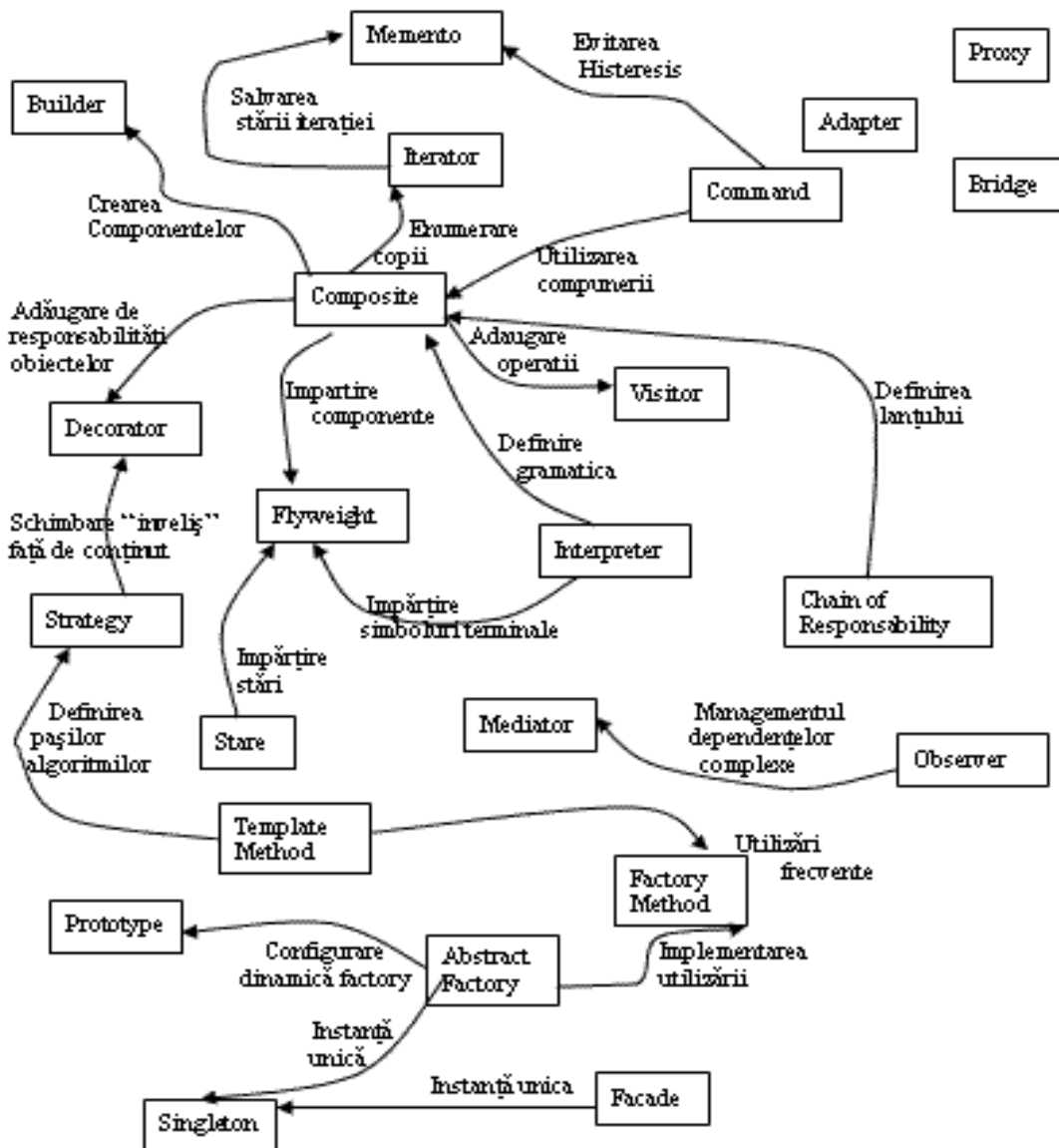


Fig. 6.1. Relațiile între șabloanele de proiectare

6.4. Cum se folosește un șablon de proiectare

O abordare pas cu pas a proiectării unui sistem software folosind șabloane ar putea să decurgă în felul următor:

1. *Se citește descrierea șablonului pentru a avea o imagine generală. Se acordă o atenție deosebită secțiunilor de aplicabilitate și consecințe pentru a avea garanția că șablonul de proiectare corespunde problemei;*
2. *Se revine și se studiază secțiunile de structură, participanți și colaborări. Trebuie să se înțeleagă exact clasele și obiectele din șablon și modul în care acestea relaționează unele cu altele.*
3. *Se examinează apoi secțiunea de cod care prezintă un exemplu concret. Studiind codul se poate învăța cum se implementează șablonul.*
4. *Se alege numele participanților din șablon, nume care trebuie să aibă o semnificație în contextul aplicației. Numele participanților din design patterns sunt de obicei prea abstracte pentru a apărea direct în aplicație și de aceea este util să se încorporeze numele participanților în numele care apar în aplicație 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 interfețele, se stabilesc relațiile de moștenire și se definesc variabilele care reprezintă datele și obiectele referite. Se identifică clasele existente din aplicație pe care șablonul le va afecta și se pun de acord.*
6. *Se definesc numele specifice aplicației pentru operațiile din șablon. Încă o dată, numele depind în general de aplicație. Se folosesc responsabilitățile și colaborările asociate cu fiecare operație ca un ghid. De asemenea, este bine să existe consecvență în convențiile de notare. De exemplu, se folosește prefixul "Creează-", de fiecare dată când desemnăm o metodă *Factory*.*

7. *Se implementează operațiile având grijă de responsabilitățile și colaborările din șablon.* Secțiunea de implementare oferă sugestii pentru implementare.
8. *Se definesc clasele.* Se declară interfețele, se stabilesc relațiile de moștenire și se definesc variabilele care reprezintă datele și obiectele referite. Se identifică clasele existente din aplicație pe care șablonul le va afecta și se pun de acord.
9. *Se definesc clasele.* Se declară interfețele, se stabilesc relațiile de moștenire și se definesc variabilele care reprezintă datele și obiectele referite. Se identifică clasele existente din aplicație pe care șablonul le va afecta și se pun de acord.
10. *Se definesc numele specifice aplicației pentru operațiile din șablon.* Încă o dată, numele depind în general de aplicație. Se folosesc responsabilitățile și colaborările asociate cu fiecare operație ca un ghid. De asemenea, este bine să existe consecvență în convențiile de notare. De exemplu, se folosește prefixul "Creează-", de fiecare dată când desemnăm o metodă Factory.
11. *Se implementează operațiile având grijă de responsabilitățile și colaborările din șablon.* Secțiunea de implementare oferă sugestii pentru implementare.

Tablelul 6.2. Aspectele de proiectare care pot fi modificate în design patterns

Scop	Șablon de proiectare	Aspecte care pot fi modificate
Creațional	Abstract Factory	Familiile de obiecte produs
	Builder	Modalitatea în care pot fi create obiectele compozite
	Factory Method	Subclasa obiectelor care sunt instanțiate
	Prototype	Clasa obiectelor care este instanțiată
	Singleton	O singură instanță a unei clase
Structural	Adapter	Interfața cu un obiect
	Bridge	Implementarea unui obiect
	Composite	Structura și compoziția unui obiect
	Decorator	Responsabilitățile proprii ale unui obiect fără subclase
	Facade	Interfața cu un subsistem
	Flyweight	Memorarea costurilor obiectelor
	Proxy	Modul în care este accesat obiectul- locația sa
Comportamental	Chain of Responsibility	Obiectul care poate îndeplini o solicitare
	Command	Când și cum poate fi îndeplinită o solicitare
	Interpreter	Gramatica și interpretarea unui limbaj
	Iterator	Cum sunt accesate și parcurse elementele unei mulțimi
	Mediator	Cum și care obiecte interacționează unele cu altele
	Memento	Ce informație privată este memorată în afara obiectului și când.
	Observer	Numărul obiectelor care depind de alt obiect; Cum poate fi actualizat obiectul dependent.
	State	Stările unui obiect
	Strategy	Un algoritm
	Template Method	Pașii unui algoritm
	Visitor	Operațiile care pot fi aplicate obiectului (obiectelor) fără schimbarea clasei (claselor).

7

Proiectarea sistemelor software

7.1. Procesul de proiectare

Proiectarea software-ului implică următoarele stadii:

(1) Studiarea și înțelegerea problemei. Fără o înțelegere profundă și corectă a sistemului proiectarea efectivă a software-ului ar fi practic imposibilă. Problema trebuie examinată din unghiuri diferite, astfel încât să permită o privire din interior a cerințelor de proiectare.

(2) Identificarea caracteristicilor principale și, în cele din urmă, o posibilă soluție. Adesea este util să se identifice mai multe soluții și să se evalueze fiecare dintre ele. Alegerea soluției depinde de experiența proiectantului, de componentele reutilizabile pe care le are la dispoziție, de simplitatea soluțiilor derivate.

(3) Descrierea fiecărei abstractizări utilizate în soluție. Înainte de a crea documentația formală, totuși proiectantul trebuie să stabilească dacă este necesar să construiască o descriere informală a proiectului. Erorile și 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 entitățile de proiectare, cum ar fi procesele, funcțiile sau tipurile, iar legăturile reprezintă relațiile existente între entități. Scopul

procesului de proiectare este de a crea un asemenea graf fără inconsistențe și în care toate relațiile dintre entități sunt legale (posibile).

Procesul de proiectare implică utilizarea formalismului ca proiectare progresivă, cu un “backtraking” constant pentru a corecta mai repede și mai puțin formal, proiectele. Proiectantul începe cu o imagine foarte neformală a proiectului și o rafinează, adăugându-i informație pentru a realiza un proiect mai bine formalizat (Fig.7.1).

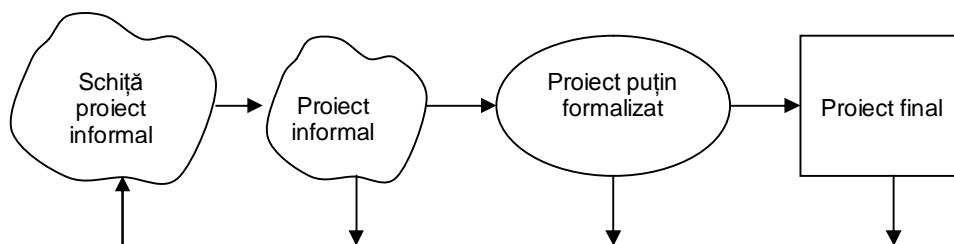


Fig.71. Procesul de proiectare

Procesul de proiectare implică descrierea sistemului într-un număr diferit de niveluri de abstractizare, permițând astfel descoperirea mai devreme a omisiunilor sau a erorilor. Această reacție creează posibilitatea ca diferitele stadii de proiectare să fie rafinate.

Fig.7.2. prezintă stadiile procesului de proiectare și al descrierilor de proiectare produse ca rezultat al acestor activități.

Ieșirea fiecărei activități de proiectare este o specificație. Această specificație poate fi abstractă, o specificare formală care este produsă pentru a clarifica cerințele, sau poate fi o specificare a modului în care o parte din sistem va fi realizată.

Astfel, procesului de proiectare continuă, adăugându-se din ce în ce mai multe detalii la specificare. Ultimile ieșiri sunt specificațiile algoritmilor și ale structurilor de date care vor constitui baza pentru implementarea sistemului.

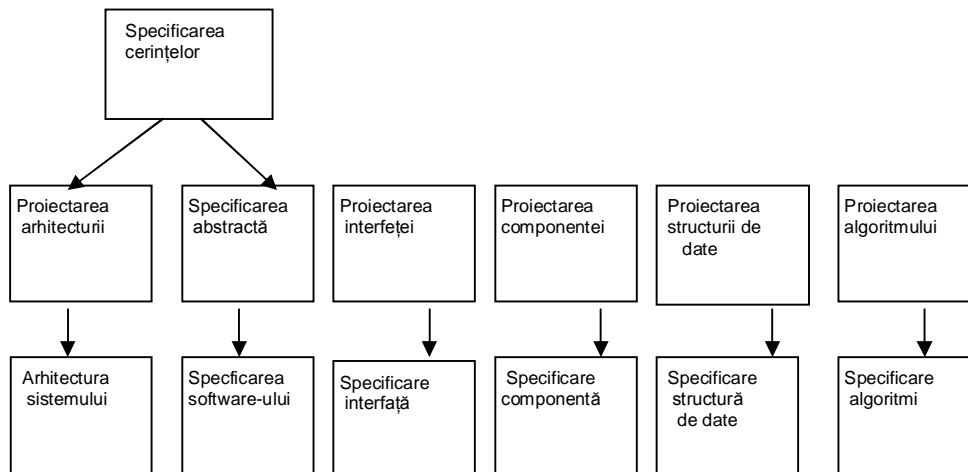


Fig.7.2. Activități de proiectare și produse proiectate

Fig. 7.2 sugerează că stadiile procesului de proiectare sunt secvențiale. În realitate, activitatea de proiectare se desfășoară în paralel cu alte produse de proiectare, aflate în faze de detaliere diferite ale procesului de proiectare.

Activitățile prezentate în fig. 7.2 sunt esențiale în proiectarea sistemelor software mari și cuprind următoarele:

(1). *Proiectarea arhitecturii.* Se realizează subsistemele întregului sistem identificându-se și documentându-se relațiile dintre ele.

(2). *Specificarea abstractă.* Se generează specificări abstracte ale serviciilor pentru fiecare subsistem în parte și se stabilesc restricțiile sub care va opera sistemul.

(4). *Proiectarea componentei.* Serviciile produse de subsistem sunt partiționate în funcție de componentele din acel subsistem.

(5). *Proiectarea structurilor de date.* Structurile de date folosite în implementarea sistemului vor fi proiectate în detaliu și specificate.

(6). *Proiectarea algoritmului.* Algoritmii utilizați pentru a furniza servicii vor fi proiectați în detaliu și specificați.

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 funcții.

O abordare recomandată pentru proiectarea pe scară largă este proiectarea top-down în care problema este recursiv împărțită 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 încrucișare observându-se în graf pe nivelurile scăzute ale arborelui de proiectare, astfel încât proiectanții pot identifica posibilitățile de reutilizare.

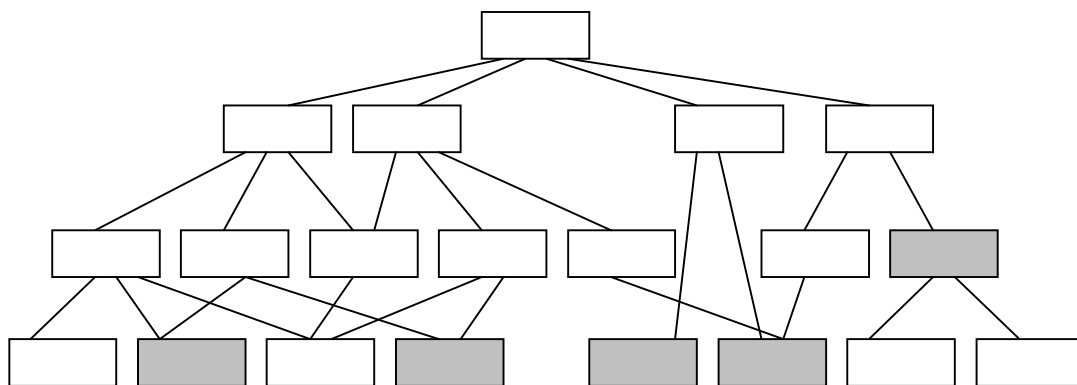


Fig.7. 3. Structura proiectării software

De fapt nu este recomandabil să se proiecteze sistemele într-o manieră care este strict top-down. Proiectanții își utilizează adesea experiența anterioară de proiectare și nu au nevoie întotdeauna să descompună toate abstractizările, știind exact care parte a sistemului poate fi construită.

Proiectarea top-down a fost propusă în conjuncție cu descompunerea funcțională și este o abordare validă în care componentele proiectate sunt strâns legate. Totuși când este adoptată o proiectare orientată pe obiect și multe obiecte existente urmează să fie reutilizate, proiectarea top-down nu este cea mai indicată.

Proiectantul utilizează obiectele existente într-o rețea de proiectare și construiește 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 in care se construiește **solutia problemei**. Rezultatul este un **model fizic** al viitorului sistem, care este descris in **Documentul de Proiectare Arhitecturala**.

Cerintele din documentul Cerintelor 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 cerintelor. Documentul de proiectare arhitecturala trebuie sa specifice rolul fiecărei componente, cerintele care i-au fost alocate, interfata de comunicare cu celelalte componente ale sistemului. De asemenea, in etapa de proiectare arhitecturala se intocmeste **Planul Testelor de Integrare**.

7.2.1. Procesul: obținerea 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 oprește 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 arhitecturală

- 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 relatiile ierarhice se bazeaza pe generalizare si specializare
 - diagrame de structura (Fig.7.4), in cazul unei descompuneri functionale : descompunerea iterativa a functiilor pe care trebuie sa le implementeze sistemul

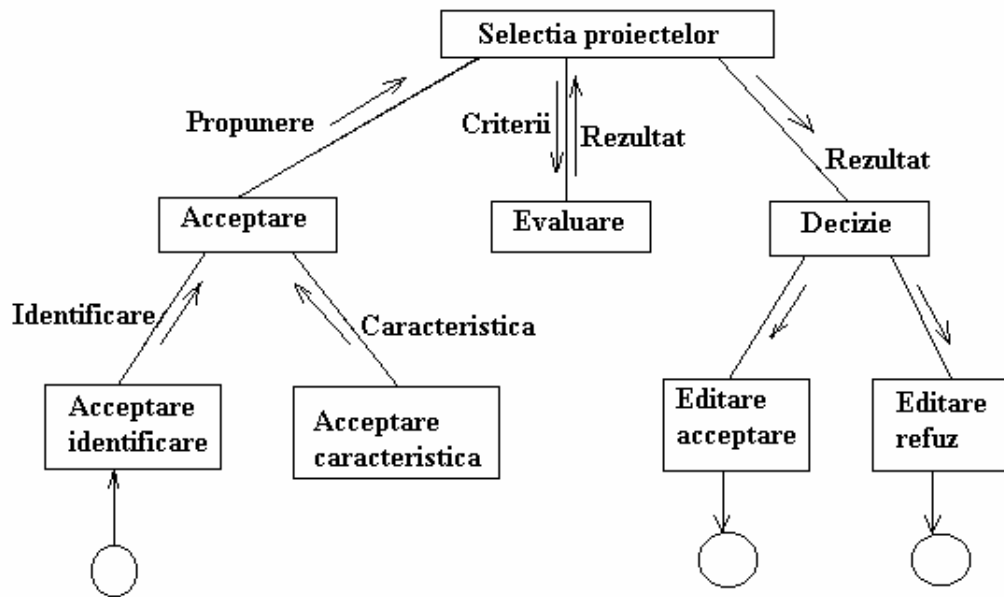


Fig.7.4. Diagrama de structura

- 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
- Componentele 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 precizează ordinea în care vor fi integrate modulele în subsisteme și apoi subsistemele până la nivel de sistem precum și testele care vor fi efectuate la integrare.

7.3. Proiectarea calității

Nu există nici în momentul de față o modalitate absolut garantată care să ateste că un proiect este fără nici o eroare. În funcție de domeniul de aplicație și de cerințele 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 mentenanța cea mai bună.

Un proiect mentenabil poate fi cu rapiditate adaptat pentru a i se adăuga noi funcționalități 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 părțile componente să fie legate logic între ele. Ele trebuie să poată fi cu ușurință decuplate, cuplajul fiind o măsură a independenței componentelor.

Multe preocupări și cercetări științifice au avut în vedere stabilirea unor metrici de proiectare a calității, care să ateste în final că un proiect este bun. Cele mai multe asemenea metrici au fost dezvoltate în conjuncție cu metodele structurate ale lui Yourdan. Caracteristicile de calitate sunt în mod egal aplicabile atât proiectării orientate pe obiect cât și proiectării orientate funcțional, și sunt : coeziunea, cuplajul, inteligibilitatea, adaptabilitatea.

7.3.1. Coeziunea

Coeziunea unei componente este o măsură a cât de bine se potrivește ea logic cu celelalte. O componentă poate implementa o singură funcție logică sau o singură entitate logică și toate părțile componente trebuie să contribuie la această implementare. Dacă componenta include părți care nu sunt direct legate de funcția ei logică (de exemplu, dacă există un grup de operații care se execută

în același timp și nu au legătură cu funcția), aceasta înseamnă că gradul de coeziune este scăzut.

Constantine și Yourdan (1979) au identificat șapte niveluri de coeziune în ordine crescătoare a gradului, și anume:

1). *Coeziune de coincidență*. Părțile componente nu sunt într-o relație, ci pur și simplu sunt asamblate într-o singură componentă.

2). *Asociere logică*. Componentele care realizează funcții similare, cum ar fi operații 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 singură secvență de control.

5). *Coeziune de comunicare*. Toate elementele unei componente operează pe aceleași date de intrare sau produc aceleași date de ieșire.

