Universitatea "Politehnica" din Timişoara
Facultatea de Automatică şi Calculatoare

Stoicu-Tivadar Vasile

CONTRIBUŢII LA CREŞTEREA GRADULUI DE REUTILIZARE AL PROGRAMELOR PENTRU CONDUCERE DE PROCES

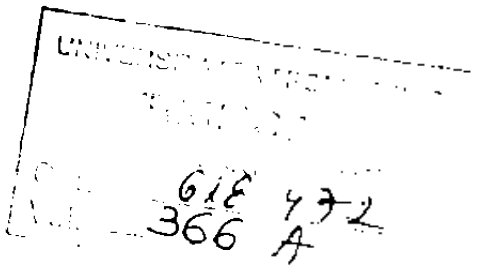
Teză de doctorat

Conducător ştiinţific :
Prof. dr. ing. Toma Leonida Dragomir

- Timişoara, 1997 -

Soției mele,

Lăcri



CUPRINS

Notaţii şi abrevieri utilizate în lucrare

Notaţii :

Capitolul 1

T - task
A - aplicaţie
P_i, P(i) - program
ISA - operator de generalizare
f_j - omomorfism
PG_j - program generic
Form(i) - *Formă*
Trans(i, Form(.)) - operator de trecere de la o formă la alta (proiectare)
P - entităţi de program
T - mulţimea continuă "timp"
obj(C) - clasă de obiecte
mor(C) - morfism

Capitolul 3

I - vocabularul unui program
L - lungimea observată a unui program
$\hat{L}$ - lungimea estimată a unui program
V - volumul programului
D - dificultatea programului
L_1 - nivelul programului
E - efortul
B - numărul de erori
T - timpul
C - complexitatea statică
C_g - complexitatea statică generalizată
η_1 - numărul de operanzi distincţi din program
η_2 - numărul de operatori distincţi din program
N_1 - numărul de incidenţe ale operatorilor
N_2 - numărul de incidenţe ale operanzilor
N_{p1} - totalul ponderat al operatorilor
N_{p2} - totalul ponderat al operanzilor
N_p - lungimea ponderată
V_p - volumul ponderat
E_p - efortul ponderat
IF4 - măsura "*information flow*"
r_T - gradul de reutilizare
$\hat{E}$ - "efort" de proiectare
M_r - metrică a reuzabilităţii
r_D - grad de reutilizare al proiectării
r_C - grad de reutilizare al codului
r_P - grad de reutilizare al programului
VIRT - virtualitatea medie
GENER - genericitatea medie
REUSE - reuzabilitatea medie
ICOMP - complexitatea medie de moştenire

Capitolul 4

r_P - grad de reutilizare de program
r_C - grad de reutilizare de clasă
$\sim$ ($<$, $\leq$, $=$, $\geq$, $>$) - operatori de comparare *fuzzy*
$P_{i,j}^{(k)}$ - taskul j de la nivelul ierarhic i , în etapa k de calcul
$\mu_X^{i,j}$ - valoarea funcţiei de apartenenţă X pentru taskul $P_{i,j}$
V - mulţimea variantelor (clasă de aplicaţii)
v - variantă
$\tilde{P}$ - submulţime *fuzzy* de proprietăţi
$\tilde{\Im}(V)$ - clasa tuturor submulţimilor *fuzzy* de proprietăţi
$pr_V(\tilde{\Im})$ - proiecţia mulţimii *fuzzy* de proprietăţi pe mulţimea variantelor
$\{\tilde{P}(t)\}_{t\in T}$ - submulţime *fuzzy* de proprietăţi dependentă de timp
$d(\tilde{P}, \tilde{Q})$ - distanţa *Hamming*
$d(\tilde{P}, \tilde{Q})$ - distanţa *Hamming* relativă
$r_{i,j}$ - gradul de reutilizare de program al programului j în raport cu programul i (*grad de reutilizare de la un program la altul*)
$R = \| r_{i,j} \|$ - matrice *fuzzy*
$\tilde{\Re}$ - relaţie *fuzzy*
$\Re_\alpha$ - nivelul de ordin α al relaţiei $\tilde{\Re}$
$r^n_{i,j}$ - normă reciprocă de reutilizare
$\hat{v}$ - clasă de similitudine (clasă de echivalenţă *fuzzy*)

Capitolul 5

C^{α}, (α) - clasă de aplicaţii
I^{α} - mulţimea indicilor elementelor din (α)
α - indexul specificator al clasei (α)
v_i^{α} - varianta i din clasa (α)

u_i^{α} - o intervenţie a proiectantului

U^{α} - mulţimea intervenţiilor proiectantului

$d(.)$ - operator de *proiectare* (*design*)

T^{α} - mulţimea *"timp discretizat"*

v_0 - program generic

$p^{\alpha}(v_k^{\alpha}, t_k)$ - operatorul *"performanţe"*

P^{α} - spaţiul performanţelor

$\mathcal{F}^{\alpha}$ - spaţiul calităţilor

ω^{α} - funcţii de intrare admise

γ^{α} - funcţii de ieşire admise

Ω^{α} - clasă de funcţii de intrare admise

Γ^{α} - clasă de funcţii de ieşire admise

s^{α} - operatorul de tranziţie în spaţiul calităţilor

c_k^{α} - vectorul calităţilor programului k

(t_k, v_k) - fază

p^{α} - operator de *performanţe generale*

e^{a}_{p} - efort relativ de acces

$w^{a}_{p.}$ - efort absolut de acces

n_{v0} - numărul de linii-sursă al programului generic

r^{a}_{p} - grad de accesibilitate de program

e^{a}_{c} - efort mediu relativ de acces

w^{a}_{c} - efort mediu absolut de acces

r^{a}_{c} - grad de accesibilitate de clasă

$r_c^{a,(cal)}$ - grad calibrat de accesibilitate de clasă

f^{cal} - factor de calibrare de acces de clasă

$s_k^{f,a}$ - viteza de fază a proiectării

$t_{k,k-1}$ - timpul fizic (concret) de realizare a proiectării

$s^{g,\alpha}$ - viteza de grup a proiectării

t_{car} - timpul caracteristic de proiectare

$s^{g,\alpha}_{med}$ - viteza medie de grup a proiectării

ε - deviaţie de clasă

ε_C - deviaţia de consistenţă

ε_0 - ecart de omogenitate

ε_m - eroare de merit

$Ker^{\varepsilon_m}(C^{tl})$ - nucleul unei clase relativ la eroarea de merit ε_m

Abrevieri :

ABS - administrator al bazei software
BCR - blocuri de construcţie reutilizabile
BIOS - *Basic Input-Output System* (Sistem de Intrări-Ieşiri de Bază)
BS - bază software
CASE - *Computer Aided Software Engineering* (Ingineria Programării Asistată de Calculator)
DFD - *Data - Flow Diagram* (diagrama de flux de date)
GUI - *Graphical User Interface* (interfaţă grafică utilizator)
IDE - *Integrated Development Environement* (v. **MID**)
IPR - ingineria programelor reutilizabile
PG - program generic
POO - Programare Orientată pe Obiecte
MID - Mediu Integrat de Dezvoltare
MR - modele reutilizabile (*reusable patterns*)
RS - reguli asociate de specializare
S - specializare *(reusable building blocks)*
SGBS - sistem de gestiune a bazelor software
SPC - sistem de proiectare în clasă
SQR - *Structured Query Language* (Limbaj Structurat de Interogare)
TAD - tip abstract de date
TOO - Tehnici Orientate pe Obiecte
UF - utilizator final

Cuvânt înainte

O problemă esențială a realizării produselor software o constituie adaptarea tehnologiilor de realizare la presiunea pieței. Se observă, pe plan mondial, o tendință de deplasare a efortului de realizare dinspre proiectanții de software cu înaltă calificare, spre "utilizatorii finali".

Una din direcțiile acestui efort este **reutilizarea**, prin aplicarea diverselor tehnici și artificii practice, atât pentru proiectarea unor părți de aplicații, cât și a unor aplicații întregi care vor suporta modificări pentru a fi adaptate la necesități nu foarte diferite de cele vechi. Aceasta înseamnă că aplicațiile noi vor fi deja parțial testate în momentul finalizării etapei de programare, "moștenind" pentru fragmentele reutilizate, parcurgerea întregului ciclu de viață, inclusiv punerea la punct. Evident, aceasta atrage după sine creșterea productivității, atât prin scurtarea etapei de programare cât și prin reducerea etapei de testare. Pe de altă parte astfel efortul de proiectare este împins dinspre programatorii înalt calificați - creatorii modelelor originale - spre utilizatorii finali.

Alternativa la programarea structurată, paradigma **POO**, ca și model excelent pentru aplicarea consecventă a unor concepte parcă anume definite pentru reutilizare, nu va fi tratată în lucrarea de față, decât ca și model pentru preluarea unor idei. Astfel, diverse tehnici de reutilizare prezentate în lucrare vor prelua, sub o formă simplificată, trăsături ale **POO**.

A treia posibilitate de deplasare de efort spre utilizatorul final o constituie folosirea *generatoarelor de aplicații*, fie că acestea folosesc un metalimbaj de specificații, fie că au o interfață clasică cu meniuri și cutii de dialog sau folosesc programarea grafică.

Lucrarea de față tratează câteva aspecte legate de cele enunțate mai sus : *exprimarea gradului de reutilizare*, unele *modele și concepte teoretice* pentru *formalizarea* procesului de proiectare pentru reutilizare, diverse *tehnici de creștere a gradului de reutilizare*, cu exemplificări practice, precum și unele considerații despre *generatoarele de aplicații* și *despre instrumentele CASE*, cu referire la facilitățile acestora orientate spre reutilizare. Cele prezentate în lucrare se referă în special la aplicații de conducere de proces *"mici"*. În ceea ce privește evaluarea software orientată spre reutilizare, lucrarea tratează aspecte legate de aplicațiile de conducere de proces, multitasking.

În legătură cu domeniile abordate, se poate pune întrebarea: de ce nu s-a insistat asupra alternativei **POO** ? Este adevărat că în prezent paradigma **POO** este larg aplicată, însă lucrarea de față vizează mai ales domenii unde aplicarea metodelor specifice **POO** este mai dificilă

aplicații mici, cu suport hardware cu resurse reduse, eventual cu microcontrollere. Iată un exemplu: dacă pentru o aplicație mare (spre exemplu o aplicație de telemecanică cu multe sisteme locale de interfață) se pot folosi un limbaj, un mediu și o metodologie care suportă **POO**, pentru un cuptor cu microunde este mai avantajos să se scrie o aplicație în limbaj de asamblare în așa fel încât din același program, prin modificări minime, să se poată realiza oricare din variantele seriei de cuptoare.

Lucrarea de față își propune să abordeze, **teoretic și practic**, domeniul acesta al **ingineriei programării** care se referă la **proiectarea pentru reutilizare**, cu specificul amintit. Această abordare se concretizează prin dezvoltări teoretice menite a clarifica o serie de concepte ale domeniului studiat și prin prezentarea unor principii și programe care sunt destinate a contribui la îmbunătățirea activității de proiectare pentru reutilizare în domeniul aplicațiilor de conducere de proces.

Astfel, **Capitolul 1** realizează o introducere în domeniu, pentru mai buna "așezare" a contribuțiilor originale. Studiul bibliografic sintetizează principalele aspecte care pot ajuta la conturarea preocupărilor lucrării. Punctul central al abordărilor lucrării de față este constituit de gruparea aplicațiilor asemănătoare în *clase de aplicații*. **Capitolul 2** realizează o sinteză a conceptelor **POO**. **Capitolul 3** realizează o deschidere spre domeniul **măsurilor software**. **Capitolele 4** și **5** aduc cele mai multe contribuții **teoretice** la domeniul abordat, prin prezentarea unor dezvoltări originale, prin perspectiva **logicii fuzzy**, respectiv a **abordării sistemice**. Astfel, sunt prezentate **noi măsuri software**, extrapolări de **teorii despre relații fuzzy**, **un model sistemic de tip intrare-stare-ieșire** (cu analiza consecințelor teoretice și practice respective), etc. **Capitolele 6, 7** și **8** cuprind o sinteză asupra experienței **practice** a autorului, în domeniul lucrării, din perioada activității sale din cercetare-proiectare, respectiv, din perioada activității didactice. Capitolele respective tratează **tehnici și structuri de date** pentru aplicații de proces mici, concepute special pentru creșterea gradului de reutilizare, **generatoare de aplicații pentru sisteme mici de instrumentație**, respectiv instrumente **CASE** pentru sisteme multitasking (incluzând și facilități de evaluare) și sisteme de instrumentație mici. Lucrarea cuprinde și un capitol de concluzii (**Capitolul 9**), o bibliografie și anexe.

*Considerăm că lucrarea de față aduce o modestă contribuție la dezvoltarea unor noi abordări teoretice (**abordare fuzzy**, **abordare sistemică**, noi **metrici software**) și a unor **tehnici** și **instrumente** practice pentru sisteme mici de conducere de proces și sisteme multitasking.*

CAPITOLUL 1.
CLASELE DE APLICAȚII ȘI PROIECTAREA VARIANTELOR

1.1. Aspecte generale

În multe arii de aplicații, se pot delimita **clase de programe** care pot fi identificate prin aceea că fiecare program aparținând unei clase poate fi considerat ca o variantă **specializată** a unui **program generic** [Mit87]. Cel mai des, problema delimitării unor astfel de clase se pune în cazul acelor producători de produse-program care realizează aplicații în mai multe variante, pentru diverși utilizatori. același aspect se întâlnește în etapa de întreținere a programelor, când, datorită acestei activități , apar versiuni noi, adaptate la necesități ivite pe parcursul exploatării aplicațiilor.

În altă ordine de idei, unul din obiectivele fundamentale ale ingineriei programării este realizarea și creșterea adaptabilității programelor. O posibilă soluție pentru atingerea acestui obiectiv este definirea unor programe generice din care să poată fi obținute cu efort minim de proiectare, adaptări pentru situații particulare.

Trecerea, în cadrul unei clase, de la un membru la altul, situația uzuală din proiectarea software, de refolosire a unor module de program din aplicații finalizate pentru realizarea altora, noi, precum și adaptarea unei aplicații la necesități noi constituie, toate, aspecte ale **reutilizării programelor**. **Reutilizarea** este o abordare sistematică a procesului de proiectare, care are ca scop transpunerea în practica proiectării a obiectivului prioritar al ingineriei programării, **adaptabilitatea.** Această abordare are ca rezultat conferirea unei trăsături specifice, produselor-program, **reutilizabilitatea**.

Reutilizarea se exprimă cantitativ prin **gradul de reutilizare**. Acesta este prezentat ca și măsura în care un cod și o structură aparținând unui program se folosesc la proiectarea unui program nou [Mit87].

Lucrarea de față își propune abordarea reutilizării software în cadrul unui domeniu particular al aplicațiilor software: conducerea de proces. De aceea, toate consideratiile referitoare la reutilizare vor fi particularizate pentru acest domeniu, iar exemplele și aplicațiile practice vor fi cu referire la specificul astfel delimitat. În proiectarea aplicatiilor de conducere de proces, se pot de asemenea ușor delimita clase de aplicații, spre exemplu : telemecanică, instrumentație, comenzi numerice, robotică, acționări electrice, etc. Problemele generale de reutilizare sunt valabile și în acest domeniu de aplicații software, cu un plus de dificultate datorită particularităților de timp real și interfatare cu procesul.

*Acest prim capitol are rolul de a investiga domeniul abordat, în vederea corectei situări a contribuţiilor tezei propuse şi a fundamentării teoretice a acesteia. De aceea, acest capitol va cuprinde definiţii, concepte de bază, o trecere în revistă a mai multor puncte de vedere, o abordare a unor aspecte specifice (generatoare de aplicaţii, instrumente **CASE**), aspecte generale (avantaje, probleme, clasificări, etc.), modele folosite, etc. Capitolul introductiv se încheie cu încadrarea structurii tezei în domeniul abordat, respectiv, cu prezentarea structurii lucrării, care decurge de aici.*

1.2. Reutilizabilitatea

Putem prezenta, intuitiv, **reutilizarea** software ca fiind abordarea coerentă şi voluntară a proiectării software astfel încât realizarea proiectelor software să presupună preluarea în cât mai mare măsură a rezultatelor activităţilor anterioare de proiectare. În această viziune, reutilizarea este asociată unei strategii generale care presupune eforturi individuale, organizaţionale, etc. Termenul de reutilizare poate fi discutat în trei accepţiuni : pentru **module de program**, pentru **personal** şi pentru **proiecte** [Mey88]. În cele ce urmează, în cadrul lucrării ne referim doar la prima şi la ultima accepţiune.

Trăsătura specifică a programelor, care reflectă posibilitatea reutilizării pentru proiecte viitoare, se numeşte **reutilizabilitate**. Această trăsătură este un vechi deziderat al programării. Încă din anii '60 se considera că realizarea ideală a produselor-program ar fi o activitate de asamblare a unor module deja existente. Este evident că modularitatea software este o condiţie de bază a reutilizării. Respectarea riguroasă a unor principii de programare precum compunerea modulară, generalitatea, minimalitatea, duc la îmbunătăţirea reutilizabilităţii [Fre94].

Minimalitate înseamnă evitarea redundanţelor în proiectare. Astfel, se pot concepe ierarhii de module cu aceeaşi funcţionalitate, în aşa fel încît acestea să acopere diversele alternative de implementare dar în condiţiile precizării o singură dată ale caracteristicilor comune ale modulelor. Principiul acesta este strâns legat de conceptul de moştenire, din **Programarea Orientată pe Obiecte (POO)**.

Posibilitatea reunirii facile a unor module de program, în vederea realizării unui program (**compatibilitatea**), se asigură prin respectare anumitor convenţii, referitoare, spre exemplu, la standardizarea intrărilor/ieşirilor, a structurilor de date şi a interfeţelor utilizator. Modulele care implementează asemenea cerinţe sunt proiectate independent de contextul de aplicare.

Reutilizabilitatea este prezentă în nomenclatura **Standard IEEE** citată în [Ter94], şi "definită" drept răspuns la întrebarea : "cât de uşor este de a converti un program dat în vederea utilizării în altă aplicaţie". Aici, ca factori care influenţează reutilizabilitatea sunt citaţi : **perceptibilitatea, modularitatea** (pentru descrierea acestor trăsături, a se vedea [Dav86]) şi independenţa software.

1.3. Ingineria programelor reutilizabile (IPR)

Proiectarea pentru reutilizare este un obiectiv valabil pentru cel puțin 20 ani, afirma Freeman în 1987, în [Fre87] (noi am adăuga că, de la răspândirea **POO**, perioada aceasta de valabilitate are o extindere cvasi-imprevizibilă) și care necesită dezvoltarea unei tehnologii adecvate : **ingineria programelor reutilizabile**. Costul proiectării este factorul hotărâtor în dezvoltarea acestei noi tehnologii, dar cerințele de calitate au început să devină de asemenea de egală importanță. **IPR** este o cale de a atinge aceste deziderate.

În Fig. 1.1 se prezintă conceptul fundamental al **IPR**. Ideea de bază este realizarea proiectării astfel încât să fie luată sistematic în considerare experiența dobândită la realizarea unor proiecte precedente (de aceea apar ca intrare pentru procesul de proiectare, informațiile despre activitatea/activitățile precedente de proiectare), iar ieșirile sunt utilizabile imediat (respectiv, programul care rezultă, care are o valoare de întrebuințare imediată) sau și pentru viitoare cicluri de proiectare, pentru dezvoltări de viitoare programe. În figură, intrările indicate prin săgeți verticale corespund specificațiilor, iar săgețile orizontale desemnează intrări legate de aspecte propriu-zise de proiectare.

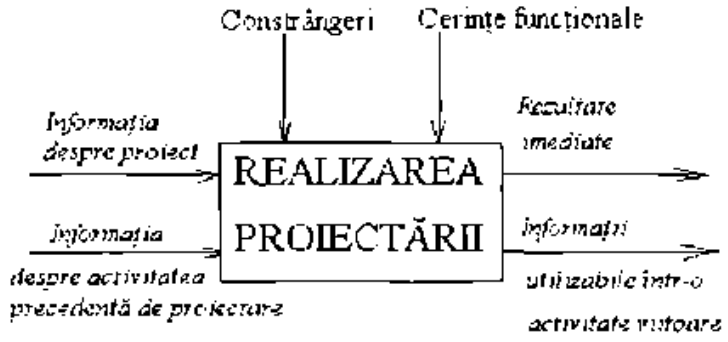


Fig. 1.1. Ideea centrală a **IPR**

Orice activitate de proiectare software (analiza cerințelor, realizarea proiectului programului, testarea proiectului, etc.) poate fi astfel organizată încât să permită și să favorizeze două subactivități specifice : utilizarea explicită a informației din precedenta realizare a activității și crearea informației noi, care să poată fi reutilizată într-o dezvoltare ulterioară aceleiași activități.

Obiectivele primare ale **IPR** sunt de a reduce costul ciclului de viață al programelor și de a crește calitatea aplicației. Aceste obiective includ scopurile specifice ale proiectării pentru reutilizare atât pentru partea de cod de program cât și pentru informațiile despre domeniul problemei pe care o rezolvă programul (respectiv, partea de specificații). În acest context, **IPR** este o parte a ingineriei programării, care urmărește promovarea cu consecvență a principiilor de reutilizare a programelor. Proiectarea trebuie astfel realizată încât să permită atingerea obiectivelor mai sus enunțate, evitându-se pe cât posibil activitatea redundantă și amortizând costurile unei proiectări inițiale mai sofisticate, prin generarea unui număr cât mai mare de aplicații.