6). *Coeziune secvențială*. Ieșirea dintr-un element al componentei servește ca intrare pentru alt element.

7). *Coeziune funcțională*. Fiecare parte a componentei este necesară pentru execuția unei singure funcții.

Aceste clase de coeziune nu sunt în mod strict definite, iar Constantine și Yourdan le-au ilustrat prin exemple. Nu este întotdeauna ușor să se facă o clasificare a unui sistem într-o anumită categorie de coeziune.

Metoda lui Yourdan este aplicabilă și este evident că cea mai bună formă de coeziune a unei unități este funcția. Totuși, cel mai mare grad de coeziune îl au sistemele orientate pe obiecte și acest lucru reprezintă de fapt o caracteristică a acestora. Într-adevăr, unul dintre principalele avantaje ale acestei modalități 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 și în care sunt incluse toate operațiile pentru acea entitate. În acest context se poate defini următoarea clasă de coeziune:

Coeziune de obiect. Fiecare operație are ca rezultat acea funcționalitate 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 soluția problemei. Dacă este necesar să se modifice sistemul, această parte există într-un singur loc și orice trebuie făcut cu ea se află încapsulat într-o singură unitate.

Dacă funcționalitatea este furnizată de un sistem-obiect care utilizează moștenirea din super-clase, coeziunea obiectului care moștenește atribute și operații 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 funcționalitatea unui obiect nu este în totalitate înțeleasă.

7.3.2. Cuplajul

Cuplajul este legat de coeziune și este un indicator al tăriei intercorelării dintre unitățile programului.

Sistemele cu cuplaj ridicat au interconectări puternice cu unitățile din program care depind unele de altele. Sistemele cu cuplaj slab sunt construite din unități independente sau aproape independente.

Ca reguli generale: toate modulele au cuplaj ridicat dacă ele utilizează variabile în comun sau dacă ele își schimbă informația de control. Constantine și Yourdan au numit această *cuplare comună* sau *cuplare prin control*. Cuplajul slab este obținut prin asigurarea ca, atât timp cât este posibil, informația reprezentativă este păstrată în interiorul unei componente, iar interfața de date cu alte unități 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 numește 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.

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 partiționată cu nici un alt obiect, iar un obiect poate fi la rândul lui înlocuit de altul, cu aceeași interfață.

Moștenirea în sistemele orientate pe obiect are o formă definită de cuplaj. Obiectele care moștenesc atribute și operații sunt cuplate cu super-clasele lor, iar schimbările făcute în superclase trebuie operate cu grijă, deoarece acestea se propagă în toate clasele care moștenesc acea super-clasă.

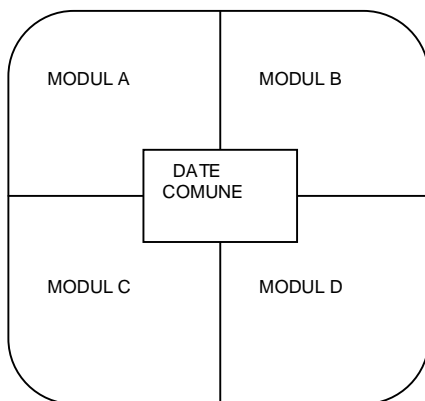


Fig.7.5. Cuplaj puternic

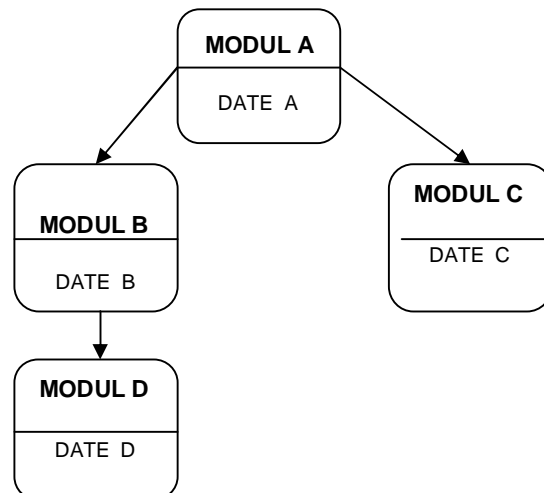


Fig.7.6 Cuplaj slab

7.3.3. Inteligibilitatea

Schimbarea componentei unui proiect are drept implicație ca persoana responsabilă de acea schimbare să înțeleagă operația făcută. Acest lucru se referă la câteva caracteristici ale componentei și anume:

(1) *Coeziunea*. Poate acea componenta să fie înțeleasă fără alte referiri la celelalte componente?

(2) *Numele*. Sunt numele utilizate în înțelesul componentei?

(3) *Documentare*. Este componenta documentată astfel încât să reflecte o legătură clară între entitățile din lumea reală și componentă?

(4) *Complexitate*. Cât de mare este complexitatea algoritmilor utilizați în implementarea componentei? În acest context se utilizează complexitatea într-o manieră neformală.

Complexitatea înaltă implică multe relații între diferite componente ale proiectului și o structură logică complexă, care poate implica secvențe “if-then-else” imbricate.

Complexitatea componentelor este greu de înțeles și 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, obținându-se astfel o măsură a inteligibilității componentei. Complexitatea afectează înțelegerea, dar sunt și alți factori care o afectează, cum ar fi organizarea datelor și stilul în care proiectul este descris.

Măsurile complexității pot numai să furnizeze un indicator al inteligibilității componentei. Moștenirea în proiectele orientate pe obiect afectează înțelegerea.

Moștenirea este folosită pentru a ascunde detalii de proiectare, iar proiectul este ușor de înțeles.

Pe de altă parte, utilizarea moștenirii solicită celui care citește proiectul să privească la multe clase diferite de obiecte din ierarhia de moștenire, iar înțelegerea proiectului este redusă.

7.3.4. Adaptabilitatea

Pentru ca un proiect să fie bine întreținut, el trebuie să poată fi cu ușurință adaptat diverselor cerințe survenite ulterior. Aceasta implică, subînțeles, ca toate componentele să fie cuplate slab. Mai mult decât atât, adaptabilitatea înseamnă ca proiectul să fie bine documentat, documentația componentelor să poată fi înțeleasă cu ușurință în concordanță cu implementarea, iar implementarea să fie scrisă într-o formă accesibilă.

Un proiect adaptabil trebuie să aibă un nivel înalt de vizibilitate și trebuie să existe o relație clară între diferitele niveluri ale proiectului. Trebuie să fie posibil pentru cititorul proiectului să găsească relația dintre reprezentarea ca o diagramă de structuri și reprezentarea transformării datelor (Fig.7.7).

De asemenea, trebuie să fie ușor să se încorporeze schimbările făcute din proiect în toate documentele proiectului. Pentru o adaptabilitate optimă componenta trebuie să se conțină numai pe sine.

Adaptabilitatea unei componente poate să implice schimbări numai ale unei părți a componentei care se referă la funcțiile externe, astfel ca specificarea acestor funcții externe să fie de asemenea luată în considerație de către cel care face modificarea.

Unul dintre principalele avantaje ale moștenirii sistemelor orientate pe obiect este acela că în acest caz componentele pot fi adaptate cu ușurință.

Mecanismul de adaptare nu se referă la modificarea componentei, ci la crearea unei noi componente care moștenește atributele și operațiile componentei originale.

Numai atributele și operațiile care trebuie schimbate sunt modificate. Această adaptabilitate simplă este motivul pentru care limbajele orientate pe obiect sunt atât de eficiente pentru prototipizarea rapidă.

Totuși, pentru sistemele cu viață scurtă, trebuie avut în vedere că moștenirea devine o problemă atunci când sunt operate din ce în ce mai multe schimbări și rețeaua de moștenire devine din ce în ce mai complexă.

Funcționalitatea este adesea distribuită în diverse puncte ale rețelei, iar componentele sunt în acest caz greu de înțeles.

Experiența în programarea orientată pe obiect a arătat că rețeaua de moștenire trebuie periodic revizuită și restructurată pentru a se reduce complexitatea și duplicarea funcționalității. În mod clar, aceasta face să crească costul schimbării sistemului.

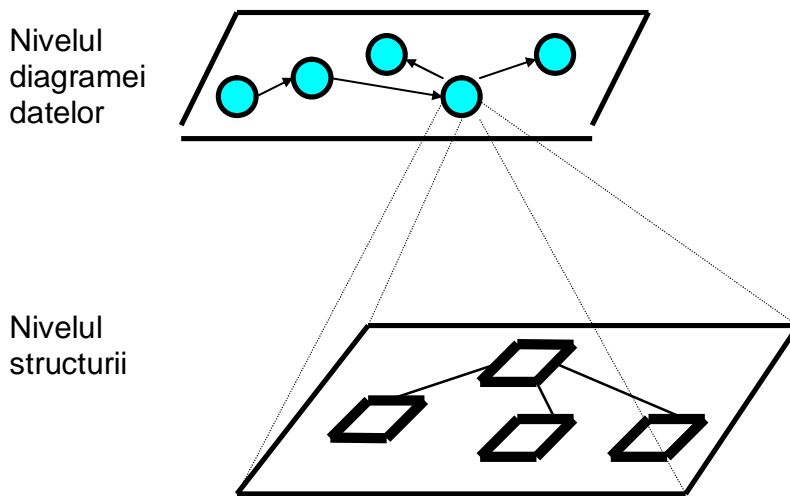


Fig.7.7 Diagrama de structură și reprezentare a datelor

7.4. Modularizarea proiectului

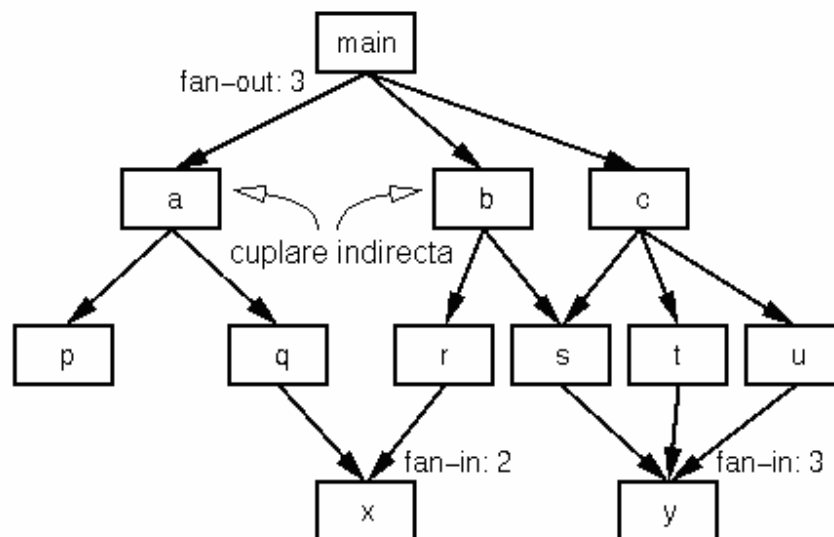
7.4.1. Principii de modularizare:

- Modulele trebuie să fie simple și cât mai independente unul de altul:
 - O modificare a unui modul are influență minimă asupra altor componente
 - O schimbare mică a cerințelor nu conduce la modificări majore ale arhitecturii software (gruparea cerințelor corelate în același modul)
 - Efectul unei condiții de eroare este izolat în modulul care a generat-o
 - Un modul poate fi înțeles ca o entitate de sine-stătătoare
- Modulele trebuie să "ascundă" modul de implementare a funcțiilor descrise de interfața lor, de ex. cum sunt memorate datele cu care lucrează (tablou, listă, arbore, în memorie sau într-un fișier)

7.4.2. Reguli de proiectare a modulelor:

- **Minimizarea cuplării** între 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 fiecărei componente: elementele grupate într-o componentă trebuie să fie corelate, de ex. să contribuie la aceeași prelucrare
 - **Restrângerea numărului de module apelate (fan-out)** de un modul:
 - **Maximizarea numărului de module care utilizează un modul (fan-in)** – încurajază reutilizarea (Fig. 7.8)
- "Fan-in" mare: un număr mare de module depind de el
 „Fan-out“ mare: modulul depinde de multe module
- **Factorizare:** funcționalitățile comune sunt definite în module reutilizabile.



Diagarama de structura

Fig.7.8. Exemplu de diagramă de structură

Documentul de proiectare arhitecturala (ADD)

Sablonul 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 efectueaza la nivelul modulelor definite in proiectarea arhitecturala.
- Poate avea loc in paralel, pentru diferite module.
- Detaliaza modelul de proiectare arhitecturala:
 - pot fi definite module de nivel mai coborat
 - se detaliza componenta claselor: attributele si functiile membre
 - se aleg biblioteci utilizate in implementare
 - se incurajeaza reutilizarea
 - sunt descrisi algoritmi

8

Testarea sistemelor software

8.1. Introducere

Un test constă în execuția programului pentru un set de date de intrare convenabil ales, pentru a verifica dacă rezultatul obținut este corect.

Un caz de test este un set de date de intrare împreună cu datele de ieșire pe care programul ar trebui să le producă. De exemplu, valorile coeficienților a, b, c ai unei ecuații de gradul doi împreună cu valorile x_1, x_2 , în testul unui program de rezolvare a ecuațiilor de gradul doi. Cazurile de test se aleg astfel încât să fie puse în evidență, dacă este posibil, situațiile de funcționare necorespunzătoare.

Testarea este activitatea de concepție a cazurilor de test, de execuție a testelor și de evaluare a rezultatelor testelor, în diferite etape ale ciclului de viață al programelor. Tehnicile de testare sunt mult mai costisitoare decât metodele statice (inspectările și demonstrările de corectitudine), de aceea în prezent se apreciază că tehnicile statice pot fi folosite pentru a le completa pe cele dinamice, obținându-se astfel o reducere a costului total al procesului de verificare și validare. Metodele tradiționale de verificare/validare presupun realizarea verificării/validării prin inspectări și teste. Aceste activități ocupă circa 30-50% din efortul total de dezvoltare, în funcție de natura aplicației.

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 execuții diferite ale programului. Testarea poate doar să demonstreze prezența erorilor într-un program. Într-un sens, testarea este un proces distructiv, deoarece urmărește să determine o comportare a programului neintenționată de proiectanți 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 cunoașterea structurii programului și a codului poate fi foarte utilă pentru alegerea unor date de test relevante.

Testarea în vederea verificării programului folosește date de test alese de participanții la procesul de dezvoltare a programului. Ea este efectuată la mai multe nivele: la nivel de unitate funcțională, de modul, de subsistem, de sistem.

Testarea în vederea validării programului, numită și **testare de acceptare**, are drept scop stabilirea faptului că programul satisface cerințele viitorilor utilizatori. Ea se efectuează în mediul în care urmează să funcționeze programul, deci folosindu-se *date reale*. Prin testarea de acceptare pot fi descoperite și erori, deci se efectuează și o verificare a programului.

8.2. Testarea pe parcursul ciclului de viață al unui program.

8.2.1. Testele "unitare"

Testarea unui modul (o funcție, o clasă, unitate Pascal, un pachet ADA, etc.) este realizată de programatorul care implementează modulul. Toate celelalte teste sunt efectuate, în general, de persoane care nu au participat la dezvoltarea programului.

Scopul testării unui modul este de a se stabili ca modulul este o implementare corectă a specificației sale (conformă cu specificația sa). Specificația poate fi neformală sau formală. De exemplu:

- o specificație de pre și post condiții pentru o funcție sau procedură;
- un invariant al clasei, care specifică mulțimea stărilor posibile ale fiecărui obiect din clasa respectivă, împreună cu specificații de pre și post condiții la nivelul funcțiilor membru;
- o specificație algebrică a clasei;

- o specificație Z/Z++ etc.

În cursul testării unui modul, modulul este tratat ca o entitate independentă, care nu necesită prezența 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 aceeași interfață cu modulul testat și realizează în mod simplificat funcția sa. De exemplu, dacă modulul testat apelează o funcție de sortare a unui vector, cu antetul:

```
void sortare(int n, int *lista);
```