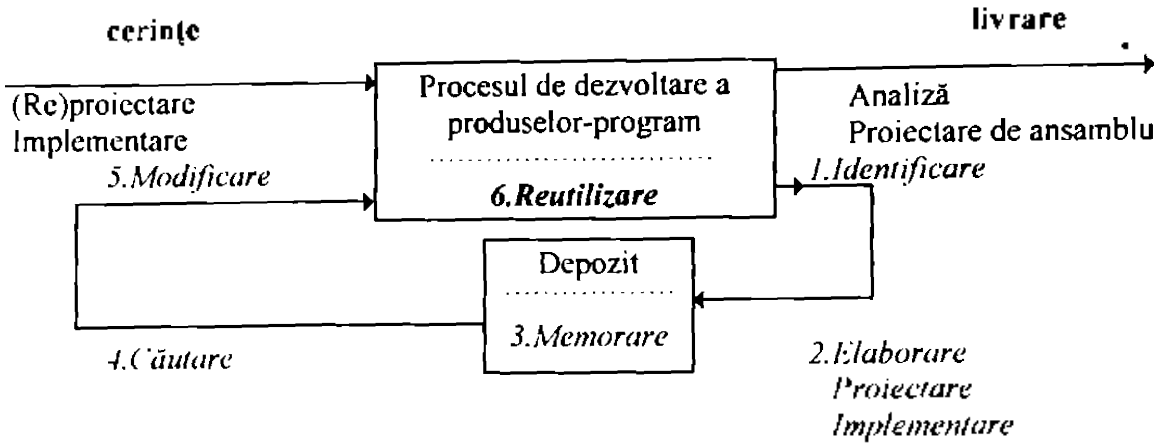


Fig. 1.2. Reutilizarea în procesul de dezvoltare software
(cu *italice*, acțiunile implicate în procesul reutilizării)

Diagrama din Figura 1.2 [Kru90] prezintă activitățile specifice procesului de construire a produselor-program (care se încadrează în așa-numitul ciclu de viață al programelor), într-o schemă care înglobează însă și modalitatea de a reutiliza fie module de program, fie proiecte deja finalizate. Diagrama aceasta constituie o dezvoltare a ideilor care au stat la baza schemei din fig. 1.1. În diagramă sunt evidențiate etapele "clasice" ale ciclului de viață al unui produs-program (analiză, proiectare, implementare, etc.). Procesul de proiectare folosind reutilizarea este împărțit în 6 etape (numerotate de la 1 la 6 în figură) - identificarea componentelor reutilizabile, elaborarea modulului reutilizabil în vederea stocării, memorarea într-un depozit de module software în vederea unei eventuale viitoare reutilizări, căutarea în baza de componente, modificarea în vederea adaptării la cerințele programului care se proiectează, respectiv reutilizarea propriu-zisă, prin înglobarea în programul în lucru.

În figură se remarcă existența unui *depozit* de componente reutilizabile (module de program) care se îmbogățește cu fiecare nou proiect. În legătură cu acest *depozit*, se cuvine făcută observația că modificările componentelor extrase din *depozit* (dacă sunt necesare) nu trebuie să afecteze componentele deja existente în depozit, fiind de preferat ca reprezentarea să producă o nouă componentă, care se va adăuga *depozitului*. Aceasta înseamnă că de fapt *depozitul* se actualizează în permanență, deci procesul ciclic de construire a produselor software duce la creșterea dimensiunii *depozitului* cu fiecare realizare a unui produs nou. Cu cât numărul de produse-program realizate este mai mare, cu atât dimensiunea *depozitului* de componente reutilizabile crește și prin urmare efortul de realizare a componentelor noi sau cel de reproiectare a celor existente scade. Explicația este la îndemână: cu cât *depozitul* este mai mare, cu atât mai probabil se găsește ceva "potrivit" pentru o situație dată, sau măcar ceva foarte "asemănător".

Nu este mai puțin adevărat, și aceasta reprezintă o reală dificultate, că este greu să se caute ceva potrivit într-un *depozit* cu multe componente. Apar dificultăți în definirea criteriilor

de căutare, în căutarea propriu-zisă şi mai ales la înţelegerea structurii interne şi a funcţiilor diverselor module-program din *depozit*. Unele din aceste dificultăţi sunt legate de reprezentările şi convenţiile folosite.

1.4. Clase de aplicaţii

Clasa ca şi noţiune primară, este folosită în lucrarea de faţă în accepţiunea "naturală" de colecţie de obiecte având un număr de atribute comune bine precizate. Noţiunea de **clasă de programe** este de fapt vag definită. Descrierea ("definirea") din [Mit87], amintită la începutul capitolului , conţine formulări imprecise (spre exemplu "*variantă specializată*"). Nu am întâlnit în literatură o formalizare completă (în sensul unei definiri riguroase, neunivoce şi compacte, chiar şi în sensul lingvistic) a conceptului (aici însă nu ne referim la definiţia folosită în **POO**).

Exemple de clase de programe sunt aplicaţiile de la automatele de eliberat bani, aplicaţiile contabile, aplicaţiile de gestiune, aplicaţiile de conducere de proces enumerate la § 1.1, etc. Se observă că în aceste exemple nu sunt prea uşor de definit graniţele aplicaţiilor, respectiv apare specificul de "vag".

Obs.:

*Această specificitate de formulare vagă deschide posibilitatea unei abordări **fuzzy** (dar nu a definiţiei!) pentru noţiunea de clasă, adică stabilirea programelor care pot să aparţină unei clase. În această abordare, accepţiunea de clasă poate fi adaptată la percepţia subiectivă, prin definirea unor funcţii de apartenenţă, reguli de inferenţă şi de defuzzificare după optica specialistului. Caracterul subiectiv poate fi atenuat prin stabilirea regulilor specifice logicii fuzzy, prin medierea părerilor mai multor specialişti.*

O încercare de definire trebuie să ţină cont de structura unui program, din care, prin modificări efectuate până la "un nivel admisibil", impus arbitrar (relativ la care se raportează trăsăturile clasei respective), să se poată obţine *versiuni* ale acelui program, numite **variante**. Programul din formularea de mai sus este numit **program generic** (**PG**). Deci clasa este compusă din programul generic şi din programe membre - variante [1]. Modificările care se efectuează asupra programului generic nu trebuie să afecteze un set de funcţii ale sistemului care înglobează un program al clasei, pe care le numim **funcţii de bază ale clasei**. De aceea, structura programului se va "conserva" suficient de mult în variantele sale, pentru a putea să le încadrăm în aceeaşi clasă de programe. Toate programele membre ale clasei vor avea deci aceleaşi funcţii de bază.

[1] Ne referim, în cele ce urmează, la programe sub forma **codului-sursă.**

Evident, păstrarea aceloraşi funcţii de bază atrage după sine păstrarea, cu anumite modificări, a codului pentru prelucrările secvenţiale (blocurile de prelucrări secvenţiale - din schemele logice). Funcţiile de bază sunt percepute aici ca fiind un set de prelucrări, privite independent de forma de manifestare. Modificările de care e vorba mai sus presupun definirea unui nivel admisibil, stabilit arbitrar (de fapt, subiectiv), dincolo de care nu se mai poate vorbi de apartenenţa la aceeaşi clasă. Deoarece oricare variantă din cadrul clasei se poate obţine prin modificări (în limite rezonabile) ale programului generic, rezultă că acesta din urmă are, cu probabilitate foarte mare, structura cea mai complexă din cadrul clasei.

Scopul obţinerii unei variante îl constituie adaptarea unui program ("generic") la nişte situaţii concrete de funcţionare. Acest scop poate fi atins prin conceperea unor tehnici de proiectare care să permită dezvoltarea rapidă a variantelor, pentru situaţii concrete de funcţionare. Aceste tehnici sunt avantajoase în cazul cererii mari de aplicaţii asemănătoare (care aparţin aceleiaşi clase). Unul din obiectivele acestei lucrări îl reprezintă dezvoltarea unora dintre aceste tehnici, pentru anumite categorii de aplicaţii de conducere sau instrumentaţie de proces.

1.5. Sisteme de gestiune a bazelor software

Încă din 1987 Mittermeir avansează ideea construirii variantelor pe un "schelet" comun pentru toate exemplarele specifice (variante) [Mit87]. Acestea se construiesc specificând părţile care rămân din programul generic, atât ca şi structuri de control (care determină structura propriu-zisă a programului) cât şi ca şi cod (adică partea de procesare secvenţială - spre exemplu taskurile). Modul de construire a variantelor, precum şi mecanismele implicate, se numesc, împreună, **sistem de gestiune a bazei software (SGBS)**. Astfel, efortul de proiectare este împins spre **utilizatorul final (UF - sau "end - user")**, ceea ce însă necesită un suport software potrivit, suficient de sofisticat, pentru SGBS.

Cheia abordării proiectării pentru reutilizare o constituie găsirea unor criterii potrivite de clasificare a programelor. O accepţiune naturală pentru aceste criterii de clasificare duce la împărţirea programelor după două astfel de criterii , după aplicaţie (s.ex. programe de telemecanică, robotică, CNC, etc.) şi după atribuţii (s.ex. interfaţă cu utilizatorul, generare de rapoarte, funcţii de timp real, etc.). Programele asemănătoare, din punctul de vedere al unuia din criteriile de mai sus, pot fi încadrate la **categoria aplicaţie**, respectiv **categoria task**. Astfel, programele, mai exact, modulele de program, notate mai jos cu P_{ij}, pot fi clasificate după apartenenţa la cele două categorii, după următoarea schemă matricială (Tabelul 1.1, în care indicele *i* desemnează apartenenţa la categoria aplicaţie, iar indicele *j* - apartenenţa la categoria task) :

Tabelul 1.1. Clasificarea modulelor de program

Taskuri -> / Aplicații ↓	T_1	T_2	...	T_m
A_1	$P_{1,1}$	$P_{1,2}$	...	$P_{1,m}$
A_2	$P_{2,1}$	$P_{2,2}$	...	$P_{2,m}$
...	...	...	...	...
A_n	$P_{n,1}$	$P_{n,2}$	...	$P_{n,m}$

(T - task, A - aplicație, P - program)

Clasificarea aceasta este utilă din punctul de vedere analizat dacă elementele clasificării respectă una din următoarele condiții :

$\exists\, s,t$ a.î. $P_{s,1} = P_{t,1}$ (*egalitate*) (1.1,a)

$\exists\, f_j,\ \exists\, s,t$ a.î. $f_j(P_{s,1}) = f_j(P_{t,1})$ (*substituibilitate*) (1.1,b)

$\exists\, PG_j : (\forall)\, s = 1,n$ a.î. $P_{s,j}$ **ISA** PG_j (*generalizabilitate*) (1.1,c)

(cu **ISA** - operator de **generalizare**)