se poate folosi următoarea funcție "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 citește lista sortată, furnizată de testor
  for(i=0; i<n; i++) scanf("%d", lista[i]);
}
```

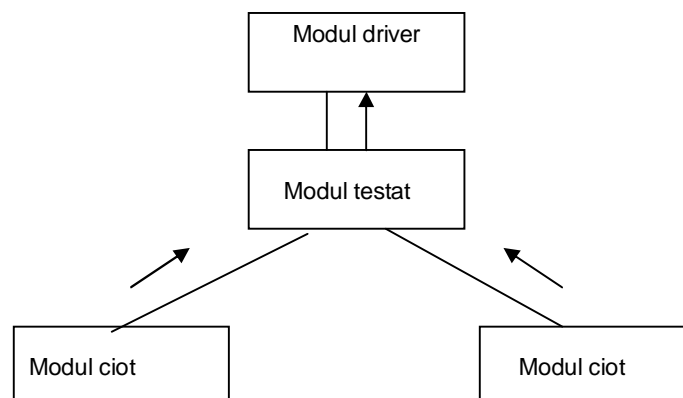


Fig.8.1. Structura programului executabil pentru testarea izolată a unui modul

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 furnizându-i ca date de intrare datele de test ale modulului. Datele de test pot fi generate de modulul driver, pot fi preluate dintr-un fișier sau furnizate de testor într-o manieră interactivă.

8.2.2. Testele de integrare

Sunt dedicate verificării interacțiunilor 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 și testele unitare, realizarea de module "ciot" și module "driver". Numărul de module "driver" și 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 și a configurațiilor.

8.2.2.1. Metoda "big-bang"

Sunt integrate într-un program executabil toate modulele existente la un moment dat. Modulele "driver" și "ciot" necesare sunt de asemenea integrate. Metoda este periculoasă căci toate erorile apar în același timp și localizarea lor este dificilă.

8.2.2.2. Integrare progresivă. 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 (și nici un modul "ciot").

Avantajele testării de jos în sus

Nu sunt necesare module "ciot". Modulele "driver" se implementează mult mai ușor 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 cerințelor este disponibil numai după testarea ultimului modul.

2. Corectarea erorilor descoperite pe parcursul integrării necesită repetarea procesului de proiectare, codificare și testare a modulelor. Principalele erori de proiectare sunt descoperite de abia la sfârșit, când sunt testate modulele principale ale programului. ceea ce, în general, conduce la reproiectare și reimplementare.

8.2.2.2. Integrarea descendentă (top-down)

Se începe prin testarea modulului principal, apoi se testează programul obținut prin integrarea modulului principal și a modulelor direct apelate de el, și așa mai departe. Metoda presupune implementarea unui singur modul "driver" (pentru modulul principal) și 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", ș.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 începutul procesului de integrare, atunci când sunt testate modulele principale ale programului. Aceste erori fiind corectate la început, se evită reproiectarea și reimplementarea majorității componentelor de nivel

mai coborât, așa cum se întâmplă când erorile respective sunt descoperite la sfârșitul procesului de integrare.

2. Programul obținut 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 și deci se poate exercita cu el în vederea validării cerințelor; acest aspect este de asemenea, foarte important în privința 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 excepția modulului principal.

2. Este dificil de simulat prin module "ciot" componente complexe și componente care conțin în interfață structuri de date.

3. În testarea componentelor de pe primele nivele, care de regulă nu afișează rezultate, este necesar să se introducă instrucțiuni de afișare, care apoi sunt extrase, ceea ce presupune o nouă testare a modulelor.

Aceste dezavantaje pot fi reduse aplicând tehnici hibride, de exemplu, folosind în locul unor module "ciot", direct modulele reale testate. Integrarea nu trebuie să fie strict descendenta. De exemplu, experiența arată că este foarte util să se înceapă prin integrarea modulelor de interfață utilizator. Aceasta permite continuarea integrării în condiții mai bune de observare a comportării programului.

8.3. Testele de sistem

Acestea sunt teste ale sistemului de programe și echipamente complet. Sistemul este instalat și apoi testat în mediul său real de funcționare. Sunt teste de conformitate cu specificația cerințelor de sistem (software) :

- *teste functionale*, prin care se verifică satisfacerea cerințelor functionale
- *teste prin care se verifică satisfacerea cerințelor ne-functionale* :
 - de performanță,

- 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 specificația cerințelor utilizatorilor, până când cele două părți cad de acord că programul este o reprezentare satisfăcătoare a cerințelor.
2. **Testarea beta:** programul este distribuit unor utilizatori selecționați, realizându-se astfel testarea lui în condiții 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 mentenanței sistemului. Pentru ușurarea 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;
- funcția / funcțiile exersate prin datele respective;
- rezultatele (datele de ieșire) așteptate;

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 și 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 specificației componentei testate, fără cunoașterea realizării ei; acest tip de testare se numește **testare “cutie neagră”** (“black box”).
2. Cazurile de test se determină prin analiza codului componentei testate. Acest tip de testare se mai numește și testare “cutie transparentă”, sau **testare structurală**.

8.4.1. Testarea “cutie neagră”.

Cazurile de test trebuie să asigure următoarele verificări:

- reacția componentei testate la intrări valide;
- reacția componentei la intrări nevalide;
- existența efectelor laterale la execuția componentei, adică a unor efecte care nu rezultă din specificație;
- performanțele 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 specificații se aplică unele metode fundamentate teoretic precum și o serie de euristici.

Alegerea cazurilor de test folosind clasele de echivalență

Cazurile de test pot fi alese partiționând atât datele de intrare cât și cele de ieșire într-un număr finit de clase de echivalență. Se grupează într-o aceeași clasă datele care, conform specificației, conduc la o aceeași comportare a programului.

O dată stabilite clasele de echivalență ale datelor de intrare, se alege câte un eșantion (de date de test) din fiecare clasă. De exemplu, dacă o dată de intrare trebuie să fie cuprinsă între 10 și 19, atunci clasele de echivalență sunt:

- 1) valori < 10

2) valori între 10 și 19

3) valori > 19

Se pot alege ca date de test: 9, 15, 20.

Experiența arată că este util să se aleagă date de test care sunt la frontiera claselor de echivalență. Astfel, pentru exemplul de mai sus ar fi util să se testeze programul pentru valorile de intrare 10 și 19.

Alte cazuri de test se aleg astfel încât la folosirea lor să se obțină eșantioane din clasele de echivalență ale datelor de ieșire (din interiorul și de la frontierele lor).

Exemplu:

Fie un program care trebuie să genereze între 3 și 6 numere cuprinse între 1000 și 2500. Atunci se vor alege intrări astfel încât ieșirea 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 puțin de 3 numere sau mai mult de 6 numere sau valori în afara intervalului [1000..2500]
- între 3 și 6 numere cuprinse între 1000 și 2500

În alegerea datelor de test trebuie să se elimine redundanțele rezultate din considerarea atât a claselor de echivalență de intrare cât și a celor de ieșire.

Unele programe tratează datele de intrare în mod secvențial. În aceste cazuri se pot detecta erori în program testându-l cu diverse combinații ale intrărilor în secvență. Metoda de partiționare în clase de echivalență nu ajută în alegerea unor astfel de combinații. Numărul de combinații posibile este foarte mare chiar și pentru programe mici. De aceea nu pot fi efectuate teste pentru toate combinațiile. În aceste cazuri este esențială experiența celui care alcătuiește setul de cazuri de test.

Testarea "cutie neagră" este favorizată de existența unei specificații formale a componentei testate.

Exemplu: Fie o funcție 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-condiția. Efectele laterale ale funcției "cauta" s-ar putea reflecta în modificarea unor variabile globale. Pentru evidențierea lor ar trebui ca la fiecare execuție de test a funcției să se vizualizeze valorile variabilelor globale înainte și după apelul funcției. Aceste verificări pot fi limitate, înlocuindu-se cu inspectarea_codului funcției, din care rezultă eventualitatea modificării unor variabile globale.

Setul minim de date de test trebuie să verifice funcția în următoarele situații:

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 poziție oarecare a tabloului
 - d). "numar" nu este în tablou
4. tablou cu număr impar de elemente ("nrelem" este impar)

și a,b,c,d ca la punctul 3.

Din specificație rezultă că funcția nu trebuie să modifice tabloul; de aceea, apelul său în programul de test trebuie să fie precedat și urmat de vizualizarea parametrului tablou.

Setul cazurilor de test ar putea fi:

- 1) intrări: orice tablou

nrelem=0

orice număr

ieșiri : valoarea funcției = -1

tabloul nemodificat

2) intrări: $x[0] = 10$

nrelem=1

număr = 10

ieșiri : valoarea funcției = 0

tabloul nemodificat: $x[0] = 10$

3) intrări: $x[0] = 10$

nrelem=1

număr = 15

ieșiri : valoarea funcției = -1

tabloul nemodificat: $x[0] = 10$

4) intrări: $x[0] = 10$, $x[1] = 20$

nrelem=2

număr = 10

ieșiri : valoarea funcției = 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 în structura de date;
- De multe ori programatorii neglijează situațiile în care colecția prelucrată în program are un număr de elemente neobișnuit, de exemplu zero sau unu;
- Uneori, programele de căutare se comportă diferit în cazurile: număr de elemente din colecție par, respectiv număr de elemente din colecție impar; de aceea sunt testate ambele situații

Concluzii:

- Setul de cazuri de test a fost determinat numai pe baza specificației componente și a unor euristici (este vorba de experiența 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 urmărește analiza fluxului datelor și a căilor de execuție, ci doar identificarea variabilelor globale cu care componenta interacționează.
- Clasele de echivalență se determină pe baza specificației.
 - Se aleg drept cazuri de test eșantioane din fiecare clasă de echivalență a datelor de intrare. Experiența arată că cele mai utile date de test sunt acelea aflate la frontierele claselor de echivalență.
 - Cazurile de test alese verifică componenta doar pentru un număr limitat de eșantioane din clasele de echivalență ale datelor de intrare; faptul că din testare a rezultat că ea funcționează corect pentru un membru al unei clase nu este o garanție că va funcționa corect pentru orice membru al clasei.
 - Se determină de asemenea clasele de echivalență ale datelor de ieșire și se aleg pentru testare datele de intrare care conduc la date de ieșire aflate la frontierele acestor clase.
 - Partiționarea în clase de echivalență nu ajută la depistarea erorilor datorate secvențierii datelor de intrare. În aceste cazuri este utilă experiența programatorului. Reprezentarea comportării programului printr-o diagramă de stări-tranziții poate fi folositoare în determinarea secvențelor de date de intrare de utilizat în testarea programului.

Testele “cutie neagră”, numite și teste funcționale sunt utile nu numai pentru testarea programului. Ele pot fi utilizate (ca teste statice) pentru verificarea unei specificații intermediare față de o specificație de nivel superior.

“Slăbiciunile” testelor funcționale:

- Nu este suficient să se verifice că programul satisface corect toate funcțiile specificate; unele proprietăți interne, nespecificate, ca de exemplu timpul de răspuns, nu sunt verificate;
- Programul este testat pentru “ceea ce trebuie să facă” conform specificațiilor și nu și pentru ceea ce face în plus, de exemplu pentru ca implementarea să fie mai eficientă sau pentru a facilita reutilizarea;

- În absența unui standard în privința specificațiilor formale, tehnicile de testare funcțională 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 instrucțiuni indivizibile maximele” și noduri care corespund instrucțiunilor de decizie. Blocurile de instrucțiuni indivizibile maximele sunt porțiuni liniare maximele de instrucțiuni care se execută întotdeauna în aceeași secvență. Fiecare bloc are un singur punct de intrare și un singur punct de ieșire. Arcele reprezintă transferul controlului între blocurile/instrucțiunile componentei program.

Graful program are un nod unic de intrare și un nod unic de ieșire. Dacă există mai multe puncte de intrare sau mai multe puncte de ieșire, acestea trebuie să fie unite într-unul singur (care se adaugă suplimentar). Nodul de intrare are gradul de intrare zero, iar cel de ieșire are gradul de ieșire 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);
```

Graful de control este redat în Fig.8.2.

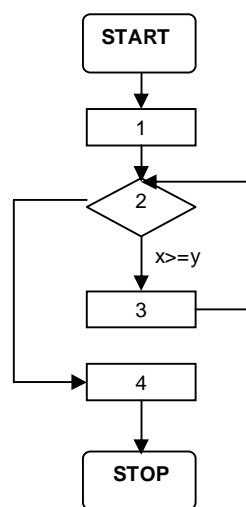


Fig.8.2. Graful de control

Se consideră căile care încep cu nodul de intrare și se termină cu nodul de ieșire. Testarea structurală constă în execuția componentei testate pentru date de intrare alese astfel încât să fie parcurse unele dintre aceste căi. Nu este necesar, și în general este imposibil, să se execute toate căile de la intrare la ieșire ale unui program sau ale unei componente program. Prezența ciclurilor conduce la un număr foarte mare (deseori infinit) de căi. De asemenea, anumite căi sunt ne-executabile.

Testele structurale favorizează evidențierea următoarelor tipuri de erori logice:

1. *Căi absente în graful program* - ca urmare a ignorării unor condiții (de exemplu: test de împărțire la zero);
2. *Selectarea unei căi necorespunzătoare* - datorită exprimării incorecte (de multe ori incomplete) a unei condiții;
3. *Acțiune 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 execuția tuturor instrucțiunilor programului.

Alegerea căilor de testat are loc pe baza unor "criterii de acoperire" a grafului de control. Dintre acestea cele mai cunoscute sunt:

Execuția tuturor instrucțiunilor

Cazurile de test se aleg astfel încât să se asigure execuția tuturor instrucțiunilor componente testate. Acesta este un criteriu de testare minimal.

El nu este întotdeauna satisfăcător.

De exemplu, pentru testarea componente cu graful din Fig.8.3, pe baza criteriului execuției tuturor instrucțiunilor, este suficient să se aleagă date de test care asigură execuția condiției și a instrucțiunilor I1 și I2. Nu se testează cazurile în care condiția are valoarea FALSE. Transferul controlului pentru astfel de cazuri poate fi eronat.

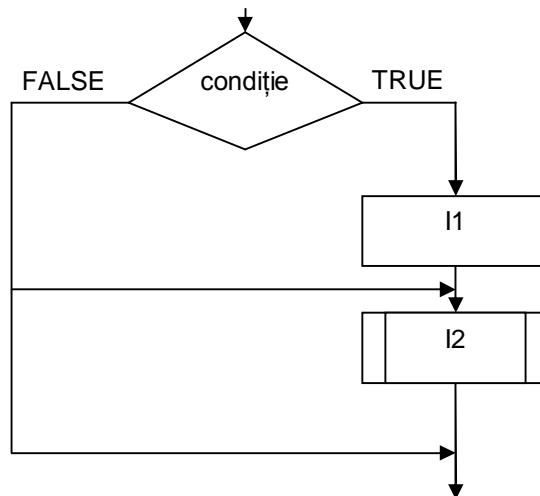


Fig.8.3. Graful componentei

Traversarea tuturor arcelor

Cazurile de test se aleg astfel încât să se asigure traversarea fiecărui arc al grafului program cel puțin pe o cale. Pe baza acestui criteriu se va putea verifica dacă transferul controlului este corect pentru valoarea FALSE a condiției, în exemplul de mai sus. În același 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 componente testate. Pentru majoritatea programelor nu se poate aplica acest criteriu, deoarece numărul de căi de execuție este infinit. Criteriul poate fi restricționat, 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 execuția fiecărei căi selectate;
- determinarea rezultatelor care trebuie să se obțină 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 ieșire;
 - b) fiecare arc din graful de control este cuprins într-o cale $c \in C$
 - c) $\sum_{c \in C} \text{lung}(c) = \text{minima}$
2. Se determină predicatul fiecărei căi, $c \in C$, $R_c(I)$, unde $I = (i_1, i_2, \dots, i_n)$ 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:

$$R_c(X_c) = \text{true}$$

Atunci setul datelor de test este:

$$DT = \{ X_c \mid c \in C, X_c \in D_I, R_c(X_c) = \text{true} \},$$

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 ieșire

Exemplu

Fie urmatorul graf program :

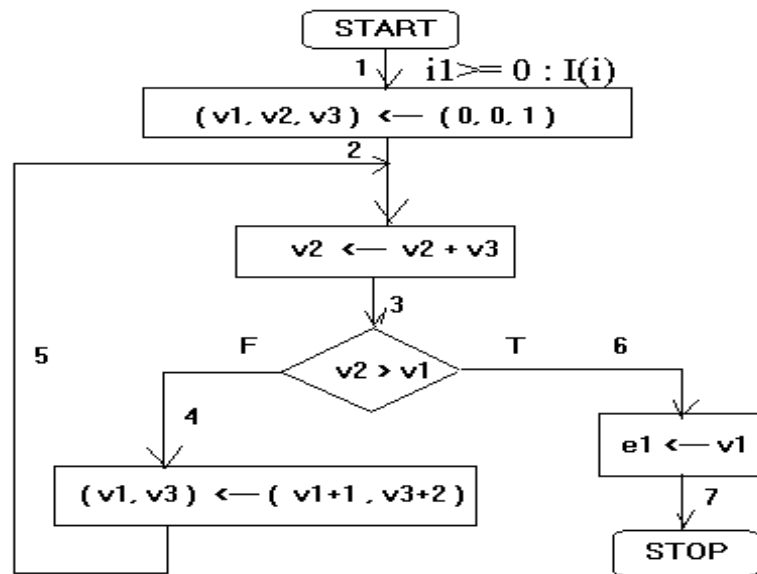


Fig. 8.4. Graf program

Obs: testul $v2 > v1$ trebuie înlocuit cu $v2 > i1$.

1) Alegerea căilor :

Se poate alege setul de căi C:

$$C = \{c_1, c_2\},$$

$$\text{unde } c_1 = \{1,2,3,4,5,3,6,7\}$$

$$c_2 = \{1,2,3,6,7\}$$

În afara condițiilor menționate s-au mai avut în vedere următoarele criterii:

- alegerea unei căi care să nu conțină ciclul
- alegerea unei căi care să conțină ciclul, parcurgându-l o singură dată, pentru ca lungimea căii să fie minimă.

2) Determinarea predicatelor de cale

O metodă de determinare a predicatului unei căi constă în concatenarea condițiilor de ramificare de pe calea respectivă pe măsura ce sunt întâlnite atunci când calea este parcursă de la nodul de ieșire până la cel de intrare. Totodată, la întâlnirea unei atribuirii, $y \leftarrow E$, dacă y face parte din expresia curentă a predicatului, se substituie y cu E .

În unele cazuri, predicatul obținut printr-o singură parcurgere a unui ciclu este fals pentru orice atribuire a variabilelor de intrare. Un astfel de caz corespunde unei căi neexecutabile. În consecință se va alege un predicat în care ciclul este parcurs de două sau de mai multe ori.

Construirea predicatelor de cale

$$\begin{array}{ll}
 Rc_1 = v_2 > i_1 & \\
 \downarrow 3 & \text{se substituie } v_2 \text{ cu } (v_2 + v_3) \\
 Rc_1 = (v_2 + v_3) > i_1 & \\
 \downarrow 5 & \\
 Rc_1 = (v_2 + (v_3 + 2)) > i_1 & \\
 \downarrow 4 & \\
 Rc_1 = (v_2 + v_3 + 2) > i_1 \wedge \sim(v_2 > i_1) & \\
 \downarrow 3 & \\
 Rc_1 = ((v_2 + v_3) + v_3 + 2) > i_1 \wedge \sim((v_2 + v_3) > i_1) & \\
 \downarrow 2 & \\
 Rc_1 = (4 > i_1) \wedge \sim(1 > i_1) & \\
 \downarrow 1 & \\
 Rc_1 = (4 > i_1) \wedge (i_1 \geq 1) \wedge (i_1 \geq 0) \Leftrightarrow Rc_1 = (1 \leq i_1 < 4) . &
 \end{array}$$

Calea c1 este executată pentru orice valoare a lui i_1 care satisface acest predicat.

Stabilirea rezultatelor execuției fiecărei căi pentru fiecare set de date de test ales, necesită cunoașterea transformării realizate de componenta analizată asupra datelor de intrare. Aceasta rezultă din specificația componentei. În cazul de față trebuie să se stabilească ce valoare trebuie să aibă ieșirea e_1 pentru fiecare valoare aleasă pentru i_1 . Programul reprezentat prin grafurile de control analizat calculează numărul maxim de termeni ai sumei $1+3+5+\dots+(2j+1)$, $\forall j \geq 0$, care pot fi adunați a.î. suma lor să fie mai mică decât i_1 . Astfel se poate verifica că pentru orice $1 \leq i_1 \leq 4$, $e_1 = 1$, căci dacă s-ar aduna doi termeni, $1+3 = 4$, $4 > i_1$.

Cazul de test al căii c1 poate fi: $i_1=2$, $e_1 = 1$.

Calea c2 = {1,2,3,6,7}

$$\begin{array}{ll}
 Rc_2 = v_2 > i_1 & \downarrow 3 \\
 Rc_2 = (v_2 + v_3) > i_1 & \downarrow 2 \\
 Rc_2 = 0 + 1 > i_1 & \\
 \downarrow 1 &
 \end{array}$$

$$R_{C_2} = (i_1 < 1) \wedge (i_1 \geq 0) \quad \Leftrightarrow \quad R_{C_2} = (0 \leq i_1 < 1) \Rightarrow \quad R_{C_2} = (i_1 = 0)$$

Cazul de test: $i_1 = 0$, $e_1 = 0$.

“Slabiciunile” testelor structurale sunt:

- testele selectate depind numai de structura programului; ele trebuie recalulate la fiecare modificare; nu pot fi reutilizate eficient de la o versiune la alta;
- testele selectate acoperă cazurile legate de ceea ce “face” componenta testată și nu de ceea ce “trebuie să facă”; astfel, dacă un caz particular a fost uitat în faza de implementare, testul structural nu releva această deficiență; el trebuie deci completat cu testul funcțional. Mai exact trebuie să se înceapă cu testarea funcțională care se completează cu testarea structurală.

Testele statistice

Testele statistice se efectuează în timpul testării 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 distribuția datelor de intrare estimate pentru exploatarea programului – aceste teste se mai numesc **teste statistice operaționale**.

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 explicație ar consta în faptul că ele asigură o bună acoperire în cazul programelor pentru care domeniile de intrare ale căilor de execuție au o probabilitate comparabilă. Ele nu acoperă căile care corespund tratării excepțiilor.

De aceea, trebuie completate cu teste folosind date de intrare în afara domeniului intrărilor.

Testele statistice operaționale sunt în general **teste de fiabilitate**. Prin ele nu se urmărește 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).

9

Estimarea costului unui proiect software

9.1. Costuri și efort

Estimarea costurilor unei aplicații software este greu de realizat cu precizie. În cele mai multe modele de estimare, este presupusă o relație simplă între cost și efort; efortul poate fi măsurat în luni-om. Există o relație între efortul necesar și mărimea produsului (KLOC – kilolines of code, kilo-linii de cod).

Pentru a determina ecuațiile unui model algoritmic de estimare a costului există mai multe abordări:

1. Un parametru variază în timp ce ceilalți parametri rămân constanți și se determină influența parametrului variabil asupra rezultatului
 - De ex. comentariile asupra depanării și întreținerii
2. Se analizează datelor unor proiecte anterioare
 - De ex. timpul necesar fazelor de dezvoltare, calificarea personalului implicat, mărimea produsului final;

Analiza costurilor permite identificarea unor strategii de creștere a productivității software:

- Scrierea de mai puțin cod
- Stimularea oamenilor să lucreze la capacitatea maximă
- Evitarea refacerii componentelor dezvoltate anterior
- Dezvoltarea și folosirea mediilor de dezvoltare integrate

9.2. Modelul Halstead

Își propune o estimare mai obiectivă a mărimii unui program pe baza unui număr de unități sintactice: operanzi și operatori

- Entități de bază:

n_1 = numărul de operatori diferiți

n_2 = numărul de operanzi diferiți

N_1 = numărul total de apariții ale operatorilor

N_2 = numărul total de apariții ale operanzilor

Vocabularul: $n = n_1 + n_2$

Lungimea implementării: $N = N_1 + N_2$

Ecuția 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:

$$\text{LOC} = 102 + 5,31 * \text{VARS}$$

- Fiecare program care va conține 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 interfața dintre componente și 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 știm că o ofertă de 100000 de euro a fost făcută de concurență, noi vom face o ofertă de 90000 de euro. Acesta este cunoscut sub denumirea de *preț de câștig*.

Dorim să ne promovăm produsul la un anumit târg de tehnică de calcul și din acest motiv programul trebuie scris și testat în următoarele 9 luni, deși realizăm că timpul este limitat. Această situație este cunoscută sub denumirea de *metoda bugetului* de estimare a costului.

Proiectul poate fi dezvoltat într-un an, dar șeful nu ar accepta acest termen. Știm că termenul de 10 luni este acceptabil și 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 experienței.

În 1968, unei echipe de programatori i s-a cerut să dezvolte un compilator FORTRAN pentru trei mașini diferite.

Rezultate:

Compilatorul	Efortul (în luni-om)
1	72
2	36
3	14

Consultarea experților

- Metoda Delphi

Fiecare expert își expune opinia în scris. Un moderator colectează estimările obținute astfel și le redistribuie celorlalți experți. Numele experților nu sunt asociate cu estimările lor. Fiecare expert va preda o nouă estimare bazată pe informațiile primite de la moderator. Procesul continuă până când se ajunge la un consens

- Distribuția beta

Un expert realizează mai multe estimări: o estimare optimistă a , o estimare realistă m și o estimare pesimistă b .

Efortul așteptat va fi:

$$E = (a + 4m + b) / 6,$$

o estimare mai bună probabil decât dacă s-ar fi considerat numai media aritmetică a lui a și b .