Prima situație (rel. 1.1,a) practic se întâlnește destul de rar : în această accepțiune noua variantă ar fi doar o succesiune de **apeluri de proceduri** dintr-o bibliotecă existentă. A doua situație (rel. 1.1,b) reflectă situația existenței unor taskuri cu adaptări pentru diverse situații, dar care să aibă aceeași **funcție** (deci care sunt dispuse pe o coloană, în Tabelul 1.1). Transformarea de la un task la altul este realizată de *omomorfismul* f_j (este omomorfism întrucât mulțimea taskurilor este bine definită și rezultatul transformării trebuie să aparțină aceleiași mulțimi, iar echivalența dintre taskuri, din punctul de vedere al funcțiilor, duce la concluzia că mulțimea valorilor transformărilor conduce la mulțimea taskurilor ; transformarea propriu-zisă, adică operația internă care constituie omomorfismul, este

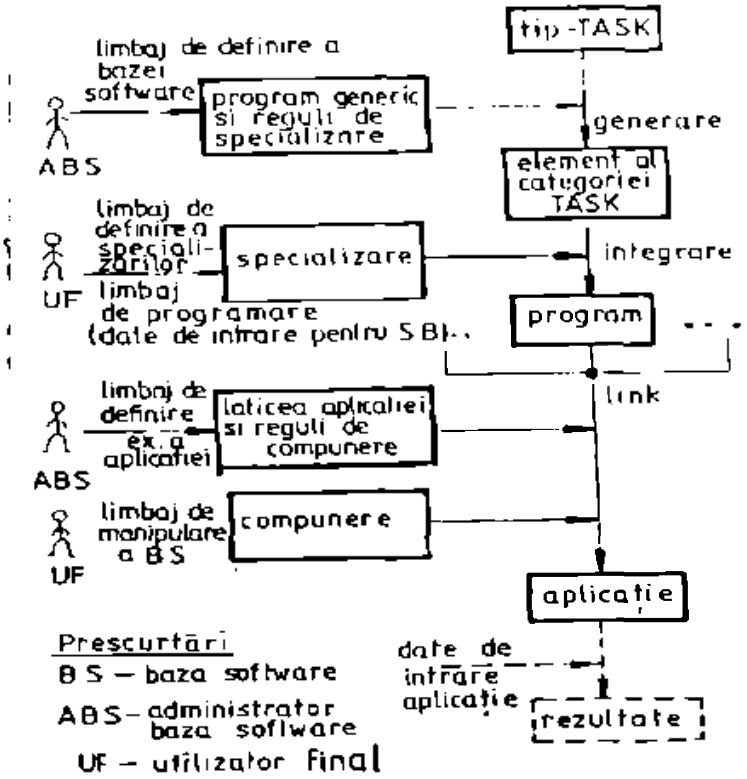


Fig.1.3. Activitățile de proiectare software folosind SGBS

operația de adaptare prin reproiectare pentru diversele situații specifice în care trebuie să funcționeze taskurile). A treia relație (1.1.c) se referă la posibilitatea existenței unui modul de program generic PG_j care să fie **generalizarea** oricărui task din coloana j, respectivă. Astfel, un program poate fi construit plecând de la programul generic PG_j, integrându-l (reunindu-l) cu o parte specifică aplicației S_{ij}. Proiectantul de nivel înalt, care este diferit de **UF** și este denumit în cele ce urmează, **administrator al bazei software (ABS)**, va defini un set de **reguli de specializare** RS_j pentru fiecare **categorie task**. Astfel, descrierea fiecărui task trebuie să urmeze un prototip general, numit **tip - task**. Fiecare obiect al **tip-task** constă într-un atribut pentru **PG** și altul pentru regulile asociate de specializare, **RS**.

Schema generală de utilizare a unui **SGBS** pentru obținerea unei aplicații este prezentată în fig. 1.3. Se observă că **ABS** aplicației la condițiile concrete de funcționare. Se poate remarca intervenția, în mai multe rânduri operează la nivelul definirii regulilor, pe când **UF** este cel care realizează specializarea, adică adaptarea, a proiectantului software de nivel (foarte) înalt, respectiv a **ABS**, iar **UF** nu face decât să utilizeze regulile pe care le-a formulat **ABS**, în etapa anterioară fiecărei intervenții a **UF**. Blocurile de pe fluxurile orizontale reflectă acțiuni ale celor care manipulează **SGBS** (adică ale **ABS** și **UF**) iar cele de pe fluxul vertical reprezintă rezultatele diverselor prelucrări specificate alături de acest flux.

Fluxul prelucrărilor este următorul :

- - plecând de la prototipul furnizat de **tip-task**, **ABS** definește regulile de specializare pentru fiecare categorie-task în parte ;
- -utilizatorul final realizează specializarea, conform cerințelor aplicației proprii, pentru fiecare program al aplicației, plecând de la categoriile-task respective și de la regulile de specializare definite;
- - **ABS** stabilește regulile după care modulele de program obținute pot fi integrate (înglobate) în sistem (în acest caz, este vorba nu atât de reguli cât de o structură generală în care modulele pot fi incluse) ;
- - **UF** realizează asamblarea sistemului final (adică aplicația dorită) din modulele de program, ținând cont de structura anterior definită.

Prezentăm mai jos, unele aspecte legate de fiecare etapă a parcurgerii unui **SGBS**.

În etapa de definire a regulilor de specializare, trebuie ținut cont de modalitatea de obținere a unui program, descrisă anterior. Ținând cont de aceasta, Tabelul 1.1 poate fi refăcut, obținându-se o formă mai utilă din punctul de vedere analizat. În acest nou tabel (Tabelul 1.2), CT_i reprezintă categoria-task i, IT_i semnifică interfața taskului i, RS_i reprezintă regulile de specializare pentru categoria task i, S_{ij} sunt specializări ale categoriei-task i, iar PG_i sunt programe generice.

Tabelul 1.2. Clasificarea modulelor de program ţinând cont de descompunerea **PG-RS-S**

		Categorii task									
		CT_1			CT_2			...	CT_n		
		PG_1	RS_1	$\{IT_1\}$	PG_2	RS_2	$\{IT_2\}$	... ...	PG_n	RS_n	$\{IT_m\}$
Aplicaţii	A_1		$S_{1,1}$			$S_{1,2}$		...		$S_{1,m}$	
	A_2		$S_{2,1}$			$S_{2,2}$		...		$S_{2,m}$	
	...		...			...		$S_{i,j}$		...	
	A_n		$S_{n,1}$			$S_{n,2}$		...		$S_{n,m}$	

Interacţiunile dintre taskuri pot fi de asemenea prezentate tabelar, spre exemplu ca în Tabelul 1.3. Prin acest din urmă tabel (numit matrice de interfaţă task) se definesc unele restricţii de acces. În matrice, există legături permise (adică dacă între două taskuri există legături definite şi admisibile, la intersecţia liniei şi coloanei corespunzătoare celor două taskuri există un *1*) legături interzise (cu *0* la intersecţia respectivă) şi legături nedefinite (marcate prin punct).

Tabelul 1.3. Matricea de interfaţă task

	T_1	T_2	T_3	T_4
T_1	*0*	.	*1*	*1*
T_2	*1*	*0*	*0*	*1*
T_3	*1*	*0*	*0*	*1*
T_4	.	.	*0*	*1*

unde:
. - legătură nedefinită
1 - există interfaţă
0 - legătură interzisă

Foarte importantă pentru corecta definire a elementelor unui **SGBS** este definirea unor restricţii de integritate. Câteva exemple de astfel de restricţii sunt :

- la nivelul task:
  - interfeţele taskurilor trebuie să fie compatibile cu ale taskurilor deja existente;
  - la fiecare regulă de specializare trebuie definită o alternativă în lipsă (*default*);
  - doar specializările de interfeţe compatibile cu matricea de interfaţă sunt acceptate (altfel, trebuie modificată matricea);
- la nivelul - program:
  - fiecare program trebuie să fie specializarea unui program generic existent;
  - interfeţele de program trebuie să fie compatibile cu interfeţele de task;

- nu trebuie să existe porțiuni de program care nu au fost explorate (parcurse), în procesul de proiectare, în vederea efectuării specializării pentru versiunea curentă, dacă e cazul ;

- la nivelul aplicației :
  - pentru toate datele trebuie să existe câte un producător (generator);
  - compunerea aplicației trebuie să se realizeze conform laticei aplicației;
  - nu trebuie să existe specializări în aplicație, incompatibile cu interfețele.

Specializarea propriu-zisă are loc prin acțiunea **UF**. Acesta efectuează selectarea specializărilor dorite, pe baza cerințelor specifice ale aplicației sale. Această activitate poate fi reprezentată prin schema din fig. 1.4. Aici, termenul de "integrare" este folosit în sensul de "reunire" (sau **compunere**) a componentelor, adică a părții generice și a părților de program specializate pe baza regulilor definite de **ABS** și selectate de **UF**. Evident, dacă se folosesc alte reguli de specializare (setul k din figură), se va obține un program specializat pentru o altă aplicație (programul i,k, obținut pe calea punctată). Regulile de specializare sunt definite pentru aspecte particulare ale programului (spre exemplu, cele din figură).

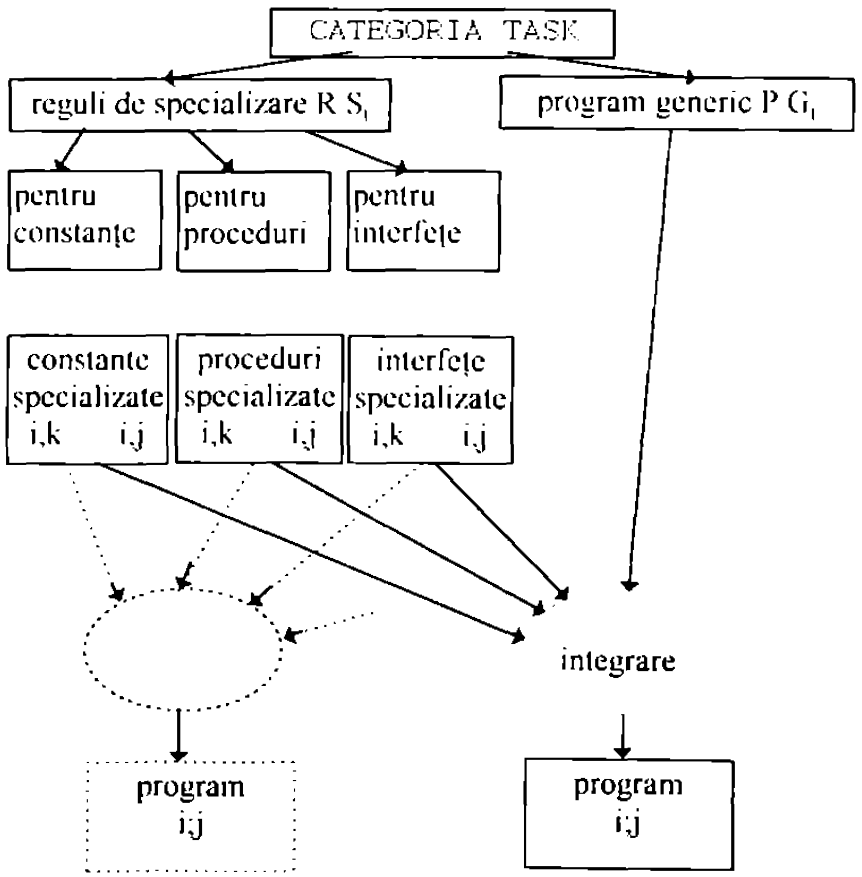


Fig. 1.4. Obținerea unui program prin utilizarea unui **SGBS**

Programele obținute trebuie integrate în aplicația finală, cea dorită de **UF**. Integrarea acestor programe (în sensul uzual al cuvântului) se realizează urmând o schemă de încadrare, practic o structură generală, o schemă de apartenență, numită **latticea aplicației**.

Această ultimă etapă de compunere se desfășoară urmând regulile induse de această latice. Aceasta este o structură generică pentru toate aplicațiile unei clase date. Ea va fi

completată de către **UF** cu toate taskurile obținute prin specializarea potrivită. Astfel, laticea aplicației asigură cadrul în care aceste taskuri pot să-și desfășoare activitatea. Laticea constă, concret, din acele fragmente mici de cod care asigură legătura între modulele de program. Laticea este definită de **ABS**, furnizând un cadru pentru obținerea oricăror aplicații din clasa pentru care este definită baza software. Un exemplu de astfel de *latice* este prezentat în Fig.1.5.

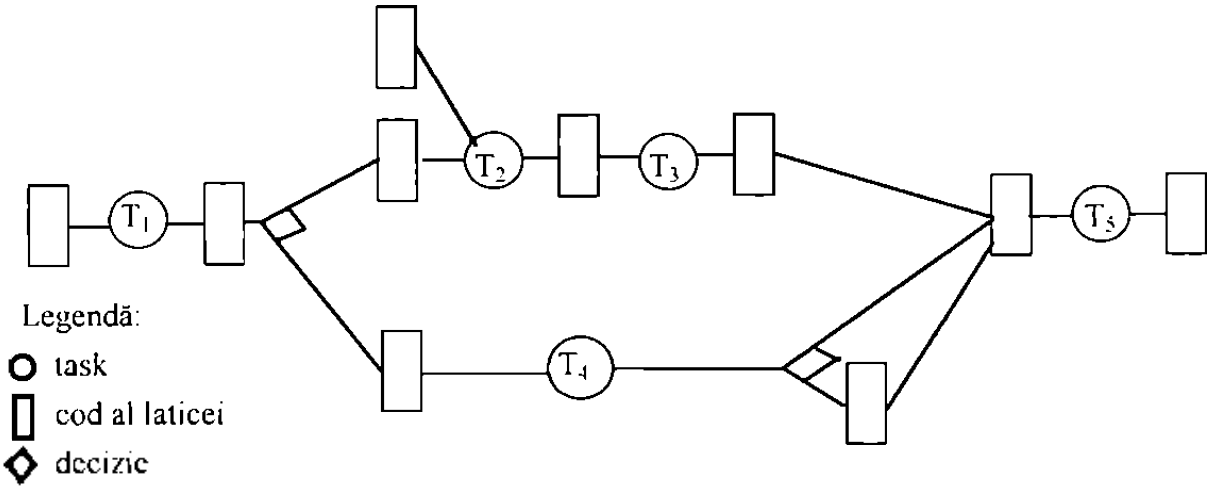


Fig. 1.5. Exemplu de *latice* de aplicație.

Se pot eventual specifica, în procesul integrării taskurilor în latice, părți ale structurilor de control lăsate la latitudinea **UF** de către **ABS** (în locurile marcate pe schemă prin blocuri de decizie).

Un exemplu concret al aplicării acestei abordări a proiectării pentru reutilizare este dat în [Mit87].

Sintetic, **elementele de interes** din acest paragraf sunt deci :

- *Folosirea conceptului de clasă, respectiv variantă, s-a generalizat pentru situațiile de proiectare a unor aplicații cu trăsături și funcții "asemănătoare".*
- *Există abordări din perspectiva ingineriei programării, care sunt menite să contribuie la creșterea productivității proiectării, prin utilizarea unor **SGBS**. Aceasta înseamnă că există variante generice, atât la nivelul structurii cât și la nivelul taskurilor, care urmează să fie adaptate la specificul aplicației de către **UF**.*
- *Pentru această adaptare, sunt necesare următoarele :*
  - *definirea, pentru fiecare task (sau modul de program), a părții generice și a regulilor de specializare (pentru constante, proceduri și interfețe) ;*
  - *definirea unor restricții de acces și integritate ;*
  - *definirea structurii generice a aplicației și a regulilor de compunere.*

Aplicarea acestui algoritm de proiectare, fie că are ca suport un mediu adecvat, fie că este realizată direct de proiectanți prin metode clasice, doar urmând disciplina de proiectare impusă, este o metodă de foarte mare productivitate. În prima versiune de mai sus, mediul se numește **generator de aplicații (GA)**, iar în al doilea caz, numim ansamblul de tehnici și algoritmi de proiectare, **tehnici și structuri de program pentru creșterea gradului de reutilizare** sau, pe scurt, *proiectare pentru reutilizare*. Cei mai avantajați utilizatori ai acestor abordări sunt acei producători de produse software care produc variante în mare număr, pentru mulți clienți și care variante conțin taskuri de bază care se prezintă în multe versiuni puțin diferite între ele. Această metodă poate fi folosită și în etapa de întreținere din ciclul de viață al programelor, atunci când, în exploatarea sistemului apar noi cerințe care atrag după sine necesitatea proiectării unei noi variante, care refolosește cea mai mare parte a proiectului și codului existent.

1.6. Puncte de vedere referitoare la reutilizare

Proiectanții software de la diverse firme producătoare de software sau din mediile academice, conform experiențelor și propriilor dezvoltări personale, au creat diverse tehnici și contexte de utilizare, care dau imaginea unei mari varietăți, a unui veritabil mozaic de situații de reutilizare în proiectare. În Anexa 1.1 prezentăm câteva puncte de vedere referitoare la uzul tehnicilor de reutilizare software, așa cum sunt acestea înțelese de diferiți autori, și folosite ca atare. Articolele analizate prezintă astfel de tehnici și contexte de utilizare, de asemenea clasificări și viziuni proprii autorilor, referitoare la domeniul de interes al lucrării de față. Un asemenea demers este justificat în vederea încadrării corecte a contribuțiilor originale ale tezei.

Deși punctele de vedere analizate prezintă un caracter istoric, sunt interesante pentru a ne face o imagine globală completă asupra domeniului (v. Anexa 1.1).

1.7. Generatoare de aplicații

Un **generator de aplicații** este un program conceput pentru a "*traduce*" direct specificațiile aplicației fie într-o formă executabilă imediat fie în cod-sursă (într-un limbaj de programare uzual). Specificațiile pot fi introduse de către utilizator (**UF** din § 1.5) în programul generator fie prin intermediul unei interfețe adecvate (de tip *spread-sheet* sau *interfață grafică*) fie prin folosirea unui (meta)*limbaj de specificații*. Utilizarea unui generator de aplicații reduce drastic timpul de proiectare, însă de obicei aria de competență a unui asemenea program este

mai restrânsă, întrucât apar dificultăţi importante fie la definirea unui limbaj de specificaţii pentru o clasă cât mai largă de sisteme, fie la descrierea unei interfeţe suficient de complexe pentru a permite definirea aplicaţiei aparţinând unei astfel de clase.

În paragrafele următoare prezentăm câteva exemple de generatoare de aplicaţii realizate în cadru academic, generatoare comerciale şi câteva exemple din domeniul instrumentaţiei.

1.7.1. Generatoare de aplicaţii dezvoltate în cadru academic

În mediile univesitare au fost dezvoltate diverse generatoare de aplicaţii care au meritul de a fi clarificat o serie de aspecte teoretice şi practice ale dezvoltării unor astfel de instrumente software. Din acest motiv, ele constituie modele pentru aplicaţiile comerciale.

a) În [Nei84] şi [Fre87] este prezentată abordarea **DRACO** realizată la Universitatea Irvine din California. Paradigma aceasta se referă la reutilizarea analizei (cuprinzând aici şi referinţe la particularităţi ale limbajelor specializate) şi a proiectării (deci aspecte ale implementării).

Paradigma **DRACO** utilizează:

- tehnici de meta-compilare;
- limbaje de nivel înalt;
- aspecte "*delicate*" tratate în limbaje de nivel scăzut;
- modelarea domeniului aplicaţiei;
- transformări de program-sursă-în-sursă;
- componente software reutilizabile.

DRACO este un **GA** extensibil, în sensul că acest instrument software permite, având asociată o mulţime accesibilă de domenii, configurarea unor aplicaţii din aceeaşi clasă. Aceasta poate însă să suporte modificări (nu foarte radicale). Cu alte cuvinte, instrumentul este un generator de generatoare de aplicaţii. Un domeniu este aici o porţiune bine delimitată a realităţii, asupra căruia trebuie efectuate intervenţii reflectate prin anumite funcţii ale unei aplicaţii software dintr-o clasă (în sensul din [Mit87]) precizată.

În [Fre87] se face diferenţierea între paradigma **DRACO** şi instrumentul software asociat : prima cuprinde şi aspecte care nu pot fi incluse în instrument. Astfel, nu se pot include unele aspecte ale gândirii care defineşte proiectul, precum este modul de obţinere a informaţiei necesare definirii limbajului de specificaţii.

DRACO asigură în principal următoarele două funcţii :

1) definirea şi implementarea limbajelor de specificaţii ;

Fig. 1.6. Structura internă a specificaţiilor

2) asistarea şi automatizarea parţială a sarcinii traducerii cerinţelor din limbajele de specificaţii într-un limbaj concret (din care să se poată obţine codul executabil).

Corespunzător acestor funcţii, **DRACO** lucrează în două faze distincte :

- **definirea domeniului** : aceasta presupune folosirea unor meta-compilatoare pentru limbaje specifice de specificaţii. Proiectarea limbajului domeniului şi deci determinarea semanticii sale sunt dificile
- **rafinarea specificaţiilor** ; aici, esenţial este să delimităm o structură logică internă a acestei etape, ca în Fig. 1.6. Cele 4 elemente din figură concură la traducerea descrierii domeniului în cod executabil, descrierea în limbajul domeniului parcurgând *transformări* succesive dinspre exterior spre interior, spre componentele care asigură corespondentele semantice ale fiecărei operaţii şi fiecărui obiect din descrierea domeniului, care fixează comportarea, de fapt, a rezultatului prelucrării. O definire *analitică* descrie modul cum **DRACO** recunoaşte şi traduce propoziţii din limbajul domeniului în limbajul intern cu care lucrează iar definirea *bun de tipar* descrie transformarea inversă, într-un limbaj specific. Pentru fiecare operaţie este definită o componentă care poate să aibă mai multe *instanţieri*, corespunzând unor cerinţe de performanţă (spre ex. "rapid" versus "încet dar mai precis"). Fiecare componentă (cu elementul său de limbaj asociat) este de fapt un tip abstract de date (posibil cu mai multe variante de performanţe diferite - acele instanţieri amintite mai sus).