9.3. Modelele algoritmice clasice - Modele liniare

Efortul pentru realizarea unui proiect este:

$$E = a_0 + \sum_{i=1}^n a_i x_i$$

- Coeficienții 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 specificațiilor cerințelor	0-2
x_2	Instabilitatea proiectării	0-3
x_3	Procentajul de instrucțiuni matematice	0-100
x_4	Procentajul de instrucțiuni I/O	0-100
x_5	Numărul subprogramelor	număr
x_6	Utilizarea unui limbaj de nivel înalt	0(da) / 1(nu)
x_7	Aplicație comercială	0(da) / 1(nu)
x_8	Program de sine stătător	0(da) / 1(nu)
x_9	Primul program pe această mașină	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}	Mașină gazdă diferită de mașina țintă	1(da) / 0(nu)
x_{13}	Număr de erori	număr
x_{14}	Dezvoltare pentru o organizație militară	0(da) / 1(nu)

Dacă avem o estimare E , atunci efortul real R va verifica formula:

$$P((1-\alpha)E \leq R \leq (1+\alpha)E) \geq \beta,$$

unde valori acceptabile pentru α și β sunt:

$$\alpha = 0,2 \text{ și } \beta = 0,9.$$

Exemplu: Să presupunem că estimarea este de 100 luni-om. Atunci probabilitatea ca proiectul să necesite în realitate între 80 și 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 și un număr de nivele de complexitate.

Tipul modului	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. Interfață utilizator	13	16	19	23	29
5. Control	20	25	30	35	40

Fiind dată o matrice de costuri C , un modul de tip l , complexitate j și mărime S_k , va rezulta un cost al modului $M_k = S_k \cdot C_{ij}$

9.4. Modele algoritmice moderne – Modele neliniare

Forma generală este:

$$E = (a + b \cdot KLOC^c) \cdot f(x_1, \dots, x_n),$$

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
<i>Halstead</i>	$E = 0.7 \cdot KLOC^{1.50}$
<i>Boehm</i>	$E = 2.4 \cdot KLOC^{1.05}$
<i>Walston-Felix</i>	$E = 5.2 \cdot KLOC^{0.91}$

- $c < 1$: analogie cu producția de masă (costurile fixe se împar pe mai multe unități de produs);
- $c > 1$: efortul crește exponențial cu mărimea datorită creșterii complexității
 - Pentru proiecte mari, această relație 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 influențează productivitatea. Pentru fiecare din aceste variabile au fost considerate trei niveluri: mare, mediu și mic.

Variabila	Productivitatea medie pentru valoarea variabilei			PC
	< normală	normală	> normală	
Complexitatea interfeței utilizator	500	295	124	376
Calificarea și experiența personalului	mică 132	medie 257	mare 410	278
Experiența anterioară cu aplicații similare	minimă 146	medie 221	vastă 410	264
Procentajul de programatori participanți în faza de proiectare	< 25% 153	25 - 50% 242	> 50% 391	238
Raportul dintre mărimea medie a echipei și durata proiectului (persoane/lună)	< 0,5 305	0,5 – 0,9 310	> 0,9 171	134

Walston și Felix consideră că indexul productivității I poate fi determinat pentru noile proiecte după următoarea relație

$$I = \sum_{i=1}^{29} W_i X_i$$

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 aplicațiile militare);
- *Semidetășate*: o formă intermediară;

Exemple

Clasa de proiect	b	c
organică	2,4	1,05
semidetașată	3,0	1,12
integrată	3,6	1,20

KLOC	organic	semidetașat	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 funcționale

Se bazează pe numărarea diferitelor structuri de date utilizate. Este potrivită mai ales pentru aplicațiile comerciale, în care structura datelor are o foarte mare importanță. Este mai puțin indicată pentru proiectele în care algoritmi joacă rolul dominant (de ex. compilatoarele sau aplicațiile în timp real).

Analiza se bazează pe modalitățile în care diverși utilizatori interacționează cu aplicațiile.

Se consideră că sistemul îndeplinește cinci funcții fundamentale:

- Funcții referitoare la date
 1. Fișiere interne logice;
 2. Fișiere externe de interfață ;
- Funcții tranzacționale
 3. Intrări externe;
 4. Ieșiri externe;
 5. Cereri externe;

- 1. Fișiere interne logice** – în engleză “Internal Logical Files”, FIL. Permit utilizatorilor să folosească datele pe care trebuie să le întrețină. De exemplu, un pilot poate introduce datele de navigare la un terminal din carlingă înainte de plecare. Datele sunt stocate într-un fișier și pot fi modificate în timpul misiunii. Pilotul este deci responsabil pentru întreținerea acestor date.
- 2. Fișierele externe de interfață** – în engleză “External Interface Files”, FEI. Utilizatorul nu este responsabil pentru întreținerea datelor; acestea sunt localizate în alt sistem care le întreține. Utilizatorul sistemului analizat solicită datele doar pentru informare. De exemplu, un pilot se poate informa asupra poziției cu ajutorul sateliților 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 întrețină fișierele interne logice prin operații de adăugare, modificare și ștergere.
- 4. Ieșirile externe**, în engleză, “External Output”, EE. Permit utilizatorului să producă date de ieșire. De exemplu, pilotul poate să afișeze separat viteza la sol și viteza reală în aer, informații derivate din datele interne (pe care le poate întreține) și cele externe (pe care le poate accesa).
- 5. Cererile externe**, în engleză, “External Inquiries”, CE. Pentru ca utilizatorul să poată selecta și afișa datele din fișiere, el trebuie să introducă informații de selecție pentru a găsi datele în conformitate cu anumite criterii. În această situație datele din fișiere nu sunt modificate, ci doar căutate și furnizate. De exemplu, dacă pilotul afișează date cu privire la relieful solului, date stocate anterior, rezultatul este regăsirea directă a informațiilor.

Puncte funcționale neajustate

Prin încercări repetate, s-au stabilit ponderi pentru fiecare dintre aceste entități. Numărul de puncte funcționale neajustate este:

$$PFN = 10 \cdot FIL + 7 \cdot FEI + 4 \cdot IE + 5 \cdot EE + 4 \cdot CE$$

În funcție de complexitatea tipurilor de date, se disting o serie de valori pentru aceste puncte funcționale, 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 funcționale ajustate

Pentru ajustarea suplimentară a estimărilor, se iau în calcul și alte 14 caracteristici care influențează dezvoltarea aplicațiilor:

- Comunicațiile de date
- Funcțiile distribuite
- Performanța
- Folosirea masivă a configurațiilor
- Rata tranzacțiilor
- Intrările de date online
- Eficiența utilizatorilor finali
- Actualizările online
- Prelucrările complexe
- Refolosirea
- Ușurința la instalare
- Ușurința la folosire
- Locațiile multiple
- Facilitarea modificărilor

Influența fiecărei caracteristici este evaluată pe o scară de la 0 (nu influențează) la 5 (influență puternică). Gradul de influențare GI este suma acestor puncte pentru toate caracteristicile

- Se calculează apoi factorul de complexitate tehnică: $FCT = 0,65 + 0,01 \cdot GI$
- Punctele funcționale ajustate (PF) se obțin astfel: $PF = PFN \cdot FCT$.

Avantaje

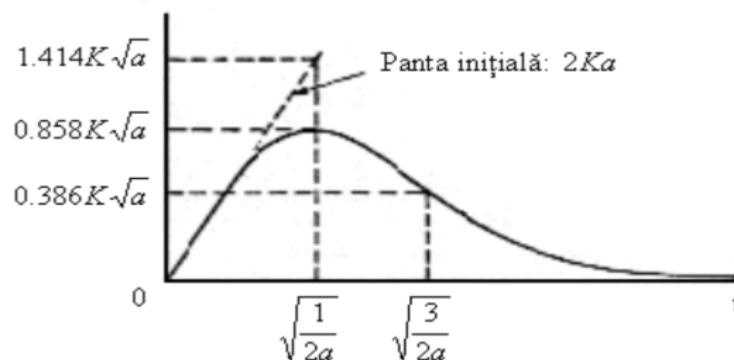
- Măsura productivității este un rezultat natural, deoarece punctele funcționale sunt independente de tehnologie și deci pot fi utilizate pentru a compara productivitatea pe platforme diferite și cu instrumente de dezvoltare diferite;
- Ele pot fi folosite pentru a stabili o rată de productivitate (PF / h) care facilitează estimările privind durata proiectului ca întreg;

Distribuirea forței de muncă în timp

Modelul Putnam-Norden

Distribuția forței de muncă în timp are de multe ori o formă caracteristică, bine aproximată de distribuția Rayleigh. Astfel, forța de muncă necesară la un moment de timp t este:

$$FM(t) = 2 \cdot K \cdot a \cdot t \cdot e^{-at^2}$$



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 întreținere.

Ecuția software-ului

Putnam a folosit observații empirice legate de nivelurile de productivitate pentru a deriva *ecuația software-ului* din curba Rayleigh:

$$D = k \cdot E^{1/3} \cdot t^{4/3},$$

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 și tehnicile de management;
- Gradul de utilizare a tehnicilor de ingineria programării;
- Nivelul limbajelor de programare folosite;
- Capacitatea și experiența echipei de dezvoltare;
- Complexitatea aplicației.

Efortul

Pentru estimarea efortului, Putnam a introdus *ecuația acumulării forței de muncă*:

$$A = E/t^3,$$

unde A este numită accelerarea forței de muncă iar E și t au semnificațiile de mai sus.

Accelerarea forței de muncă este 12,3 pentru proiecte software noi, cu multe interfețe și interacțiuni cu alte sisteme, 15 pentru sisteme de sine stătătoare și 27 pentru reimplementări ale sistemelor existente.

Pe baza celor două ecuații putem elimina timpul și determina efortul:

$$E = (D/k)^{9/7} \cdot A^{4/7}$$

Acest rezultat este interesant deoarece arată că efortul este proporțional cu dimensiunea la puterea $9/7 \approx 1,286$, valoare similară cu factorul Boehm, între 1,05 și 1,20.

Consecințe

- 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 inițială a curbei; vârful curbei Rayleigh se deplasează spre stânga și în același timp în sus;
- Astfel obținem o creștere a puterii necesare la începutul proiectului și o forță de muncă maximă mai mare.

Legea lui Brooks

Mai multe studii au arătat că productivitatea individuală scade odată cu creșterea echipei. Conform lui Brooks, există două cauze ale acestui fenomen:

- Crește timpul acordat comunicării cu ceilalți membri ai echipei (pentru consultare, sincronizarea sarcinilor etc.);
- Mai întâi scade productivitatea, deoarece noii membri ai echipei nu sunt productivi de la început. În același timp ei necesită ajutor, deci timp, de la ceilalți membri ai echipei în timpul procesului de învățare
- *Legea lui Brooks*: adăugarea de personal la un proiect întârziat îl va întârzia și mai mult.

Mărimea echipei și 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 relație simplă între prețul unui sistem și costul său de dezvoltare;
- Factorii care afectează productivitatea includ aptitudinile individuale, experiența în domeniu, natura proiectului, dimensiunea proiectului, instrumentele utilizate și mediul de lucru;
- Prețul unui produs software poate fi stabilit astfel încât să se câștige un contract și apoi funcționalitatea este ajustată conform prețului;
- Pentru estimarea costului pot fi folosite tehnici diferite;
- Modelele algoritmice de estimare a costurilor se bazează pe analiza cantitativă a caracteristicilor proiectelor, permițând compararea influenței acestor caracteristici;
- Timpul necesar terminării unui proiect nu este proporțional cu numărul de persoane care lucrează la proiect.

10

Calitatea sistemelor software

10.1. Indicatori de calitate

Deși software-ul a apărut mai târziu ca domeniu al ingineriei, calitatea lui trebuie demonstrată și garantată. Problema care a apărut astfel în ingineria software se referă la modul în care se definește în acest caz calitatea.

În conferințele specialiștilor din domeniu au existat argumente că s-ar putea defini drept: "ceva potrivit pentru utilizare", sau "satisfacerea cerințelor utilizatorilor" sau "absența defectelor" (Grady, 1993).

Unul dintre principalele obiective ale ingineriei software este de a pune la dispoziție dezvoltatorilor metodologii și 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. Acești factori pot fi contradictorii, cum ar fi de exemplu portabilitatea și integritatea care aduc adesea prejudicii eficacității, dar oferă o platformă comună de comparare și evaluare a produselor software.

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 puțin voluntară, de către constructorul instrumentului software respectiv.

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

Validitatea este aptitudinea software-ului de a-și îndeplini corect funcțiile.

Transpusă în planul concepției, această aptitudine ajută ca specificațiile definite să reflecte într-adevăr cerințele utilizatorilor.

Fiabilitatea este aptitudinea produsului software de a funcționa în condiții anormale, în conformitate cu specificațiile sale de proiectare (definiție dată de Mayer). De exemplu, rezistența la introducerea de date incorecte, reluări posibile în caz de pană a sistemului (software sau hardware), posibilitățile de a lucra când totuși sistemul este degradat, etc. Acest factor nu este independent de contextul tehnic în care se utilizează aplicația software.

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

Factor	Definiții
Validitate	Aptitudinea produsului software de a îndeplini exact funcțiile sale, definite prin caietul de sarcini și prin specificare.
Fiabilitate	Aptitudinea produsului software de a funcționa în condiții anormale
Extensibilitate	Ușurința cu care un software se pretează la o modificare sau la o extindere a funcțiilor solicitate.
Reutilizabilitate	Aptitudinea unui produs software de a fi reutilizat, în totalitate sau și parțial, într-o nouă aplicație.
Compatibilitate	Ușurința cu care un sistem poate fi combinat cu altele
Eficacitate	Utilizarea opțională a resurselor materiale (procesoare, memorie internă și externă, protocoale de comunicație, etc)
Portabilitate	Ușurința cu care un produs poate fi transferat pe medii diferite hardware și software.
Verificabilitate	Ușurința cu care se pregătesc procedurile și testele de validare (în particular "jocurile de încercare") și procedurile de detectare a erorilor care urmează unui incident în exploatare.
Integritate	Aptitudinea software-ului de a-și proteja codul și datele de accesări neautorizate.
Ușurința de utilizare	Facilitatea de înțelegere, de utilizare, de pregătire a datelor, de interpretare a erorilor, de revenire în caz de utilizare eronată.

Extensibilitatea este posibilitatea de a modifica cu ușurință software-ul, în ideea adăugării de noi funcții sistemului . Este un factor de calitate dificil de

măsurat, pentru că adesea se leagă de nivelul limbajului de programare (de exemplu, este mai ușor de modificat în Cobol decât în limbaj de asamblare, este mai ușor de modificat în Pascal decât în Cobol).

Reutilizarea este aptitudinea software-ului de a putea fi utilizat în aplicații noi. Conceptul nu este nou, de multă vreme se fac subprograme generale de tratare a datelor, sau schelete de programe care să automatizeze diverse funcții.

Proprietatea de moștenire apărută în abordarea orientată pe obiecte a permis ca metodele scrise pentru un obiect să poată fi moștenite de un altul.

Compatibilitatea, a nu se confunda cu portabilitatea, reprezintă ușurința cu care un produs software poate fi combinat cu altele. Este nevoie în unele situații să se achiziționeze aplicații din exterior (care vor deveni subsisteme în aplicația care se realizează) și care trebuie să funcționeze împreună cu aplicațiile deja existente.

Eficacitatea este utilizarea optimală a resurselor materiale. De când există Informatica, puterea mașinilor se dublează la fiecare 18 luni, dar tot este insuficient pentru a putea ignora problemele de performanță.

Pe de o parte, se realizează aplicații din ce în ce mai sofisticate care presupun calculatoare performante, pe de altă parte gradul de putere al mașinii este consumat pentru confortul utilizatorului : pe un Macintosh sau un PC sub Windows, mai mult de 80% din capacitate este utilizată pentru a trata interfața grafică, și numai 20% rămân pentru a putea fi utilizați de aplicație.

Portabilitatea este ușurința cu care un produs software poate fi transferat pe un alt hardware sau sistem de operare.

Acesta este un criteriu pe care furnizorii de instrumente îl pun în față, pentru că permite diminuarea dependenței utilizatorului de hardware.

Verificabilitatea este facilitatea prin care se pregătesc procedurile de tratare și validare a sistemului. Este un criteriu interesant la care trebuie reflectat puțin: se poate cu ușurință verifica dacă un sistem merge sau nu, se pot construi teste de încercări care să garanteze că o parte din software sau tot sistemul funcționează ?

Când ne confruntăm cu o problemă de testare a unui software care nu este cunoscut și nici nu a mai fost realizat până atunci, nu este ușor de stabilit dacă rezultatul furnizat de sistem este corect sau nu.

Integritatea este aptitudinea sistemului de a-si proteja codul și datele de utilizări neautorizate; este mai mult o caracteristică generală de mediu în care trebuie să funcționeze sistemul, decât o caracteristică directă furnizată prin instrumentul software. Dar, instrumentul trebuie eventual să suplinească carențele mediului de dezvoltare.

Facilitatea de utilizare reprezintă tot ceea ce este legat de ergonomia aplicației, de ușurința cu care este utilizat. Nu este același lucru pentru utilizator dacă înțelege intuitiv și repede modul în care funcționează sistemul sau îi trebuie trei luni pentru a-l face operațional.

De asemenea, nu este același lucru dacă primește un mesaj de eroare clar, sau trebuie să consulte un manual pentru a-i înțelege semnificația.

Ergonomia este pe punctul de a deveni un criteriu foarte important pentru funcționalitatea însăși a sistemului, pentru că, fără îndoială, se consideră drept "competență" 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 și productivitatea acoperă aspecte diferite, ele presupun adesea un compromis. Dacă se vizează productivitatea în mentenanță, va trebui fără îndoială să se investească mai mult în concepție, pentru a găsi structuri de date mai generale și 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 funcționează, modul de abordare este altul. Sunt situații în care nu există altă alegere sau sunt cazuri extreme care pot justifica existența chiar a două instrumente de dezvoltare, unul care să producă mai repede, dar care furnizează un cod mai puțin eficient sau mai puțin

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 coborâ.

Productivitatea unui proiect trebuie să se măsoare în mod global, la capătul mai multor ani de utilizare care să includă toate fazele, de la concepție la mentenanță. 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 și implicit și costul produsului va fi mai mic. Mai mult, interesul producătorilor de software pentru achiziționarea de instrumente performante de lucru care să mărească productivitatea va fi în continuă creștere.

10.3. Asigurarea fiabilității produselor software

10.3.1. Niveluri prescrise de fiabilitate

Încă din faza stabilirii cerințelor și specificațiilor pentru o aplicație informatică trebuie să se impună nivelul de fiabilitate al sistemului ca un compromis între prețul de cost și consecințele defectărilor. Astfel un produs software conceput pentru cercetări spațiale cu participare umană trebuie să aibă o fiabilitate mai mare decât unul pentru jocuri pe calculator. Este necesar deci ca produsele software să se înscrie în anumite clase de fiabilitate. Clasele de fiabilitate sunt următoarele:

a) Foarte scăzută (very low). Acest nivel se prescrie atunci când defectarea are ca singură consecință inconvenientul producătorului de a înlătura o neregulă în program. Asemenea situație 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, ușor de recuperat, pentru beneficiar. Sistemele utilizate în predicție pe termen lung sunt exemple tipice.

c) *Nominală (nominal)*. Acest nivel corespunde cazului când defectarea implică o pierdere moderată pentru beneficiar, dar remediarea situației nu este prea costisitoare. Sistemul de gestionare a stocurilor sau sistemele informaționale 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 și cele ale distribuției energiei electrice.

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

În funcție de nivelul de fiabilitate prescris, se poate stabili ce efort necesar să fi alocat în fiecare etapă a realizării produsului. În fig. 7.5 se reprezintă acest efort normat la cel necesar obținerii unei fiabilități nominale. Figura poate fi utilizată și în vederea evaluării produsului pe baza efortului investit în procesul de realizare. Se observă că diferențele în ceea ce privește efortul investit devin mai mari în fazele de integrare și testare.

10.3.2. Proiectarea structurii pentru asigurarea fiabilității

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

P_1 = numărul total de module din program

P_2 = numărul de module dependente de intrări (I) sau ieșiri (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 și o ieșire

Cu ajutorul mărimilor $P_1 - P_7$ se evaluează șase 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) și cu 1 în caz contrar;

- D_2 exprimă dependența la nivelul modulelor;
- D_3 exprimă dependența de procesarea anterioară;
- D_4 arată mărimea bazei de date;
- D_5 arată compartimentarea bazei de date;
- D_6 arată mărimea interacțiunii programului cu exteriorul;

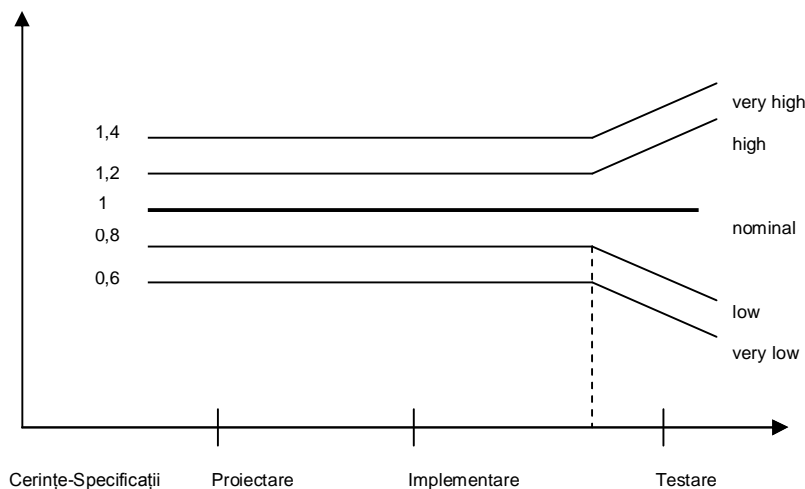


Fig.10.1 Reprezentarea efortului pentru asigurarea nivelului de fiabilitate prescris

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 funcție de importanța fiecărei caracteristici D_i . Valori apropiate de 1 pentru $D_1 - D_6$, respectiv pentru DSM, indică o deficiență a proiectului care trebuie corectată prin re-proiectare, astfel încât să se micșoreze 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 îmbunătățește proiectul. Indicele de structurare permite de asemenea evaluarea comparativă a unor proiecte diferite.

10.3.3. Complexitatea fluxului informațional

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 informație, care parcurge structura. În acest scop, se determină fluxul de informație între module, considerându-se că un flux de informație de la A la B are loc dacă:

- (1) A apelează B sau,
- (2) B apelează A și A returnează o valoare utilizată ulterior de B sau,
- (3) atât A cât și B sunt apelați de un alt modul care face să treacă o valoare de la A la B.

Fie:

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

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

$$fanin = l_{fi} + datain;$$

$$fanout = l_{fo} + dataout$$

În ansamblu, complexitatea informațională IFC este dată de expresia:

$$IFC = length (fanin * fanout)^2.$$

Cu ajutorul acestei caracterizări se pot identifica acele module care, având o complexitate informațională mai mare, sunt probabil afectate de erori, deci au o fiabilitate mai redusă. Se identifică, de asemenea, procedurile care au mai mult decât o funcție, punctele de trafic informațional intensiv și modulele cu o complexitate funcțională excesivă.

Examinarea complexității informaționale trebuie întreprinsă în faza proiectării detaliate și 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 și Kemerer, 1991) șase metrici de proiectare orientate obiect și anume:

- DIT (Depth of the Inheritance Tree) - adâncimea arborelui de moștenire;
- NOC (Number of Children) - numărul de moștenitori;
- 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 poziția clasei în ierarhia de clase. Se adresează conceptului de moștenire din OOP. Se poate face ipoteza că cu cât este mai mare metrica **DIT** cu atât este mai greu de menținut 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 moștenire

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

b). Metrica **NOC** măsoară numărul de moștenitori direcți ai unei clase. Se are în vedere același concept de moștenire din OOP dar se consideră că cu cât numărul de moștenitori direcți ai unei clase este mai mare cu atât este mai afectat potențialul de moștenire.

De exemplu, dacă sunt mai multe subclase a unei clase care au dependențe de metode sau instanțe 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 menținut clasa. Deci:

NOC = numărul subclaselor directe

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

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

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

$RFC = \text{numărul de metode locale} + \text{numărul de metode apelate de metodele locale}.$

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

d). Metrica **LCOM** măsoară lipsa coeziunii clasei. Coeziunea unei clase este caracterizată prin cât de strâns sunt legate metodele locale de variabile locale instanțiate în clasă. Această metodă se adresează conceptului de metodă din OOP. Pare logic că o clasă care are o mai mare coeziune este mai ușor de întreținut. Deci cu cât metrica LCOM este mai mare cu atât este mai dificil de întreținut 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 și partiționată.

$LCOM = \text{numărul de seturi disjuncte din metodele locale}.$

Nu există intersecție a două mulțimi și 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 înțeles și de întreținut.

$WMC = \text{suma complexităților ciclice McCabe din metodele locale}.$

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

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

Pentru programele structurate o echivalență mai simplă pentru complexitatea ciclică este contorizată de condițiile 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. Metrice de definire și adăugare orientate pe obiect

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

Formele de cuplare ale obiectelor sunt: cuplare prin moștenire, cuplare prin transmitere de mesaje și cuplare prin date abstracte.

Cuplarea prin moștenire

Moștenirea promovează reutilizarea software-ului în metodele orientate pe obiect, dar creează de asemenea posibilitatea violării încapsulării și ascunderii informației. Aceasta apare deoarece proprietățile din superclase sunt expuse subclaselor fără nici o restricție de acces. Utilizarea moștenirii, dacă nu este bine proiectată, poate introduce complexitate suplimentară în sistem, datorată atributelor care sunt încapsulate în superclasă care sunt expuse fără restricții accesului din subclase. Mai mult, o clasă moștenește mai multe atribute neprivate decât accesează. DIT și NOC sunt folosite pentru măsurarea caracteristicilor de moștenire.

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 și lista de parametrii pentru metodă.

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

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

MPC = numărul de "instrucțiuni" definite și 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 recepționate 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 definiție a clasei, determinând astfel un tip special de cuplare dintre X și cealaltă clasă, pentru că X are acces la proprietățile clasei ADT. Acest tip de cuplaj poate produce violarea încapsulării dacă limbajul de programare permite accesul direct la proprietățile 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 definițiile altor clase.

O altă metrică utilizată este numărul de metode dintr-o clasă (NOM). Deoarece metodele locale dintr-o clasă constituie interfață clasei, NOM servește drept cea mai bună metrică de interfață.

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 interfață 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 funcție, iar cumularea metricilor LOC din toate procedurile și funcțiile este folosită pentru măsurarea programului. Totuși 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 același lucru.

SIZE1 = numărul de ";" dintr-o clasă.

A doua metrică de dimensiune folosită este numărul de proprietăți (incluzând atribute și metode) definite într-o clasă. Această metrică este SIZE2 și 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 și 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 și eventuale instrumente de "înregistrare" ale acestor metrici.

10.5. Modele de studiere "a posteriori" a fiabilității produselor software

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

Așa cum s-a arătat anterior, aceasta conduce la mărirea timpului de testare și nu garantează în totalitate că la sfârșit 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 inițial;

2. Rata de detecție a erorilor este proporțională cu numărul erorilor reziduale;
3. Fiecare eroare descoperită este imediat corectată, astfel încât numărul erorilor conținute în program scade cu o eroare la fiecare moment de detecție;
4. Rata erorilor între două momente de detecție succesive este constantă.

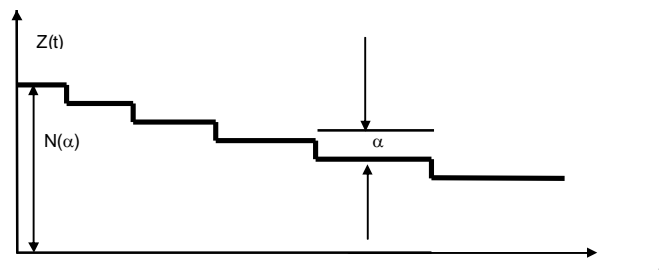


Fig.10.2. Curba lui $z(t)$

Pornind de la aceste ipoteze, rata de detecție a erorilor $z(t)$ se modelează cu relația:

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

unde:

N - numărul inițial al erorilor;

α - un coeficient de proporționalitate, reprezentând decrementul lui $z(t)$ corespunzător detecției și corecției unei erori;

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

Fig. 7.6 reprezintă dependența lui z în funcție de t .

MODELUL II

Acest model se bazează pe următoarele ipoteze:

1. Numărul de erori existente într-un program la începutul fazei de testare descrește pe măsură ce erorile sunt corectate;
2. Numărul de erori reziduale N_r este egal cu diferența între numărul de erori inițial prezente N și numărul cumulativ de erori corectate N_c ;

3. Numărul de instrucțiuni în limbaj mașină rămâne constant.

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

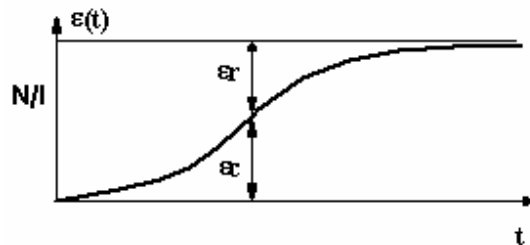


Fig.10.3. Curba erorilor reziduale

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

ε_c - număr de erori corectate pe instrucțiune

ε_r - număr de erori reziduale pe instrucțiune

În intervalul de timp $(t, t + \Delta t)$ 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 instrucțiuni.

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 și formulările modelului I, este valoros prin aceea că permite o delimitare mai precisă a timpului (timp real de execuție - 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 corecție a erorilor este proporțională cu rata de detecție a erorilor;
2. Rata erorilor instantanee este proporțională cu numărul erorilor reziduale;
3. Manifestarea tuturor erorilor care intervin în cursul execuțiilor programului este efectiv observată.

Rata de detecție a erorilor $z(t)$ este modelată de relația:

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

unde:

N - numărul inițial al erorilor;

t - timpul cumulat de funcționare al unității centrale;

α - coeficient de proporționalitate, estimat;

φ - coeficientul frecvență de execuție (definit ca raportul dintre numărul mediu de execuții a unei instrucțiuni și numărul total de instrucțiuni);

d - numărul de erori detectate și 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 relațiile 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 și erori necritice; erorile critice sunt cele care conduc la oprirea misiunii în curs de execuție, în care caz sistemul este indisponibil. Compararea unui produs software în conformitate cu acest model este dată de graficul din fig. 10.4. Se fac următoarele notații:

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

λ - rata de detecție a erorilor;

μ_c - rata de corecție a unei erori critice;

μ_n - rata de corecție a unei erori necritice.

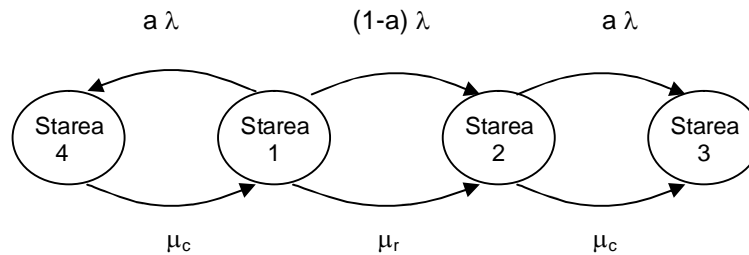


Fig.10.4. Graful de stare

Din graf se poate distinge existența a patru stări, care au semnificațiile:

starea 1 - nu sunt erori, programul funcționează conform specificațiilor sale;

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

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

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

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

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

$$A(t) = P_1(t) + P_2(t)$$

unde $P_1(t)$ și $P_2(t)$ sunt probabilitățile 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 conține un număr n de erori;
2. Fiecare eroare detectată este corectată, rata de detecție fiind ca și în cazul modelelor precedente proporțională cu numărul de erori rezultate;
3. Variabilele aleatoare, timpul de funcționare fără eroare și timpul de corecție a erorii sunt distribuite exponențial.

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

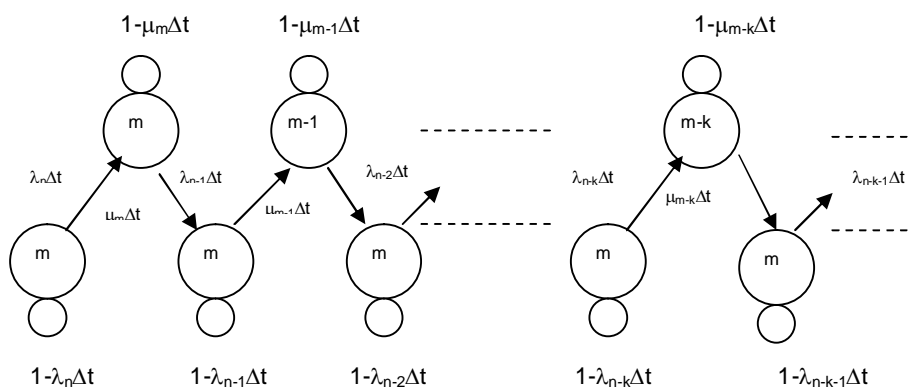


Fig.10.5. Graful de tranziție în cazul modelului V

Se pot defini:

- Starea $(n-k)$ - starea pentru care a fost corectată a $(k-1)$ eroare și nu a apărut încă eroarea numărului k ;

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

MODELUL VI

Se fac următoarele 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 mulțimea 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 funcție de formele posibile ale acestuia:

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

N_1 - reprezintă numărul de forme posibile ale aceluiași program în funcție 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 apariție a intrării i;

b_i - o funcție binară care ia valoarea "1" dacă programul i este eronat și "0" dacă programul este corect.

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

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

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

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

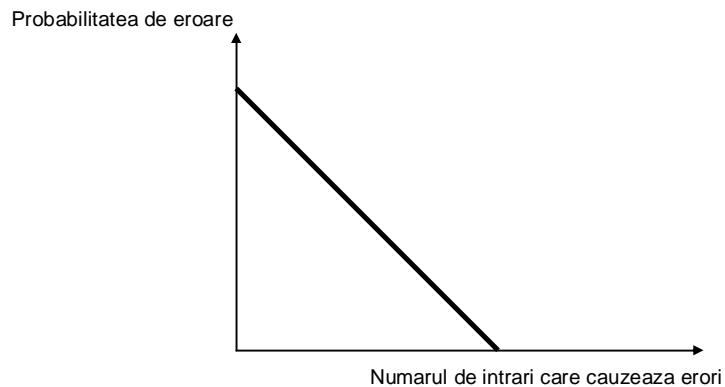


Fig.10.6. Probabilitatea de eroare

În ipoteza ca o eroare odată detectată nu mai apare în aceleași condiții, 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 notațiile:

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} și F_k pot fi calculate pe baza relației $F=e/N$, iar N_1 se estimează ca fiind panta dreptei din Fig.10.6, dreaptă care poate fi ridicată experimental în perioada de rodaj.

MODEL EMPIRIC DE STUDIU AL FIABILITĂȚII - MODELUL COMPLEXITĂȚII

Se consideră următoarele categorii de complexitate:

a) COMPLEXITATEA LOGICII, se referă la codul sursă al programului (număr de instrucțiuni executabile, număr de instrucțiuni de selecție, număr de instrucțiuni de iterație, etc.).

$$C_{\log} = \frac{I_t}{I_e} + C_{it} + C_{sel} + C_{salt}$$

I_t - numărul total de instrucțiuni;

I_e - numărul de instrucțiuni executabile;

C_{it} - complexitatea iterațiilor;

$$C_{it} = \sum_i m_i w_i$$

în care: m_i - numărul de cicluri de nivel i ;

w_i - factor de ponderare.

astfel încât $\sum_{i=1}^Q w_i = 1$

Q este numărul maxim de imbricări de cicluri în program

C_{sel} este complexitatea instrucțiunilor de selecție (IF)

$$C_{sel} = \sum n_i w_i$$

în care: n_i - numărul de instrucțiuni de decizie de nivel i

w_i - factor de ponderare

C_{salt} - număr de instrucțiuni de salt necondiționat.

b) COMPLEXITATEA INTERFERENȚELOR, se măsoară prin numărul de apeluri de rutine din (de exemplu, sistemul cognitiv și rezolutiv) sistem.

$$C_{int} = u + \frac{s}{2};$$

în care: u - numărul de apeluri de rutine din sistem

s - numărul de apeluri de rutine sistem.

c) COMPLEXITATEA INTRĂRII-IEȘIRII se măsoară prin numărul de operații de I/E:

$$C_{i/e} = \frac{I_{i/e}}{I_e} \cdot \frac{C_{rut}}{\sum_i i_{i/e}} \cdot I_{i/e}$$

unde:

$I_{i/e}$ - numărul de instrucțiuni de I/E;

$C_{rut} = \sum_i C_{log/rut}$ - suma complexității logicii tuturor rutinelor;

$\sum_i i_{i/e}$ - suma instrucțiunilor de I/E din toate rutinele.

d). COMPLEXITATEA CALCULULUI se măsoară prin numărul de instrucțiuni de asignare de valori ce conțin operatori aritmetici:

$$C_{calc} = \frac{c}{I_e} \cdot \frac{C_{rut}}{\sum_i c_i} \cdot c ;$$

c - numărul de instrucțiuni de calcul;

I_e - numărul de instrucțiuni executabile;

C_{rut} - suma complexității logicii tuturor rutinelor;

$\sum_i C_i$ - suma instrucțiunilor de calcul ale tuturor rutinelor.

e). CLARITATEA, se măsoară prin raportul dintre numărul de instrucțiuni cu comentariu și numărul total de instrucțiuni.

$$L = \frac{I_{com}}{I_t}$$

I_{com} - numărul de instrucțiuni cu comentariu;

I_t - numărul total de instrucțiuni

Complexitatea totală se definește astfel:

$$C = C_{log} + 0.1C_{it} + 0.2C_{calc} + 0.4C_{i/e} - 0.1L$$

În cazul unei complexități ridicate se consideră că fiabilitatea programului este scăzută.

10.6. Metode software pentru reducerea erorilor

Ceea ce este unanim acceptat este faptul că trebuie focalizate eforturile către depistarea și îndepărtarea defectelor care, în software se numesc de regulă erori. Un defect (o eroare) este orice lipsă din specificațiile de proiectare sau implementare a procesului (Grady, 1992).

Ingineria se străduiește nu numai să creeze și să îmbunătățească noi produse, ci să le producă cu costuri eficiente.

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

Ținând cont de faptul că există costuri relative mai mari decât 100 la 1, un beneficiu major al practicilor ingineresti este de a reduce costul și varietatea relărilor.

Cele mai multe evaluări ale calității software se fac de-a lungul activităților de testare.

Îngrijorarea managerilor se manifestă în general în acest stadiu și în consecință, unele dintre produsele finale care suportă îmbunătățiri în urma testării de-a lungul dezvoltării au suportat de fapt primele îmbunătățiri ale calității.

Criteriile utilizate pentru a judeca când s-a încheiat testarea sistemelor software sunt foarte largi și adesea subiective. Din cauza unei variații largi în ceea ce privește luarea în considerare a acestora în aprecierea calității software (permisă de o lipsă de standardizare), firma Hewlett- Packard (HP) a creat un set de criterii de măsurare a calității. Propunerea lor este de a certifica produsele software realizate și gata de livrare (Grady, 1992).

Cerințele 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 funcționalitatea pe faze;

⇒ Încurajează automatizarea unui număr cât mai mare de teste;

⇒ Mărește fiabilitatea produsului care funcționează la utilizatori.

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

Standardele de testare includ cerințele următoare:

- **întindere**: extinderea testării atât la ceea ce este accesibil utilizatorului, cât și peste funcțiile interne;

- **adâncime**: testare acoperitoare a ramificațiilor;

- **fiabilitate**: operare continuă sub "stress";

- **stabilitate**: abilitate de acoperire a condițiilor 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 și stabilitate. Fiecare din ele au cerințe ridicate de întindere, adâncime, operare continuă și densitate de defect. Aceste cerințe 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 și mai stabilă a erorilor care apar (Fig.10.7).

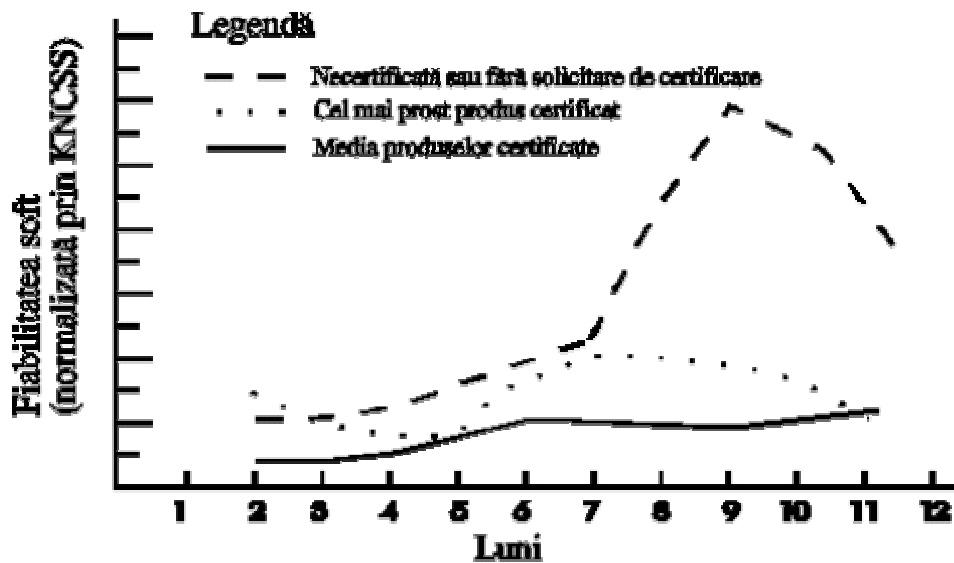


Fig.10.7. Spectrul tehnic al unui program

10.6.1. Inspecții, acoperire cu cod și complexitate

Procesul de testare implică de regulă numeroși ingineri, care trebuie să consulte, să semnaleze și corecteze sistemul. În contrast cu aceasta, inspecțiile pot fi startate chiar și de un singur inginer și, mai mult, există rezultate mai bine documentate pentru acest gen de activitate decât pentru orice altă practică din ingineria software.