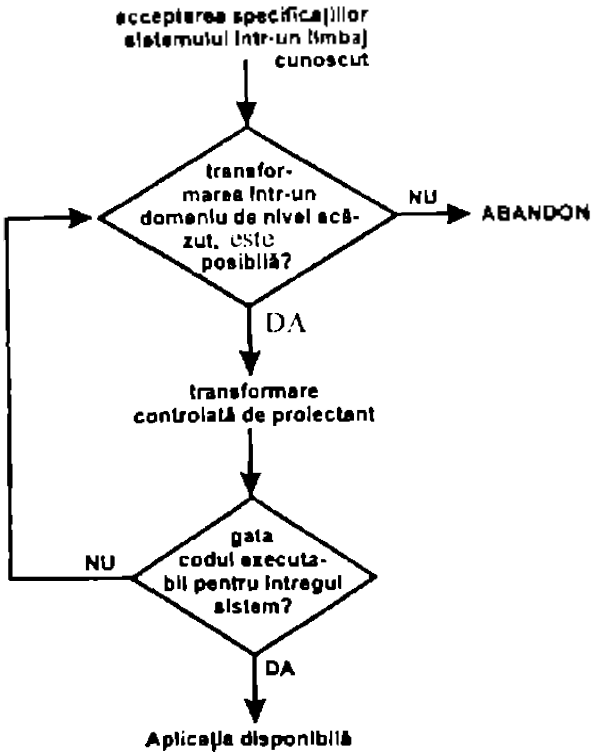


Fig. 1.7. Logica operării **DRACO**

DRACO este un generator de aplicații în care baza de transformare este modularizată după criterii specifice domeniului. Logica de derulare a proiectării **DRACO** este prezentată în Fig.1.7.

Pentru analiza **DRACO**, se folosește un cadru conceptual constituit din șase elemente (concepte) diferite:

1) *Nivelele de abstractizare* - care presupun nu numai înlocuirea fragmentelor particulare prin nume (o trăsătură abstractă) ci și înglobarea abstractizărilor într-un sistem ierarhic.

2) *Structura produselor* - prin utilizarea **DRACO**, ceea ce interesează este structura specificațiilor, nu a codului produs. Deoarece **DRACO** folosește limbaje adaptate domeniului, se poate obține o bună structurare a programului în domeniu.

3) *Prelucrarea sistematică* - este un principiu care vizează buna ordonare a activităților, într-un mod sistematic.

4) *Modelarea* - neavând acces la software-ul propriu-zis, prin utilizarea generatorului de aplicații, trebuie să efectuăm activități de ingineria programării prin descrierea și manipularea (prelucrarea) modelelor aplicației care este în curs de proiectare, în vederea obținerii modelului corespunzător.

5) *Compartimentarea cunoștințelor* - fiecare punct al dezvoltării este purtătorul unui fragment de cunoștințe (informații) care pot să cuprindă potențial decizia care trebuie luată.

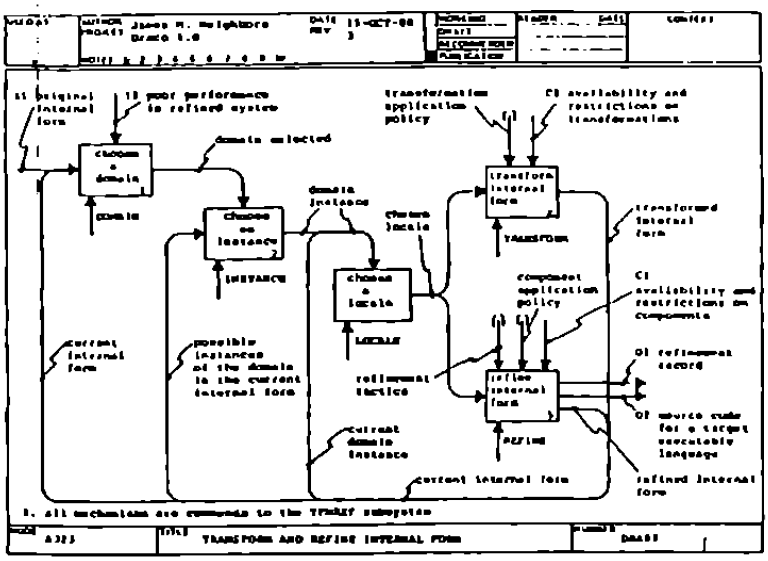


Fig. 1.8. Exemplu de lucru cu **DRACO** (aspectul consolei)

6) **Reutilizare** - toată paradigma **DRACO** este de fapt orientată spre reutilizarea de fragmente de program, teste, tehnici, ori de câte ori este posibil.

În [Frc87] sunt date câteva exemple de folosire a mediului de proiectare **DRACO**. În Fig. 1.8 se prezintă aspectul consolei pentru un caz concret de utilizare.

b) La Universitatea din Austin, Texas, a fost dezvoltat produsul **CODE/ROPE** [Bro90]. Acest produs este un mediu de dezvoltare pentru programare concurentă, care utilizează o interfață grafică *declarativă ierarhică*. Această interfață asigură folosirea unor componente cu grad diferit de abstractizare, permițând refolosirea pentru noi aplicații a tuturor tipurilor de componente oferite de mediul de dezvoltare. Astfel, utilizând **ROPE**, s-au obținut noi aplicații care cuprindeau module refolosite din biblioteca **ROPE** în proporție de 80%. **ROPE** (*Reusability-Oriented Parallel-programming Environment*) este parte a mediului **CODE** (*Computation-Oriented Display Environment*) al cărui scop este construcția programelor concurente utilizând un graf ierarhic ca și model al procesării. **CODE** oferă o platformă potrivită pentru a construi un sistem reutilizabil, punând la dispoziția utilizatorului facilități de clasificare, găsire, înțelegere, modificare, combinare a componentelor. **ROPE** răspunde necesităților de a manevra astfel componente de diverse grade de abstractizare și de a combina chiar și componente care sunt subgrafuri arbitrare, realizând și gestiunea modulelor de cod în mai multe limbaje de nivel superior (**ADA, FORTRAN, C, PASCAL**). În concluzie, spre deosebire de **GA** prezentate anterior, produsul **CODE/ROPE** are mai mult rolul de a gestiona și combina module existente, rolul de proiectant al structurii revenind în bună măsură proiectantului; deci, nu are ca intrare o listă de specificații.

Sunt trei aspecte caracteristice ale produsului, care merită subliniate:

1) Clasificarea și regăsirea componentelor - Schemele tradiționale de regăsire sunt *ierarhice* sau bazate pe *cuvinte-cheie*. **ROPE** folosește o combinație a acestor scheme de regăsire, combinație numită **clasificare structurată relațională**. Această metodă este prezentată în Fig. 1.9. Schema presupune existența unui tabel de componente, cu atributele asociate, iar domeniului de valori al fiecărui atribut îi sunt asociate structuri care rezultă din semantica domeniului. Astfel, **regăsirea** unei componente prin căutare presupune selectarea valorilor structurate ale unui atribut sau a tuturor componentelor care au valorile atributelor corespunzătoare unui set dat.

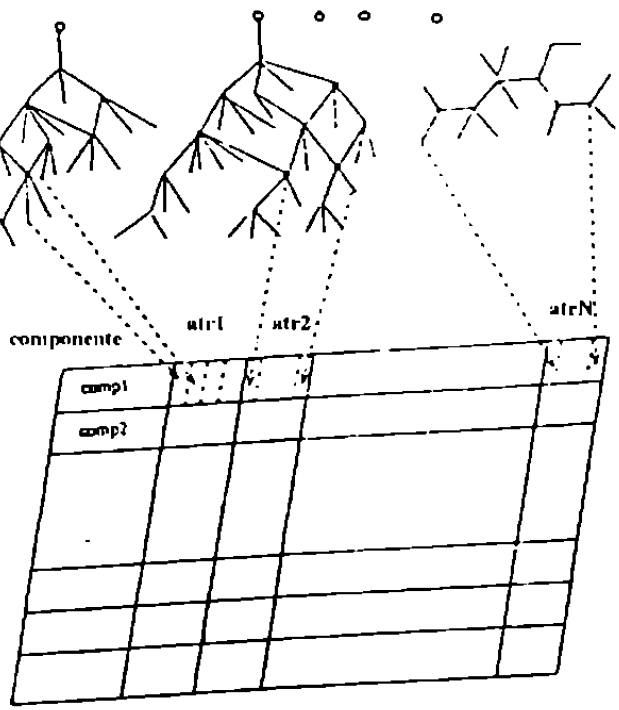


Fig. 1.9. Schemă de clasificare structurată relațională

2) Înțelegerea componentelor este

realizată prin accesul la o bază de date de documentare a fiecărei componente.