Deși inspecțiile sunt în primul rând tehnici de detecție a erorii, experiențele în domeniu au arătat că și activitățile de pregătire a inspecției au drept rezultat prevenirea erorii. În cazurile în care "producția" de software nu se face într-o forma standardizată, pregătirea pentru o inspecție forțează o clarificare a cerințelor. Inspecțiile 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, **inspecția impune disciplină**. Trebuie create produse inspectabile la intervale regulate de timp, ceea ce va avea drept rezultat obținerea diferitelor imagini ale părților produsului final.

Produsele au din proiect, de regulă, mai multe părți, iar inspectarea acestora va produce un feedback măsurabil și obiectiv. În final, inspecțiile vor

constitui o cale de educare a echipei de software și de încurajare a celor mai bune practici, acestea fiind aspectele unei bune inginerii software.

O modalitate importantă de a păstra inspecțiile eficiente este de a produce rezultate măsurabile. Obiectul unei bune inspecții este de a găsi, mai repede decât pe alte căi, unde se afla hibe. Se poate măsura procentajul de defecte găsite și timpul în care au fost depistate.

De exemplu, o echipa HP a găsit câte un defect la fiecare patru ore de testare a sistemului. Rata de găsimă a defectului prin inspectarea codului a fost ca timp de 4,4 ori mai bună decât fără inspectare (Grady, 1992). Ei au reușit să evalueze dacă au ajuns la un punct de diminuare a răspunsurilor sistemului și, de asemenea, cât timp și efort au investit pentru aceasta.

Adesea inginerii software cred că au făcut toate testările posibile, când de fapt nu este așa. Reflectarea erorilor în codificare este măsurată prin utilizarea unui instrument care inserează instrucțiuni în codul sursă precompilat. Când se rulează testul, "extracodul" marchează segmentele care au fost executate și care nu.

Fig. 10.8 prezintă reflectarea erorilor în codificare pentru 22 de proiecte HP pe care realizatorii afirmă că le-au testat cu atenție.

Datele prezentate de acest grafic au fost foarte persuasive în a da o asigurare, atât managerilor cât și inginerilor, în ceea ce privește utilitatea măsurării codificării.

Astfel, s-a găsit că nu este foarte greu de a obține 80%-85% testare pentru codificare pentru un limbaj familiar programatorilor în timpul testării sistemului sau a integrării diverselor niveluri cu ajutorul unui instrument care identifică codul netestat.

Încă o dată trebuie făcută precizarea că această tehnică nu necesită o echipă mare de testare.

Una dintre metricile definite cu câțiva ani în urmă a fost complexitatea ciclică a software-ului, care derivă dintr-o analiză a "drumurilor" potențiale prin codul sursă. Aceasta este o măsură care anticipează unele dificultăți potențiale din timpul testării, semnalate de instrumentele de prezentare a codificării despre care s-a discutat anterior.

Este o metrică foarte utilă inginerilor software pentru semnalare modulelor complexe, care pot fi predispușe la erori și pot deveni greu de întreținut. Există de asemenea, instrumente comerciale care măsoară capacitatea combinând complexitatea cu imagini vizuale.

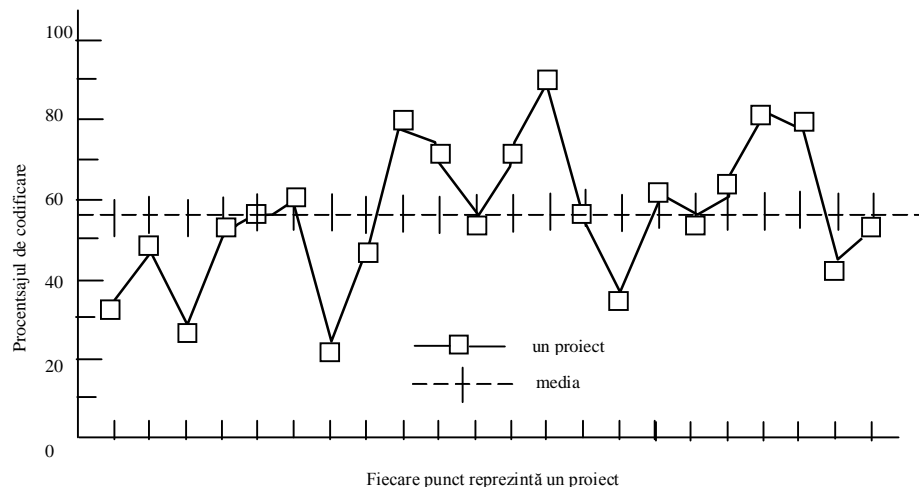


Fig.10.8. Reflectarea tipică prin codificare înainte de măsuratori

Rezultate încurajatoare în acest domeniu a obținut David Card în "Measuring Software Design Quality". El definește complexitatea proiectată ca o funcție a complexității structurale bazate pe răspândirea în formă de evantai a complexității datelor în funcție de numărul variabilelor de I/O și complexității procedurale în funcție de numărul de decizii. Rezultatele sale arată o foarte strânsă corelație între densitatea defectelor pentru modulele din opt proiecte.

O analiză similară pentru modulele unui proiect a arătat că, deși proporția pentru complexitatea individuală a modulelor diferă de cele din studiul lui Card, se poate găsi de asemenea o bună corelație a defectelor. În acest caz corelația s-a stabilit exact pentru componentele "în evoluție". Rezultatele sunt prezentate în Fig. 10.9.

Standardele de testare, inspecție și complexitatea ciclică, precum și relatarea prin codificare sunt exemple ale practicilor suficient de evaluate și demonstate pentru a putea fi luate în considerație de ingineria software.

Rezultatele prezentate în Fig.10.9. sunt încurajatoare și ele arată că este posibil să se utilizeze aceste metrici și în îmbunătățirea deciziilor proiectate.

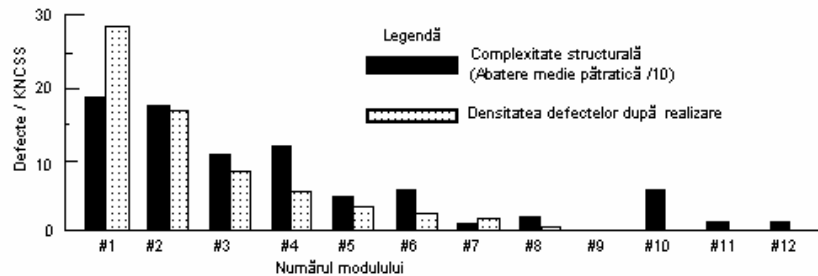


Fig. 10.9. Analiza defectelor prin codificarea unui modul

10.7. Utilizarea datelor eronate pentru îmbunătățirea deciziilor

La sfârșitul anului 1986 Consiliul de Metrici Software HP a făcut cunoscute definițiile categoriilor standard de cauze ale defectelor. Fig. 10.10 reprezintă modelul definițiilor prezentate în lucrarea lui Grady (1992). Modelul este utilizat în scopul selectării unui descriptor pentru origini, tipuri și moduri, pentru fiecare raport de defectare care este rezolvat. De exemplu, un defect ar putea fi o eroare de proiectare care face parte din interfața cu utilizatorul dar care lipsește ca descriere din specificațiile interne. Un alt defect ar putea fi o eroare de codificare, atunci când ceea ce este logic, este eronat.

Fig. 10.11 ne dă o idee a modului în care erorile variază de la o entitate la alta. Umbrele diferite reflectă partea de origine din fig. 10.10. Sectoarele de cerc care se decupează din zona de mijloc a fig. 10.10, indică tipul erorilor. Cele mai largi 8 surse de erori pentru diferite divizii HP sunt prezentate în fiecare sector de cerc. Toate cele 4 rezultate profilează defectele găsite numai de-a lungul testării sistemului și testării integrărilor în sistem.

Unele diferențe se datorează inconsecvenței cu care diferitele persoane utilizează definițiile. Deoarece definițiile au chiar scopul de a concentra procesul de îmbunătățire a eforturilor în domeniul costurilor mai scăzute de corectare, inconsistențele sunt rezolvate atunci când grupul definește cauzele hibelor și le

fixează printr-un "brainstorm". Din acest motiv datele prezentate în Fig. 10.11 sunt așa de importante. Continuând în timp marcarea erorilor se poate de asemenea măsura cât de bine sunt adoptate soluțiile.

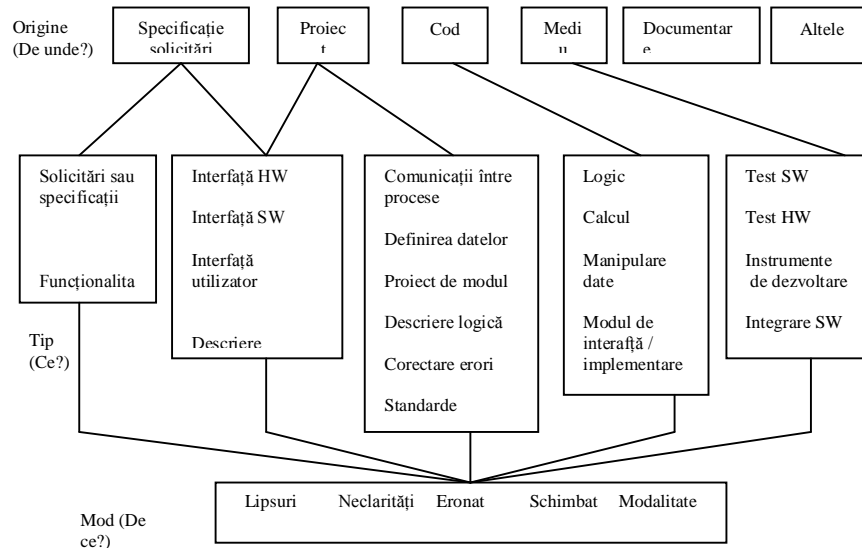


Fig.10.10. Categoriile surselor de defecte software

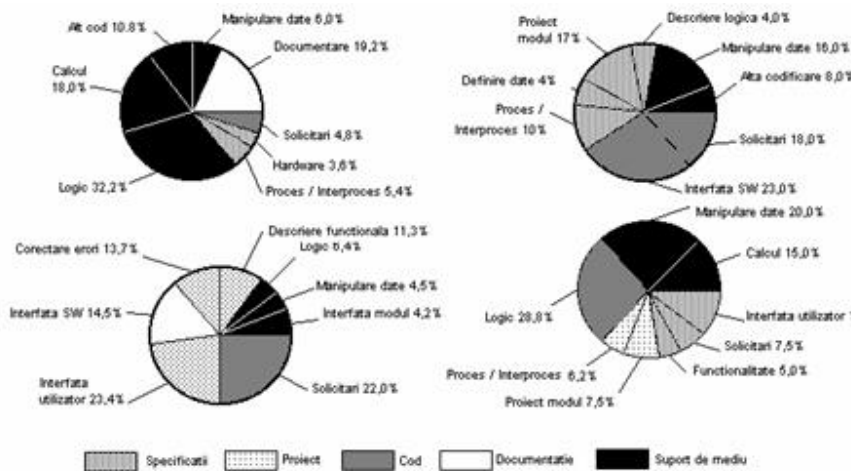


Fig.10.11. Sursele de erori pentru patru divizii HP (erori găsite în testare)

Aceasta abordare activă de identificare și de îndepărtare a proceselor care prezintă slăbiciuni constituie surse de neprețuit pentru a îmbunătăți calitatea. Se pare că există mai multe atribute care ajută în obținerea succesului în ceea ce privește calitatea software și anume:

- O diagramă conceptuală simplă, cum este cea din fig. 10.10 ajută evaluatorii să înțeleagă ce este cel mai simplu de făcut, și îi ghidează în vederea proporționării cantității de efort necesare raportărilor;
- Definițiile standard pentru tipurile de defecte ajută la rezolvarea pe diferite căi de raționament 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 acțiuni;
- Pregătirea în colectarea datelor despre erori și 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 îmbunătățire a acestei situații. Echipele HP, de care s-a mai amintit deja, au cules date pentru specificații, proiect și inspecțiile de cod. Toate acestea sunt prezentate în Fig. 10.12.

Trebuie manifestată puțină precauție 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 aceeași fază.

Barele verticale reprezintă ocaziile favorabile de a reduce, semnificativ, aceste surse de defecte. De exemplu, numărul mare de defecte dintr-un modul proiectat sugerează că ar fi nevoie să se înlocuiască tehnica de proiectare sau să se completeze metodele existente. 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 și ocazii de detectare mai precoce a erorilor.

De exemplu, solicitările, funcționalitatea și descrierea funcțională a defectelor se combină în a sugera că proiectele trebuie schimbate datorită unei definiri (specificații) inițiale neadecvate a produsului. În asemenea situații ar fi util de folosit prototipurile.

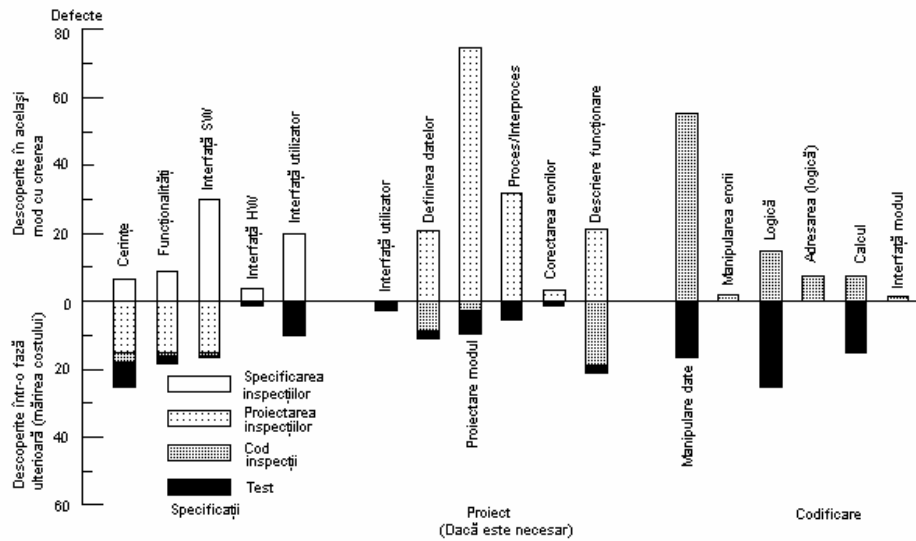


Fig.10.12. Profilul erorilor

Aceste tipuri de date creează posibilitatea de a lua decizii mai bine informate și 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 obținut 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 și efectiv s-a înlocuit cea mai mare parte a erorilor rezultând cele două produse B și C (Grady, 1992).

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

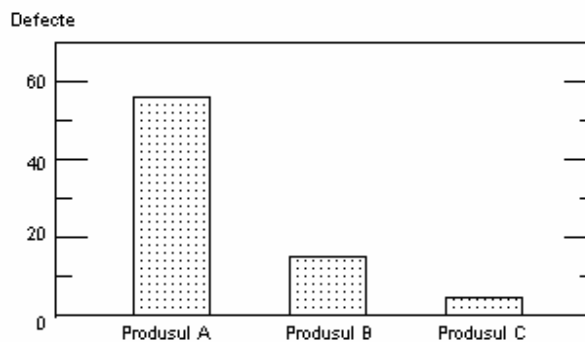


Fig.10.13. Erori de manevrare

Toate organizațiile trebuie îndrumate să adopte practici de inginerii software atât pentru considerente de afaceri, cât și pentru cerințe profesionale, aceasta fiind singura formulă de obținere a unui produs software cât mai aproape de necesitățile utilizatorilor.

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

Producerea sistemelor software de înaltă calitate pe scară largă în cadrul unor restricții de proiect și de buget a devenit o interesantă competiție în ingineria software. Modificarea acestor sisteme pentru a încorpora noul și posibilitățile de schimbare supun tot timpul sistemul la o competiție 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 și conducând la reducerea calității.

O abordare tradițională în dezvoltarea produsului software de înaltă calitate constă în aplicarea unei metodologii de dezvoltare, cu sublinieri accentuate pe detecția erorilor. Acest proces de detectare a erorilor constă într-o căutare continuă, prin inspectare și 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 și analiza datelor de defect software în ordinea identificării cauzelor lor. Din momentul în care cauzele sunt identificate, pot fi aduse îmbunătățiri proiectului pentru a preveni aparițiile viitoare ale erorilor.

O lucrare de specialitate de la firma IBM prezintă o metodologie de abordare a proiectului care utilizează analiza cauzală și feedback-ul ca mijloace pentru obținerea îmbunătățirii calității și, în final, prevenirea defectăunilor.

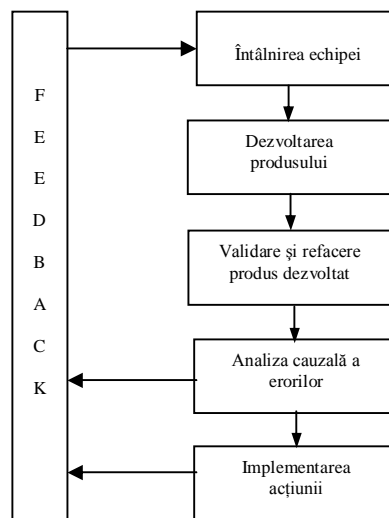


Fig.10.14. Procesul de analiză cauzală

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

- 1) Proiectanții și-au evaluat propriile greșeli;
- 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 activității echipei de analiză cu următoarele obiective:

- a) Revederea datelor eronate de intrare;
- b) Revederea liniilor directe din metodologie;
- c) Revederea listelor de verificare potrivite;
- d) Identificarea cerințelor echipei.

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

Produsele rezultate sunt apoi validate și 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) soluția (soluțiile) pentru prevenirea apariției erorii în viitor.

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

O altă echipă de analiză cauzală se întâlnește apoi pentru analiza datelor din baza de date. În plus, grupul poate avea nevoie la un moment dat să se consulte cu alte persoane din afara echipei (proiectanți, 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 folosește metoda de rezolvare a problemelor pentru: a analiza datele, a determina punctele asupra cărora trebuie să lucreze, cauzele grupurilor de probleme și pentru a dezvolta planuri de implementare și recomandări care să prevină apariția tipului sau tipurilor similare de probleme în viitor.

Aceste recomandări sunt apoi supuse unei acțiuni în echipă, care are următoarele responsabilități:

- a) Evaluarea și prioritizarea recomandărilor;
- b) Implementarea recomandărilor;
- c) Răspândirea feedback-ului.

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

Starea acțiunii este de asemenea păstrată în baza de date a analizei cauzale și monitorizată de acțiunile echipei.

Cele mai multe eforturi raportate la activitățile de analiză cauzală au fost focalizate pe dezvoltarea procesului. Aceasta reprezintă, din păcate, un procentaj ridicat de efort consumat în timpul activităților de modificare software. Modificările software apar pe măsura adăugării de noi posibilități sau pe măsura modificării celor existente ale sistemului. Modificarea în linii mari a unui sistem complex este o mare consumatoare de timp și o activitate susceptibilă la erori. Această sarcină devine și mai dificilă în timp, pe măsură ce sistemul se mărește și structurile sale se deteriorează.

Analiza cauzală

Pașii analizei cauzale sunt prezentați sumar în Fig.10.15.

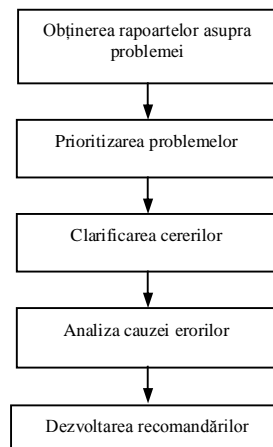


Fig.10.15. Analiza cauzală în procesul de modificare software

Pasul 1. Obținerea 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 creșterea priorității problemei și constă în împărțirea erorilor în trei categorii, și anume:

- 1) Critice;

- 2) Majore;
- 3) Minore.

Erorile critice slăbesc funcționalitatea și interzic testarea viitoare. Erorile majore slăbesc parțial funcționarea, 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. Deși clasificarea nu este neapărat necesară în efectuarea unei analize cauzale, datorită faptului că sistemele bazate pe cunoștințe nu sunt produse software obișnuite, nu este lipsit de interes o astfel de încercare, care să adauge la categoriile de erori din software-ul tradițional, clase noi, specifice sistemelor de inteligență artificială (Novac 1994).

Clasificarea erorilor furnizează de asemenea informații 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 cunoașterii;
- C) Erori de interfață 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 cerințelor în proiect, de-a lungul perioadei de modificare.

Sunt incluse proiectul la toate nivelurile, achiziția și structurarea cunoașterii, realizarea prototipului. Exemple tipice de erori de proiectare sunt:

A1) Erori logice: Condiții eronate logic în reguli, transpunere logică greșită a cunoașterii în reguli, etc.

A2) Erorile de calcul sunt legate de inacuratețea și greșelile făcute în implementare legate de operația de calcul, în special în cazul în care se folosesc cunoștințe care rezultă din algoritmi de calcul, cum ar fi algoritmi precompilați care studiază rezistența navei în cazul unui sistemului expert.

A3) Lipsa excepției de manipulare. Aceste erori includ greșeala de a lucra cu condiții unice sau cazuri unice chiar și atunci când există excepții și acestea au fost prevăzute în specificațiile 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 inferențe atunci când răspunsul sistemului trebuie să fie foarte rapid, luând în considerare numai cunoștințe de suprafață.

A5) Erori de manipulare a datelor. Aceste erori includ greșelile făcute la inițializarea datelor înainte de utilizare, utilizări improprii ale constantelor din sistem, control neadecvat al fișierelor de date adiționale sistemului, etc.

A6) Erorile în conceptele I/O includ forme de intrare sau ieșire în/din fișiere eronate, erori de formatare, definirea unor înregistrări de dimensiune greșită, protocoale de I/O eronate sau improprii dispozitivelor sistemului.

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

B). Erori provenite din validarea cunoașterii

Această categorie, specifică sistemelor bazate pe cunoștințe, se referă la erorile care apar în oricare dintre fazele de prelucrare a cunoașterii, începând cu achiziția primară a cunoștințelor și continuând cu mentenanța sistemelor (vezi 8.4 Validarea cunoașterii).

C). Interfață incompatibilă

Erorile legate de interfață apar atunci când interfețele 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 diferiți, față de cei care sunt așteptați;

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

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

4) Rutine de execuție sau alte rutine, inexistente;

5) Dezacord între software și hardware;

6) Dezacord între software și firmware (microprogramare) atunci când anumiți parametri trebuie obținuți prin microprogramare.

D) Sincronizarea incorectă dintre proiectele paralele

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

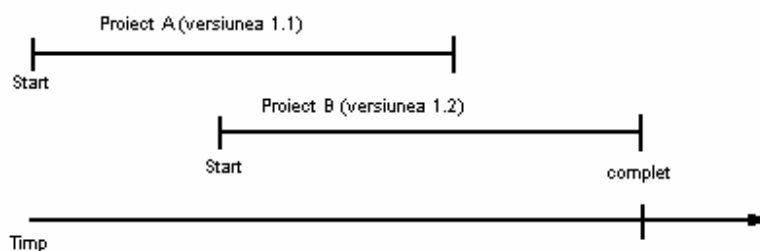


Fig.10.16. Evoluția sistemului software

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 mentenanță 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 (părți) de obiecte fac parte din această categorie.

F). Epuizarea resurselor sistemului

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

Pasul 4. Analiza și 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, proiectanții software care au introdus erorile trebuie intervievați, dar într-o manieră neoficială și cu foarte mare grijă pentru a nu provoca reacția lor de apărare. Analiza pentru prevenirea defectului trebuie subliniată clar ca fiind scop și prioritate în acest interviu. Bazat pe informația obținută de la proiectantul care a introdus eroarea se poate selecta o categorie cauzală de erori.

Considerăm că se pot identifica șapte categorii majore de cauze care conduc la erori în modificarea software-ului. Acestea diferă de altele prezentate în literatura de specialitate și 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 mentenanță.

Cerința 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 eficienței costului implementării recomandărilor care elimină sau reduc cauzele erorii.

Categoriile cauzale obișnuite în activitățile de modificare sunt:

I) Cunoașterea sistemului / experiența

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

- 1) Înțelegerea greșită (confuziile) a proiectului existent;
- 2) Înțelegerea greșită a procesului de modificare;
- 3) Înțelegerea neadecvată a mediului de programare. Mediul de programare incluzând personal, mașini, management, instrumente de dezvoltare și alți factori care afectează posibilitatea de dezvoltare și modificare a sistemului;
- 4) Înțelegerea neadecvată a solicitărilor clientului.

II) Comunicația

Această categorie cauzală reflectă problemele de comunicare în ceea ce privește modificările care trebuie efectuate. Se identifică cauzele care nu au fost atribuite lipsei cunoașterii sau experienței și care apar datorită comunicării incorecte sau incomplete.

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

1. Greșeala 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 implicațiilor posibile ale modificării software-ului. Un exemplu este omiterea unei codificări de acoperire a erorii după adăugarea unei noi piese hardware.

IV) Metode / standarde

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

V) Caracteristica desfășurării

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 susținere

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

VII) Eroarea umană

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

11

Evaluarea sistemelor software

11.1. Set de metrici software pentru conducerea proceselor de mentenanță a programelor

Problema evaluării sistemelor software a devenit cu atât mai acută și importantă cu cât dimensiunea, costul și locul strategic ocupat de sistem a crescut în timp.

Așa cum fiecărui produs i se atașează la livrarea către beneficiar un certificat de calitate, tot așa sistemele software, indiferent de tehnica de realizare (clasică sau din domeniul inteligenței artificiale), trebuie să fie garantate și studiate din punctul de vedere al fiabilității, calității și întreținerii (mentenabilității) lor.

Un studiu de specialitate (Stark, 1994) face cunoscute rezultatele cercetărilor unui grup însărcinat cu administrarea programelor din cadrul NASA, care a definit și implementat un set de metrici software conținând 13 metrici destinate să corecteze și să adapteze acțiunile de mentenanță.

Echipa numită MOD a răspuns de astfel de activități de mentenanță software la diferite niveluri (sisteme, subsisteme, module) utilizând paradigma solicitare (întrebare) metrică, care identifică o mulțime de 13 metrici pentru utilizarea curentă și de viitor în studiul sistemelor. Aceste metrici permit evaluarea stării curente a activităților de mentenanță și predicționează 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 mentenanță software precizează metricile pentru astfel de activități.

Tabelul 11.1 Concluziile privind paradigma cerere/întrebare/metrică

Cerere	Întrebare	Metrici
Maximizarea satisfacției clientului	Câte probleme afectează utilizatorul (clientul)?	- Raportul discrepanțelor (DR) și solicitările de service (SR) de-a lungul desfășurării. - Fiabilitate software - Raport întrerupere/ funcționare
	Cât timp durează pentru a definitiva o problemă?	- Terminarea DR/SR - DR/SR de-a lungul desfășurării.
	Unde sunt "îngustările"	- Utilizarea staff-ului - Utilizarea resurselor calculatorului
Minimizarea efortului și proiectului	Unde se consumă resursele?	- Utilizarea staff-ului- Planificarea SR- Instanțieri gen ADA- Distribuția 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 susținut (confirmat) efectiv inginerește?	- Densitatea erorilor - Raport întrerupere/ funcționare - Instanțieri ADA - Fiabilitate software

De asemenea, în lucrarea lui Gill (1991) se pune problema datelor care trebuie păstrate în timpul mentenanței sistemului, sau care determină activități de mentenanță (Zuse 1992).

Pornind de la acestea, se pot sintetiza problemele legate de activitățile de mentenanță a software-ului inteligent ca evoluând pe direcția "cerere- întrebare-metrici" (Tabelul 11.1).

Observații:

Metricile sunt utilizabile individual, dar cel mai mare beneficiu derivă din faptul că ele sunt folosite ca o mulțime de pași în solicitări de concurență cum ar fi calitate față de productivitate sau calitatea în raport cu datele realizate, etc.

Pentru a obține aceste beneficii se raportează mai mult de o metrică la fiecare cerere, iar alte metrici întrețin desfășurarea cererii (de exemplu fiabilitatea software). Aceste desfășurări pot furniza nu numai observații din interiorul proiectului, dar permit de asemenea verificări asupra consistenței datelor.

Șefii proiectului pot să combine metricile pentru a investiga pași 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 și calității software-ului și este strâns legată de alte metrici din mulțime (cum ar fi densitatea erorilor, complexitate, etc).

Metrica este definită ca numărul de linii de cod sursă (NLCS) care au fost întreținute în proiect. Aceasta poate fi utilizată în proiectul de management pentru următoarele activități:

Mulțimea de metrici pentru mentenanța software-ului

1a). Urmărirea efortului necesar pentru întreținerea codului. O creștere în dimensiune a software-ului poate conduce la menținerea unui personal mai numeros de întreținere. Dacă se poate stabili o relație între specificațiile proiectului, prin utilizarea unui model de cost sau a unei regresii liniare

simple, atunci mărimea privind schimbarea staff-ului de mentenanță poate fi predicționată prin schimbarea în dimensiune a software-ului.

1b). Anticiparea problemele de performanță în hard, în timp real. O tendință către creșterea dimensiunii software-ului ar iniția acțiuni corective care, fiecare, ar contoriza sau s-ar armoniza cu efortul crescut de depanare și/sau puterea calculatorului.

Fig.11.1 ilustrează dimensiunea software-ului unor sisteme MOD pentru 6 ani, între 1986-1991(Kern, 1992). Se înregistrează o medie de aproximativ 7,5% creștere pe an. Această evaluare poate fi combinată cu limbajul de programare (sau cu generatorul de sisteme expert folosit), care este ghid pentru numărul de NLCS pe care un programator trebuie să-l întrețină și cu numărul așteptat de erori din cod pentru a dezvolta programele proiectului, semnificativ pentru intensificarea, actualizarea și precizarea numărului de rapoarte cu discrepanțe care sunt de așteptat de-a lungul unui an.

De exemplu, dacă codul conține 1,5 erori/1000 LCS (KNLCS), în medie, atunci prin adăugare, $600 * 15 = 900$ erori ar fi înserate în soft.

Mai mult, dacă inginerul soft poate "suporta" 80 KNLCS (în medie), atunci solicitările proiectului vor fi de aproximativ 240 de oameni pentru a întreține aceste sisteme.

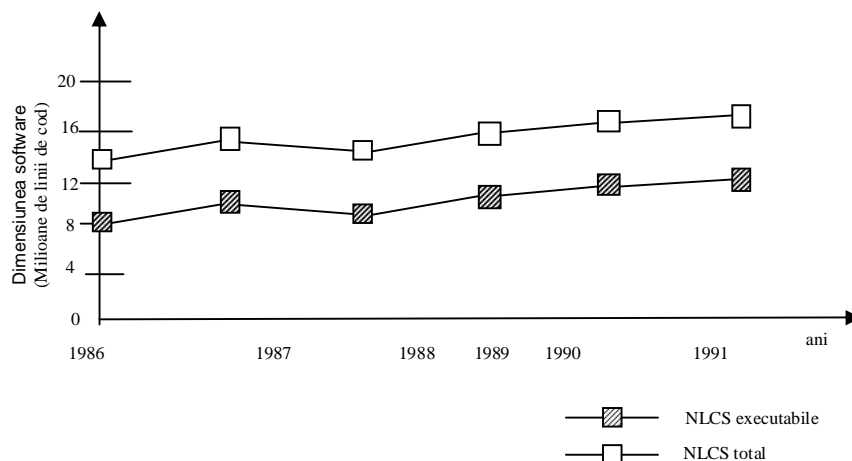


Fig.11.1. NLCS executabil (n) și total (o) pe an

2. Echipa de software

Schimbările în echipa de software au un impact direct asupra costurilor proiectului și asupra progresului în mentenanța software.

Această metrică este reprezentată de numărul de ore de-a lungul unei luni consumate de inginerii software direct implicați în activitățile de mentenanță și furnizează informații de management legate de datele care ar fi necesare pentru o predicție a viitoarelor cereri ale staff-ului proiectului.

De asemenea, poate fi folosită pentru următoarele acțiuni:

2a). Să estimeze eficacitatea în activitățile de inginerie software prin evaluarea numărului de pași și a resurselor necesare în fiecare proces de mentenanță. Este posibil să se continue sau să se elimine pași din proces, astfel încât să-l facă mai eficient și să răspundă mai bine utilizatorilor.

2b). Să identifice cele mai intensive resurse din activitățile de mentenanță (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 comparații cu regulile de acțiune recunoscute în industrie pentru cheltuiala cu personalul de-a lungul fazei de mentenanță, din ciclul de viață al sistemului software.

De exemplu, Fig. 11.2 prezintă situația pentru un proiect MOD.

Se observă că sistemul 5 consumă cele mai multe ore-persoană cu activități de mentenanță (evaluări SR, implementări și ș.a.m.d). Combinând aceste informații cu metrica de densitate, frecvență a erorii, s-a recomandat o schimbare în echipa de software a acestor sisteme. Orice sisteme care deviază semnificativ de la profilul planificat al proiectului va atrage atenția asupra sa.

3. Procesarea solicitării de mentenanță

Această metrică monitorizează fluxul activității de mentenanță software și determină nivelul de satisfacere a clientului.

Maximizând satisfacția utilizatorului, este important să se manipuleze solicitările cele mai vizibile pentru client, asigurându-i astfel o asistență permanentă.

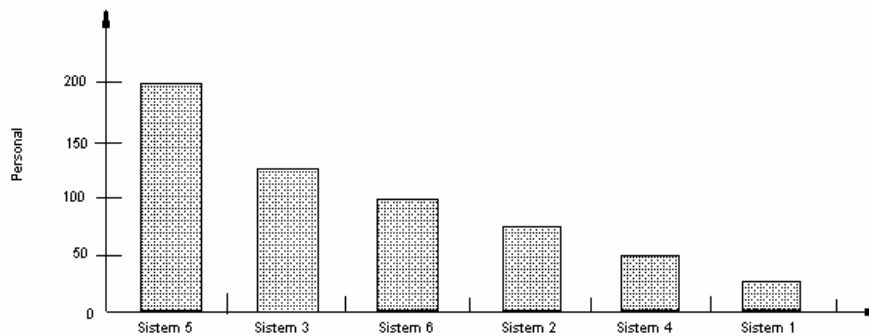


Fig. 11.2. Schimbări software cerute de eficiența desfășurării proiectului

Metrica permite și unui analist să execute următoarele funcții:

3a). Să facă o predicție asupra cantității 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 evoluția în echipa de management.

3c). Să efectueze studii de ocupare a personalului, a resurselor calculatorului, reducând posibilele rămășițe din modelul de simulare a evenimentelor discrete ale proceselor în cauză.

Fig. 11.3 dă un exemplu a acestei metrice pentru un proiect MOD. Linia continuă orizontală (Progres) reprezintă diferența dintre cererile incluse și nou sosite într-o perioadă de un an.

Această metrică reprezintă o bună cale de a observa tendința 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.

De exemplu, o linie a progresului care crește la +20 (~10%) justifică o investigație a motivului acestei creșteri a restului. Dacă linia descrește la - 20, atunci proiectul furnizează o descriere a metodelor folosite pentru a satisface un mare număr de solicitări.

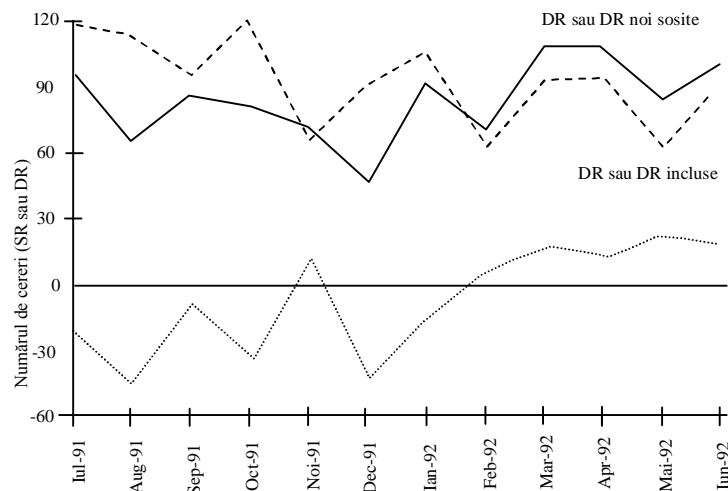


Fig.11.3. Schimbări software cerute de eficiența desfășurării proiectului

4. Planificarea mărită a software-ului

Metrica de planificare sporită a software-ului trasează mărimea timpului necesar pentru includerea unei cereri sporite și efortul ingineresc consumat pentru această solicitare. Sunt folosite două date primare pentru a calcula această metrică:

- 1). Numărul planificat și actual de ore de inginerie cheltuite cu această suprasolicitare;
- 2). Timpul calendaristic scurs de la apariția cererii până la rezolvarea ei, ca facilitate de utilizare.

Mulțimea de metrici include și această metrică deoarece permite managerilor proiectului să identifice producțiile de plan cu cel mai înalt risc, să evalueze timpul necesar, pentru a oferi utilizatorilor o cerere sporită și să predicționeze cantitatea de muncă pe care o incumbă această cerere.

De exemplu, când se primește o cerere de service i se asignează o prioritate (aceasta semnifică risc pentru plan, sistem, proiect), datele de care are nevoie și se estimează personalul necesar pentru satisfacerea completă a task-ului.

Trasând aceste estimări peste actuala desfășurare de date a solicitării adiționale, conducerea proiectului poate să-și modifice procesul estimat și să furnizeze utilizatorilor o informație mai bună în ceea ce privește 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, și anume: CPU, disc, memorie și utilizarea canalului I/O. Metricile CRU sunt raportate ca procentaje din capacitatea resurselor și furnizează un cadru pentru prezentarea concluziilor analizei rezultatelor sistemului în raport cu contractul (Kern, 1992).

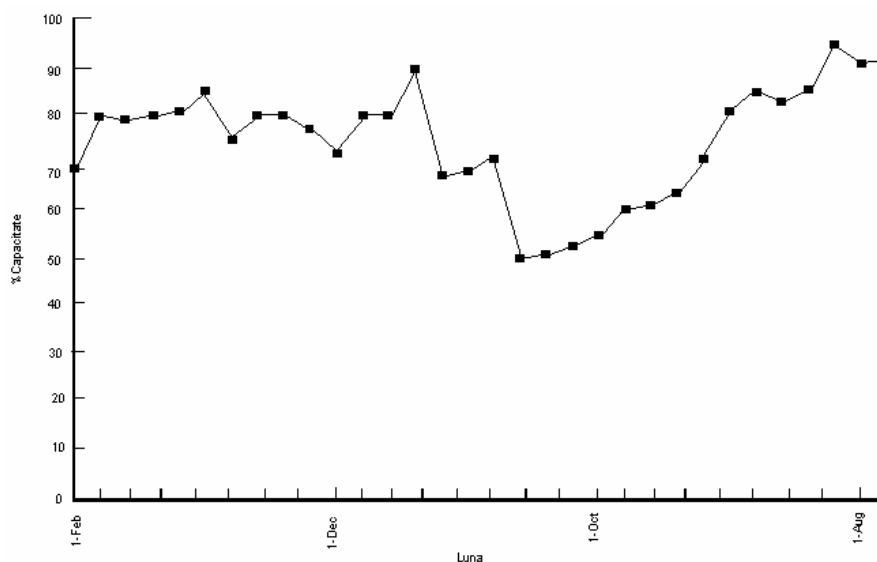


Fig.11.4. Utilizarea capacității de memorare pentru un proiect

De asemenea, avertizează mai devreme conducerea proiectului în cazul în care utilizatorii s-au apropiat de limitele de capacitate ale resurselor sistemului.

Modul în care sunt utilizate aceste resurse în cadrul unui proiect MOD sunt ilustrate în Fig.11.4. Când sistemul ocupă 90% din capacitate, proiectanții au înțeles că trebuie să îndepărteze din sistem fișierele perimate și să creeze

mai mult spațiu de memorare. Aceste acțiuni au redus procentajul la mai puțin de 50%.

Previțiunea că volumul de date memorate se apropie de capacitatea de 90% se potrivește cu o dreaptă de regresie de forma:

$capacitate = rata_de_ocupare * luna + constantă$, de la punctele datei până la ultima acțiune corectivă.

Această linie intersectează pragul de 90% capacitate în octombrie 1993, când a fost efectuată ultima acțiune corectivă.

6. Densitatea erorilor

Numărul rapoartelor de discrepante incluse într-un proiect cu un software fix, per KNLCS, în funcție de timp, definește metrica de densitate sau frecvența erorilor. Densitatea erorilor măsoară calitatea codificării și poate fi utilizată pentru:

6a). Predicția numărului de erori remanente în cod prin utilizarea unui model de calitate;

6b). Stabilirea densității standard de erori în scop de comparație și predicție pentru fiecare nivel sever de defectare sau tip de eroare. Aceasta permite proiectanților să identifice sistemele sau procesele cărora trebuie să li se acorde o atenție deosebită.

Fig.11.5 este o diagramă Pareto de densitate a erorilor pentru un sistem, timp de 5 ani, în cadrul unui proiect MOD. Acest grafic este un instrument util pentru "filtrarea" îmbunătățirii calității propuse a proiectului. Deoarece timpul și banii sunt resurse limitative, proiectanții au început să investigheze cauzele "defectărilor" după densitatea de erori raportată, în cazul sistemului A.

Aceste investigații conduc la concluzia că trebuie schimbate planurile, unele componente, sau reconstituit software-ul. Conducerea proiectului poate de asemenea să utilizeze aceste informații pentru a-și lua măsuri de prevedere atunci când ia în considerare cereri de extindere a sistemului, sau a acelor module în care s-au sesizat erori.

7. Volatilitatea software-ului

Belady și Lehman au definit în 1976, pentru prima dată "volatilitatea software-ului". Este un raport dintre numărul de module schimbate din cauza cererii de mentenanță și numărul total de module eliberate în timp.