Interfatarea componentelor este similară cu cea a pinilor unui circuit integrat ; fiecare componentă are asociate informații asociate schemei de **I/O**, care pot fi accesate la cerere, în formă grafică și tabelară. Astfel, proiectantul poate identifica ușor dependențele dintre componente și poate să le interconecteze cu cele din jur. În figura 1.10 este prezentat un exemplu de utilizare a sistemului **CODE/ROPE.**

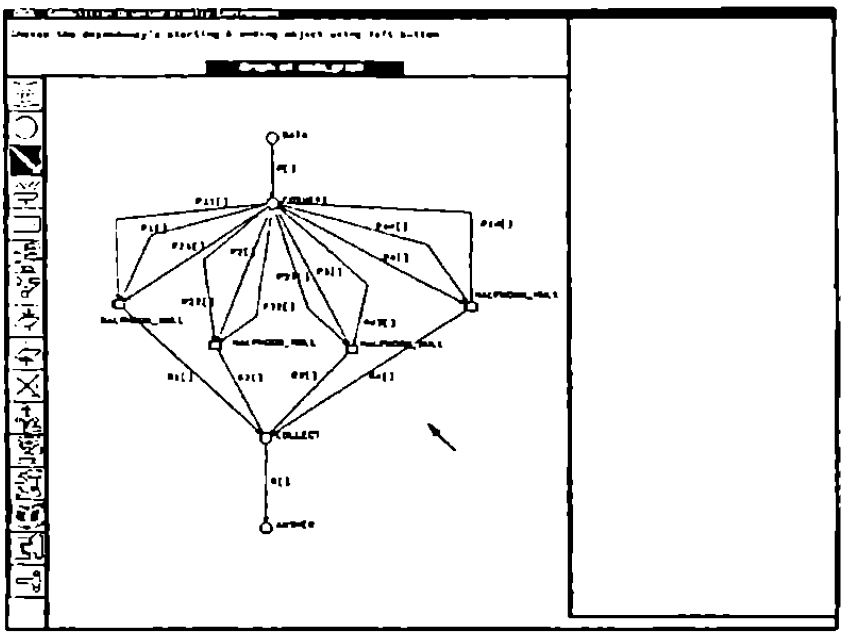


Fig. 1.10. Exemplu de utilizare **CODE/ROPE** (aspectul consolei)

c) GA descris în [Man88], are la bază niște idei din [Tiv87-B] și o colaborare pe această temă. Acest generator acoperă domeniul clasei de aplicații de telemecanică, prezentate în [Tiv87-N]. Aplicațiile din această familie constau în stații de telemecanică legate la un nivel ierarhic superior, stații destinate interfațării bidirecționale cu procesul. Funcțiile programului din stația dintr-o aplicație tipică de telemecanică din această familie, sunt :

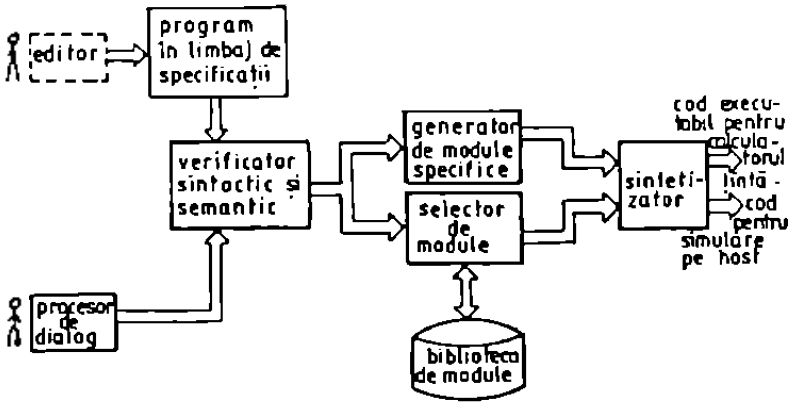


Fig. 1.11. Generatorul de aplicații pentru stații de telemecanică

- interfațarea cu procesul (în ambele sensuri);
- comunicarea cu nivelul ierarhic superior;
- urmărirea în etape a execuției telecomenzilor;

- autotestarea.

Utilizând o structură de program şi de date proiectată conform cerinţe-lor prezentate în Cap. 6, § 6.3, rolul **GA** din [Man88] este de a asigura completarea zonelor de date alocate descrierii volumului de infonnaţii din proces, precum şi a tabelelor de adrese corespunzătoare listelor cu lungimea dependentă de proces, cu valorile rezultate prin dialog cu proiectantul-utilizator. Acesta furnizează sistemului informaţii actuale despre proces, care privesc adaptarea aplicaţiei la condiţiile concrete. Informaţiile acestea sunt în primul rând referitoare la volumul de variabile de proces din categoriile admise (variabile de proces numerice pe 2 biţi, variabile pe 1 bit, intrări analogice, ieşiri de telecomandă).

Generatorul construieşte, pe baza acestor informaţii, partea variabilă, dependentă de structura procesului, din Fig. 6.5 şi o adaugă la partea de cod fixă, direct în structură de cod executabil Astfel, dintr-o singură trecere prin generatorul descris, aplicaţia nouă este disponibilă. Schema-bloc a **GA** este prezentată în Fig. 1.11.

d) În lucrarea [Che84], autorul prezintă un mediu de dezvoltare numit **PDS** (*"Program Development System"*) care permite aplicarea conceptului de prototipizare rapidă. Prin posibilităţile oferite, care de fapt combină facilităţile de **SGBS** cu elemente de generare de aplicaţii, putem să înglobăm acest program la graniţa labilă dintre **GA** şi instrumentele **CASE** (§ 1.8).

Mediul are trei părţi mari : **o bază de module software**, o interfaţă utilizator şi un set de unelte (*"scule"*) care pot fi apelate prin intermediul interfeţei, pentru a manipula astfel modulele din baza de module, în vederea prototipizării rapide, respectiv a obţinerii rapide a unei prime versiuni a aplicaţiei (de unde rezultă caracterul de **GA** al programului). Fiecare modul memorat are asociat un nume, un număr de *"derivare"* (adică un număr de ordine al adaptării din programul *"părinte"*- sau de bază), o *"istorie"* a derivărilor şi un număr de versiune. "Uneltele" includ un editor şi câteva operaţii, după cum urmează :

- un **editor** de modul, care permite studiul entităţilor şi atributelor şi apelul editoarelor specializate (de text sau de structură) pentru a crea sau edita atribute particulare;
- **Transformare** - duce la obţinerea unui modul prin selectarea unor atribute şi transformarea acestora;
- **Reunire** - determină obţinerea unui modul prin reunirea unor părţi selectate din mai multe module;
- **Analiză** - declanşează operaţia de obţinere a unui modul folosind rezultatele analizei mai multor module;
- **Împachetare** - determină obţinerea unui modul în care atributele sunt ordonate astfel încât un item să fie definit înainte de utilizarea sa (ca pregătire pentru compilare);

- **Compilare** - duce la obținerea unui modul "*derivat*", din unul sau mai multe module-subiect;
- **Fixare** - permite editarea și recompilarea pentru modificări asupra unui număr limitat de entități;
- **Agregare** - permite asocierea unor module, construind configurații de module.

Mediul de dezvoltare mai are și alte funcții, implicite, spre exemplu :

- controlul versiunii - asigură controlul corectitudinii versiunii fiecărui atribut, în sensul asigurării că acesta reflectă corect schimbările de la derivarea precedentă;
- controlul accesului - fiecare "unealtă" are acces la baza de module astfel încât să se mențină corect istoria derivărilor;
- reprezentări comune - există un set de reprezentări pentru module și atributele componentelor și o mulțime de utilități care lucrează cu acestea; introducerea unor noi "unelte" necesită eventual introducerea unor noi reprezentări pentru ca aceste noi "unelte" să poată manipula modulele, dar ceea ce contează este principiul general al utilizării de către "unelte" a reprezentărilor.

1.7.2. Diverse generatoare comerciale de aplicații

În presa de informare tehnică apar frecvent referiri la programe și medii care pot fi încadrate în categoria **GA**. Acesta este un sindrom al presiunii pieței în domeniul analizat aici, cel al **GA**, și deci o confirmare a eforturilor noastre în acest sens.

În Anexa 1.2 prezentăm foarte succint câteva din **GA** existente pe piața produselor software. Din aceste scurte prezentări ale principalelor caracteristici ale unor astfel de programe comerciale care pot fi asimilate categoriei **GA**, putem formula câteva concluzii :

- toate (sau aproape toate) programele au multe alte funcții, generarea de aplicații fiind doar una dintre acestea ; de multe ori, facilitățile de **GA** sunt înglobate ca și funcții în cadrul unui program sau chiar pot să fie programe distincte dintr-un pachet ;
- acțiunile de generare de aplicații sunt de obicei limitate la un subset al funcțiilor unui sistem (de cele mai multe ori, interfața face obiectul generării) ;
- majoritatea covârșitoare a **GA** folosesc principiile **POO**, respectiv clase și ierarhii de clase ; cele mai multe pot fi asimilate unor așa-zise **frameworks** (sau **cadre**, v. § 1.11.3) ;
- puține **GA** își propun să acopere domeniul conducerii de proces și al programării concurente în conducerea proceselor industriale ;

față un cadru în care putem plasa eforturile noastre de a aduce contribuții la dezvoltarea **GA**.

1.7.3. Generatoare de aplicații folosite în domeniul instrumentației

Majoritatea programelor de instrumentație comerciale performante au facilități de **generare de aplicații**, evident relativ la clasa respectivă, de aplicații de instrumentație. Aceste programe oferă în general următoarele facilități:

- **definirea (generarea)** de aplicații de instrumentație ;
- **rularea** unor aplicații specifice de instrumentație (există la cele mai multe variante un *run-time system* care nu permite decât rularea, nu și generarea aplicațiilor);
- **adaptarea** la diverse categorii de instrumente de măsură.

Pentru lucrarea de față, interesează prima categorie de facilități. Activând aceste facilități, utilizatorul trebuie să transmită programului specificațiile sistemului de măsură dorit. Această transmitere poate fi realizată prin intermediul unei interfețe *spread-sheet* clasice [KMe93][NIn94] sau prin intermediul programării *grafice* [NIC92][NIn94]. Rezultatul (programul generat) este în cod sursă [NIm94] sau în cod executabil (dar care cere și suportul *run-time system*) ca la [KMe93] și [NIn94].

O trecere succintă în revistă a caracteristicilor câtorva astfel de programe poate fi utilă pentru subiectul lucrării de față, în vederea investigării caracteristicilor unor astfel de programe și a depistării acelor aspecte care sunt insuficient abordate și deci pot fi dezvoltate în continuare.

Câteva exemple de programe reprezentative pentru această clasă sunt:

a) **VIEWDAC** al firmei **Keithley Instruments** [KMe93] care permite rulare concurentă a mai multor aplicații de instrumentație și/sau supraveghere de proces și chiar definirea, în paralel, a noi aplicații, prin setarea unor variabile asociate unui nou task. Facilitățile de **GA** sunt oferite de interfețe de dialog de tip *spread-sheet* care permit utilizatorului descrierea aplicației de instrumentație dorite. Fiecare aplicație de instrumentație constituie un **task** autonom, care rulează din momentul în care a fost definit complet prin intermediul interfeței de dialog. Propriu-zis, utilizatorul nu are acces la programul astfel creat, deci **reutilizarea** în acest caz are semnificația refolosirii funcțiilor de bază (corespunzătoare celor definite de fiecare element de dialog în parte), într-un mod indicat de utilizator.

b) **LabWINDOWS** al firmei **National Instruments** [Nln94] permite dezvoltarea de programe de instrumentație în **C** sau **BASIC**, fiind deci un **GA** în adevăratul sens al cuvântului. Programele generate pot realiza :

- interfațarea cu o mare varietate de instrumente (prin selectarea driverului potrivit, într-o bibliotecă) ;
- interfațarea grafică (**GUI**- *Graphical User Interface*) dorită de utilizator ;
- funcții de achiziție, prelucrare și prezentare - luate din biblioteci proprii (care oferă o serie largă de servicii standard).