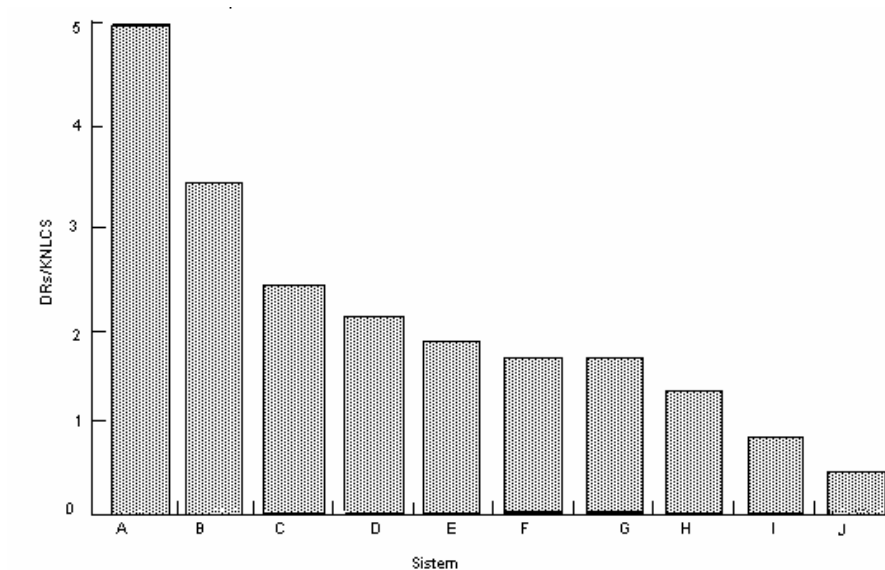


Fig.11.5. Densitatea erorilor pe sistem

Sarcina acestei metrici este de a măsura cantitatea de schimbări în structura software-ului, în timp. Utilizată împreună cu fiabilitatea, CRU și complexitatea proiectată, ea ajută managerii să decidă dacă o procedură calificată de test ar putea fi reexecutată și să identifice nevoia de reconstruire a software-ului.

Se pot stabili două reguli de bază pentru această metrică, și anume:

(1) Când volatilitatea depășește 30% este necesară o recalificare pentru o utilizare operațională;

(2) Dacă metrica depășește 80% pentru produsul realizat, sistemul va trebui reconstituit.

8. Durata raportului de discrepanțe

Această metrică (DR) oferă o măsură a timpului necesar pentru rezolvarea tuturor rapoartelor de discrepanțe din momentul descoperirii lor și până la încheierea proiectului.

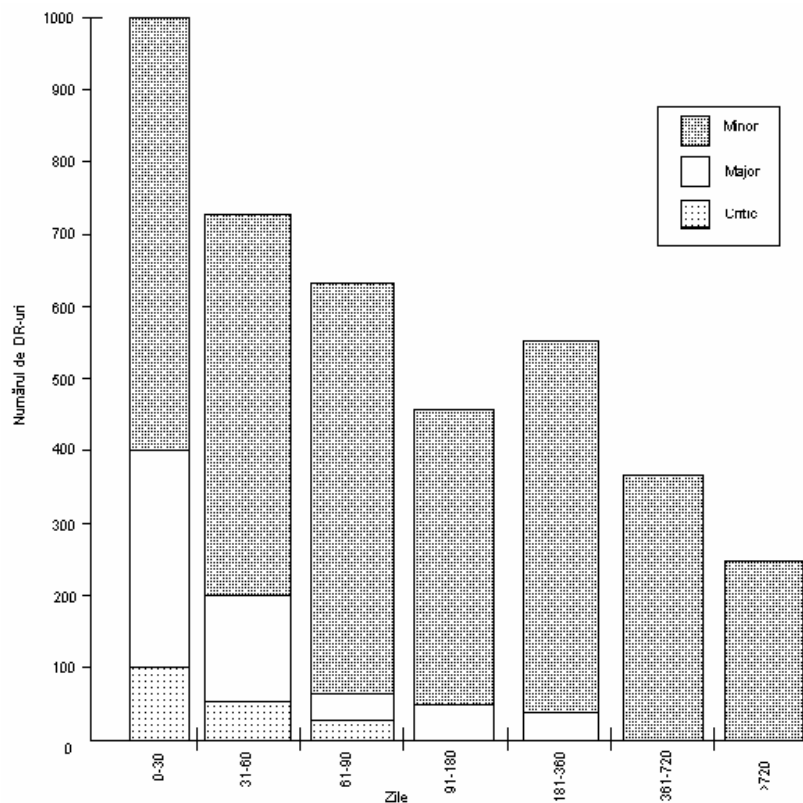


Fig. 11.6. Durata DR și zonele critice

Durata unei discrepanțe este calculată din punct de vedere al datelor raportate, minus datele supuse acțiunii și furnizează o privire din interior a eficienței 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 și 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 existența unor probleme dificile care necesită schimbări extensive pentru corecție, sau probleme care nu au fost

bine înțelese și solicită mai multă atenție. Fig. 11.6 este un raport de durată a unui DR simplu.

În plus, managerii responsabili cu procesele de discrepanțe păstrează data și durata vechilor rapoarte atunci când primesc altele noi.

Suma acestor "vârste" și ciclurile individuale de timp sunt utilizate pentru a identifica "îngustările" din domeniu și a îmbunătăți procesul.

9. Raportul întrerupere/funcționare

În lucrările de specialitate s-a comunicat că o schimbare în software are numai 20% șanse de succes dacă implică mai mult sau chiar 50 NLCS și aproape 50% șanse de modificări făcute corect, pentru mai puțin de 10 NLCS într-o primă încercare. Raportul întrerupere/funcționare este numărul de erori inserate în soft-ul operațional, în liniile de bază, împărțit la numărul de modificări făcute în produsul software.

De exemplu, dacă sunt trei discrepanțe care trebuie corectate și în urma corecției rezultă o eroare, raportul va fi 0,33. Aceasta înseamnă că activitatea de corecție a avut eficacitate 67% în rezolvarea DR-urilor. Metrica ajută de asemenea la execuția următoarelor funcții:

- 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 și dezvoltarea și aprobarea lui.

Aceasta este, de asemenea, necesară conducerii pentru urmărirea inspecțiilor care privesc codul și testarea proiectului.

Metrica raportului măsoară eficacitatea organizării mentenabilității în ceea ce privește modificările în software-ul operațional 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 și în condiții, de asemenea specificate.

Bazându-se pe datele operaționale de defectare, metrica "fiabilitatea software-ului" furnizează o indicație 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 menținută o rată acceptabilă a erorilor.

Dacă sistemul este realizat bine în ceea ce privește o "mulțime prag" de mentenabilitate, atunci vor fi permise schimbări complexe.

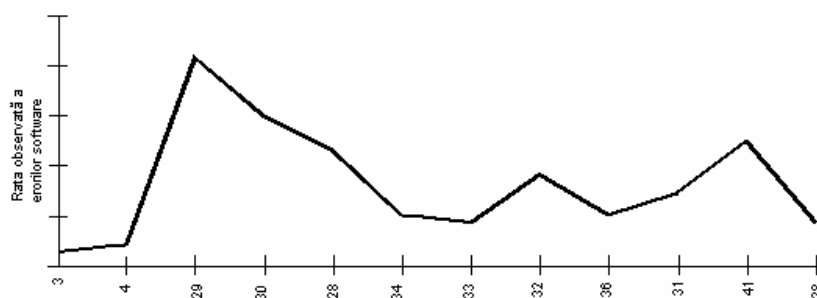


Fig.11.7. Rata erorilor software pentru 12 misiuni

Aproape toate proiectele au o cerință legată de fiabilitatea software, deoarece există un punct de la care rata erorilor din software face sistemul să fie inutilizabil.

Fig. 11.7 ilustrează forma ratei erorilor software într-o serie de misiuni de zbor a Centrului de Control NASA. Aceste date ajută la determinarea unui program acceptabil și stabilește cerințele viitoare.

11. Complexitatea proiectului

Complexitatea proiectului, ca metrică, indică numărul de module cu care complexitatea proiectului a crescut față de ceea ce s-a stabilit inițial. În acest caz se utilizează metrica complexității extinse a lui McCabe, care contorizează numărul de căi de execuție independente de-a lungul unei piese date de software, ceea ce înseamnă de fapt numărul de ramificații logice plus 1. De asemenea, ea permite conducerii proiectului să schițeze posibilitățile furnizorului

de a menține un nivel acceptabil al complexității, la nivelul fiecărui modul în parte. Complexitatea este strâns corelată cu efortul de mentenanță al programatorului și cu numărul de erori găsite de-a lungul testării și operării.

12. Distribuția tipului de erori

Această metrică prezintă raportul desfășurat de discrepanțe în trei moduri:

(1). Prin desfășurarea codului (aceasta înseamnă: hardware, software, resurse umane, neputință de a fi duplicat ș.a.m.d.);

(2). Prin tipul problemei care a fost greșită (adică logic, computațional, interfață, 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 și celor mai comune tipuri de erori introduse.

Informația provenind de la această metrică trimite înapoi în lanțul de dezvoltare și conducere, astfel încât managerii pot să ia efectiv măsuri de reducere a riscului (de exemplu o mai bună inspecție) și de asemenea să reducă tipul și cauzele erorilor. De asemenea, metrica este utilizată pentru identificarea solicitărilor pentru instrumente de mentenanță 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

Desfășurarea 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. Instanțieri tip ADA (limbajul de programare ADA)

Metrica "instanțieri tip ADA" reprezintă un număr de unități generice ADA și codificate de-a lungul activității de mentenanță, dimensiunea unei unități generice (în NLCS) și o mărime a timpului în care unitatea generică este instanțiată.

Cerința acestei metrici este de a marca numărul și dimensiunea componentelor reutilizabile dezvoltate de proiect pentru folosirea lor în interiorul proiectului sau în alte proiecte. Această metrică marchează unitățile generice ADA reutilizate și poate fi utilizată atunci când sistemele de inteligență artificială folosesc drept "cunoștințe" de profunzime algoritmi precompilați.

11.2. Programul de implementare a metricilor

După definirea mulțimii de metrici trebuie să se treacă la colectarea, analiza, raportarea și utilizarea metricilor.

Primul pas în implementarea lor este de a emite o directivă de management. Acest pas este considerat pozitiv deoarece asigură consistența de-a lungul proiectului și demonstrează că cel mai înalt nivel al proiectului, suportă efortul.

Importanța acestui suport nu poate fi exagerată; fără el metricile programului nu ar fi luate în serios, în particular de contractanți.

Al doilea pas este de a furniza instrumente automate pentru a susține metricile de program. Se utilizează spreadsheet-uri pentru a rezuma și raporta metricile datelor.

Observații:

Două dintre metricile de bază, și anume complexitatea proiectului și fiabilitatea software, sunt dificil de evaluat și calculat manual, și de aceea necesită instrumente automate (chiar gen sisteme expert) care să conducă competent astfel de activități..

Echipa MOD recomandă ca instrument "Modelarea statistică și estimarea funcțiilor de fiabilitate pentru software" (SMERFS) pentru măsurarea fiabilității software. Acest instrument include o varietate de domenii incluzând IBM-PC - urile și calculatoarele compatibile IBM-PC. Instrumentul de măsurare al

complexității este UX-metric / PC-metric de la SET Laboratories (1990). Acesta este un instrument comercial ieftin care contorizează NLCS, comentarii, linii albe și calculează metricile de complexitate pentru o varietate de limbaje de generația a treia incluzând FORTRAN, C și ADA. El lucrează pe stații și suportă sistemul de operare UNIX la fel de bine ca și pe PC-uri, IBM compatibile.

O astfel de activitate trebuie organizată sistematic, începând cu proiectarea, achiziția de cunoștințe și continuând cu implementarea, testarea și mentenanța sistemelor de programe pentru a putea "garanta" calitatea produsului software elaborat și a-i demonstra, și în această manieră, superioritatea față de programele tradiționale.

BIBLIOGRAFIE

- [Arno, 1991] Arnold P.S., S.Bodoff, D.Coleman, H.E.Gilchrist, F.M.Hayes: *An Evaluation of five Object-Oriented Development Methods*, JOOP, 1991, pp.107-121
- [Bain, 2008] Bain S., - *Emergent Design : The Evolutionary Nature of Professional Software Development*, Addison Wesley, March 2008, 448p., ISBN-0-321-50936-6.
- [Bass, 2003] Bass L., P. Clements, R. Kazman – *Software Architecture in Practice*, Addison Wesley Pub., April 2003, 560p, ISBN- 0-321-15495-9.
- [Beck, 2007] Beck K., – *Implementation Patterns*, Ed. Addison Wesley, November 2007, 176p,ISBN- 0-321-41309-1.
- [Beck, 1997] Beck K., – *Make It Run, Make It Right: Design Through Refactoring*, The Smalltalk Report, Vol.6, No.4, pp 19-24, SIGS Publications, January 1997.
- [Booc, 1996] Booch G., – *Object Solutions*, Addison Wesley, 1996
- [Booc, 2005] Booch G., J. Rumbaugh, I. Jacobson – *Unified Modeling Language User Guide*, Addison Wesley , 2005, 496p, ISBN- 0-321-26797-4.
- [Busc, 2007] Buschmann F., – *Pattern-Oriented Software Architecture, Vol.4: A Pattern Language for Distributed Computing* , Ed. John Wiley&Sons, May 2007, 636 p., ISBN-0-470-05902-8.
- [Busc, 1996] Buschmann F., R. Meunier, H. Rohnert, P. Sommerland, M. Stal – *Pattern-Oriented Software Architecture : A System of Patterns*, John Wiley&Sons, 1996.
- [Caya,2002] Cayannes C., B. Keeton –*Enterprise JavaBeans 2.0.*, Editura Teora, București, 2002
- [Coad, 1995] Coad P., D. North, M. Mayfield,-*Object Models: Strategies, Patterns and Applications*, Prentice Hall, 1995.

- [Cook, 1994] Cook S., J. Daniels,- *Designing Object Systems :Object –Oriented Modeling with Syntropy*, Prentice Hall, 1994.
- [Coop, 1997] Cooper, J. W.- *Principles of Object-Oriented Programming in Java 1.1 Coriolis(Ventana)*, 1997.
- [Coop, 1998] Cooper J.,W.- *The Design Patterns Java Companion*, Addison-Wesley, 1998, 218p.
- [Copl, 1995] Coplien J., - *A Generative Development Process Pattern Language*, Coplien & Schmidt, 1995, pp. 183-237.
- [Cosh, 1995] Coplien, James O. and Schmidt, Douglas C., *Pattern Languages of Program Design*, Addison-Wesley, 1995.
- [Doc MySql] Documentația MySQL - <http://dev.mysql.com/doc/>
- [DocJava16] Documentația Java 1.6
<http://java.sun.com/javase/6/docs/api/index.html>
- [Duva, 2007] Duvall P., – *Continous Integration : Improving Software Quality and Reducing Risk*, Addison Wesley, 2007, 336p, ISBN-0-321-33638-0.
- [Eclipse, Pr] Eclipse Project - www.eclipse.org/
- [Essent. Soft] *Essential Software Engineering-* www.rspa.com/spi/
- [Gamm, 1997] Gamma E., R. Helm, R. Johnson, J. Vlissides – *Design Patterns: Elements of Reusable Object-Oriented Software*, Ed. Addison Wesley, 1997, 395p, ISBN - 0-201-63361-2.
- [Grad, 1993] Grady, R.B .- *Practical Results from Measuring Software Quality*, Commun. ACM, Vol.36, No.11, Nov., 1993.
- [Grad, 1992] Grady, R.B. – *Practical Software metrics for Project Management and Process Improvement* , Prentice-Hall, New York, 1992, pp.17-18, 75-78, 139-140, 161, 171, 215, 232.
- [Grah, 1992] Graham M, E. Mettala – *The Domain –Specific Software Architecture Program*. Proceedings of DARPA Software Technology Conference, 1992, pp.204-210
- [Hanm, 2007] Hanmer R., – *Patterns for Fault Tolerant Software*, Ed. John Wiley & Sons, December 2007, 308p, ISBN-0-470-31979-8.

- [Fort, 2001] Forta B., R. Keer – *Dezvoltarea aplicațiilor JavaServer Pages*, Editura Teora București, 2001.
- [Fowl, 1997] Fowler M., - *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997
- [Fowl, 1999] Fowler M., K. Beck, J. Brant, W. Opdyke, D. Roberts – *Refactoring: Improving the Design of Existing Code*, Addison Wesley Pub. July 1999, ISBN-0-201-48567-2.
- [Fowl, 2003] Fowler M., – *UML Distilled – A Brief Guide of the Standard Object Modeling Language*, Addison Wesley, 2003, 208p, ISBN-0-321-19363-7.
- [Java2SDK] Java 2 SDK- *Standard Edition* –
<http://java.sun.com/products/jdk/1.4.1/>
- [J2EE Tutor] J2EE Tutorial- <http://java.sun.com/j2ee/tutorial>
- [Kosk, 2007] Koskela L., – *Test Driven : TDD and Acceptance TDD for Java Developers*, Ed. Manning, 2007, 470p., ISBN- 0-193-23985-0.
- [Kura, 1998] Kurata D., - *Programming with Objects*, Visual Basic Programmer's Journal, June, 1998.
- [Leve, 1992] Levendel, Y. – *Reliability Analysis of Large Software Systems Defect Data Modeling*, , IEEE Trans. Software Eng., No. 16, pp. 141-152, 1992.
- [Lisk, 1987] Liskov B.,– *Data Abstraction and Hierarchy*, The Conference On Object Oriented Programming Systems Languages and Applications, ACM, New York, USA, 1987, p.17-34, ISBN-0-89791-266-7.
- [MaOd,1996] Martin J., J.J. Odell- *Object-Oriented Methods: Pragmatic Considerations*, Prentice Hall, 1996
- [Mart, 1996] Martin R., *Design Principle and Design Patterns*, 1996
www.objectmentor.com/resources/articles/ocp.pdf

- [McGr,2001] McGregor J., D. Sykes – *A Practical Guide to Testing Object-Oriented Software*, Addison Wesley, 2001, 417 p., ISBN -0-201-32564-0.
- [Meye, 1994] Meyer B., – *Object Oriented Software Construction*, Prentice Hall, 1994
- [Nova, 1994] Novac, C. – *Sistem expert pentru asigurarea fiabilității complexului integrat de control al navigabilității pe timp de furtună*, Teză de doctorat, Galati, 1994.
- [Nova, 1999] Novac, C., *Inginerie software*, Ed. Tehnica, Bucuresti, 1999, 162 pagini, ISBN 973-31-1354-9;
- [Nova, 2008] Novac-Ududec, C., *Metode și tehnici pentru proiectarea sistemelor software*, Ed. Didactică și Pedagogică, București, 2008, ISBN 978-973-30-2094-3, 311 pag.
- [Nyga, 2007] Nygard M., – *Release It: Design and Deploy Production – Ready Software*, Pragmatic Bookshelf Pub., 2007, 326p, ISBN- 0-978-73921-3.
- [Pair Prog] *Pair Programming-* www.pairprogramming.com
- [Pree, 1994] Pree, Wolfgang, *Design Patterns for Object Oriented Software Development*, Addison-Wesley, 1994.
- [Pres, 1992] Pressman R., *Software Engineering. A Practitioner's Approach*, The 3th Edition, McGraw-Hill, 1992.
- [Riel, 1996] Riel, Arthur J., *Object-Oriented Design Heuristics*, Addison-Wesley, Reading, MA, 1996.
- [Rumb, 1994]Rumbaugh J. – *The life of an Object Model : How the Object Model Change DuringDevelopment*. Journal of Object-Oriented Programming, 7(1):24-32, March/April, 1994.
- [Rumb, 1996]Rumbaugh J., - *OMT Insights*, SIGS Books, 1996.
- [Shal, 2000] Shalloway A, J. Trott – *Design Patterns Explained: A New Perspective on Object-Oriented Design*, 2000, 358p.
- [Scrum Met] *SCRUM Methodology* – www.controlchaos.com

- [Shla, 1997] Shlaer S., S. Mellor – *Recursive Design of an Application Independent Architecture*, IEEE Software, Vol.14, No.1, 1997.
- [Shor, 2007] Shore J., S. Warden – *The Art of Agile Development*, Ed. O'Reilly Media, October 2007, 430p, ISBN- 0-596-52767-5.
- [Tudo, 2007] Tudosa L., – *Șabloane de proiectare software în realizarea unei aplicații OO* - Lucrare de disertație master, Departamentul de Calculatoare și Informatică Aplicată, Universitatea "Dunărea de Jos", Galați, 2007.
- [Vlis, 1996] Vlissides J., J. Coplien, N. Kerth, - *Pattern Languages of Program Design 2 [PloPD2]*, Addison Wesley, 1996.
- [Well, D. J.] Wells D. J., - *Ghid practic pentru programarea extremă* – www.extremeprogramming.org