Transmiterea cerințelor utilizatorului se realizează prin intermediul unor *spread-sheet*-uri numite **panouri funcționale**. Se pot executa imediat facilitățile programate. **GUI** se obține imediat prin utilizarea **LabWINDOWS User Interface Editor**, care utilizează interfețe *grafice* și *meniuri pull-down*. În permanență, într-o fereastră se poate urmări modul în care dialogul cu utilizatorul modifică sau adaugă ceva codului - sursă al interfeței funcțiilor de bibliotecă selectate. Mecanismul de generare de aplicații este, foarte pe scurt, următorul :

- în funcție de aplicația selectată (din setul disponibil), **LabWINDOWS** încarcă un fișier-cadru (template) care conține un schelet al aplicației ;
- utilizatorul derulează un dialog de rafinare a cerințelor :
  - fiecare opțiune nouă, indicată printr-o cutie de dialog asociată, inserează noi linii-sursă în codul aplicației, în general acestea fiind apeluri de funcții existente în bibliotecile proprii ;
  - fiecare element de dialog dintr-o cutie de dialog corespunde valorii actuale a unui parametru de apel al unei funcții : odată aleasă valoarea dorită de către utilizator, respectiva valoare este inserată în codul-sursă, în poziția corectă :
- acțiunile utilizatorului pot fi urmărite prin schimbările survenite în codul-sursă, acesta fiind vizualizat într-o fereastră specială.

În ultimul timp, acest **GA** este din ce în ce mai mult folosit pentru dezvoltare de aplicații de **telemecanică** (supraveghere și comandă a unui proces industrial) [Ung96][Pre96]. Mediul permite acest gen de aplicații, iar un argument în plus pentru folosirea acestui mediu este acela că generează cod-sursă, ceea ce permite proiectantului de nivel înalt o prelucrare calificată, inclusiv adăugarea de noi funcții care nu pot fi obținute cu **LabWINDOWS**.

c) **LabVIEW** [NIC92][NIn94] prezintă facilităţi asemănătoare celor oferite de **VIEWDAC**, deci **generare** rapidă **de aplicaţii** de instrumentaţie, dar interfaţa de definire a aplicaţiei este mult mai spectaculoasă, ea realizându-se prin **programare grafică**. Utilizatorul îşi defineşte aplicaţia de instrumentaţie dorită, pe baza dialogului cu sistemul, utilizând *uneltele* (*tools*) sau instrumentele de interfaţă disponibile. Astfel, se pot asambla grafic module de program numite **instrumente virtuale** (**IV**). Un **IV** este alcătuit din:

- *panou frontal* (manifestarea exterioară, dorită a **IV**, de obicei asemănătoare cu panoul unui aparat de măsură);
- *diagrama-bloc*, cuprinzând comunicaţia cu modulul (instrumentul) de măsură, calcule (prelucrări), subcomponente **IV** şi blocuri de vizualizare a rezultatelor, toate fiind interconectate prin linii care reprezintă fluxul de date. Deci, prin *programare grafică*, utilizatorul defineşte un **IV** printr-o *diagramă de flux de date* ;
- icoana - asociată **IV** definit.

În Fig. 1.12 a) este prezentat un exemplu de **GUI** generat prin *programare grafică*, iar în Fig. 1.12 b), *diagrama de flux de date asociată* (**DFD**). Este interesant de aflat modul de lucru al **LabVIEW**. Câteva elemente sunt date în [Cec94]. Evident, se foloseşte conceptele **POO**, dar într-o elaborare proprie a proiectanţilor firmei **National Instruments** (*"home brewed"*). Pe suportul oferit de standardul **C**, a fost construită propria implementare **POO** a proiectanţilor, pentru că astfel, aceştia au avut libertatea de a folosi POO atât cât a fost necesar. Ei au lăsat deoparte acele caracteristici care ar fi putut să le afecteze performanţele şi le-au construit pe acelea pentru care au considerat că se justifică preţul implementării.

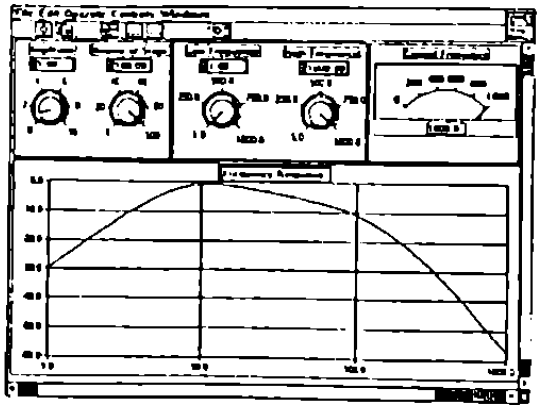


Fig. 1.12,a) **GUI** generat în **LabVIEW**

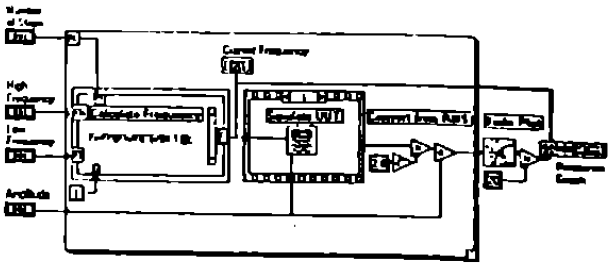


Fig. 1.12,b) **DFD** asociată

Implementarea **LabVIEW** foloseşte două concepte **POO** :

- legătura strânsă între obiecte şi metodele care operează asupra lor;
- cadrul pentru partajarea codului oferit de moştenire.

Aceste concepe se manifestă prin câteva caracteristici:

- o modalitate concisă de *trimitere a mesajelor* către obiecte şi un dispecer pentru aceste mesaje;
- un mecanism care permite unei clase să *moştenească* automat de la superclasa sa ;
- o *ierarhie de clase* care poate fi parcursă la execuţie de un al doilea dispecer pentru a permite unei clase să direcţioneze mesajele către superclasa ei.

Din implementarea respectivă lipsesc câteva caracteristici ale **POO**. Astfel, nu există o reală încapsulare a datelor. În **LabVIEW**, temporarul care defineşte variabilele de instanţă ale unui obiect este creat prin intermediul macro-urilor incluse, includerea descriind strămoşii clasei pentru obiect. În Fig. 1.13 a) şi b) se prezintă, respectiv, un obiect pentru afişarea datelor, **DDO** (***Data Display Object***) cu o ierarhie pe 4 niveluri şi exemple de instanţe ale unora din clasele de afişare (*Data Display Object, Front Panel, Numeric*). Există, în implementarea tratată, ceva similar unui *browser* de clase. Acesta tratează sistemul **POO** ca o matrice, având clasele drept linii şi mesajele coloane.

În **LabVIEW**, obiectele alocate pentru un singur document (acesta este **IV**) sunt conţinute într-o structură de date numită **ObjHeap**. Un obiect particular este referit printr-un *handle* relocabil la un **ObjHeap** şi un identificator **ObjID**.

Toate informaţiile de definire a unei clase sunt păstrate într-o structură de date numită **defProc**, care este de fapt o tabelă de pointeri la metodele clasei. Toate **defProc** sunt sursa

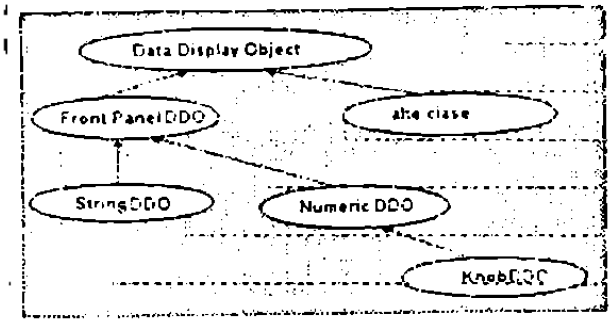


Fig. 1.13,a) Obiect pentru afişare a datelor în **LabVIEW**

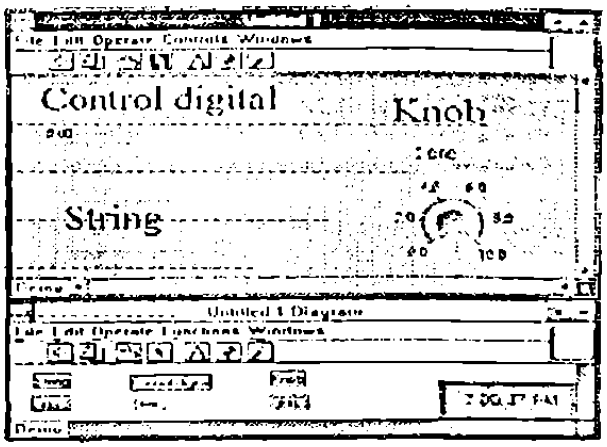


Fig. 1.13,b) Exemple de instanţe pentru clase de afişare de date

întregului mecanism **POO** propriu **LabVIEW**. Ele sunt create şi iniţializate la momentul iniţializării **LabVIEW**. Dispecerizarea mesajelor este realizată de procedura **DefProcDispatch**, scrisă în limbaj de asamblare. Întregul trafic de mesaje trece prin el.

Interesant (şi firesc !) a fost că timpul consumat la proiectare pentru construirea propriului mecanism **POO** a fost cu mult mai mare decât pentru elaborarea programului propriu-zis **LabVIEW**.

Practica a dovedit că acest program **LabVIEW** este utilizabil şi pentru dezvoltarea unor aplicaţii de telesupraveghere, ca şi **LabWINDOWS**. Într-adevăr, prin facilităţile disponibile, putem defini o foarte mare varietate de aplicaţii, facilităţile standard puse la dispoziţie permiţând o mare plajă de combinaţii, atât în ceea ce priveşte interfaţa cu procesul, cât şi în ceea ce priveşte comunicaţia cu interfaţa hardware cu procesul şi interfaţa cu utilizatorul şi prelucrările specifice. De aceea, analiza caracteristicilor acestui program este cu atât mai interesantă.

Această succintă prezentare ne permite să sesizăm următoarele :

- *din programele prezentate, doar **LabWINDOWS** generează cod - sursă, producând un rezultat "observabil" în mecanismele sale interne ;*
- *aceste programe sunt orientate spre comunicaţia cu sistemul de măsură, prelucrări şi interfaţa cu utilizatorul ;*
- *programele de mai sus sunt destinate rulării pe calculatoare de tip **PC** (sau mai puternice) şi nu acoperă gama aplicaţiilor de pe microsisteme pe 8 biţi ;*
- *nu putem utiliza facilităţile de generare de aplicaţii oferite de programele prezentate, pentru a coborî la nivelul definirii interfeţei la nivel fizic.*

Aceste observaţii surprind tot atâtea aspecte care pot fi abordate în vederea realizării unor îmbunătăţiri, extensii sau programe noi, care să ofere facilităţile care lipsesc la programele prezentate.

1.8. Instrumente CASE

Un instrument **CASE** (**Computer Aided Software Engineering** - *ingineria programării asistată de calculator*) este un mediu general care conţine instrumente de bază care suportă toate fazele ciclului de viaţă al unui produs-program [Spi95]. În paragrafele precedente s-a amintit, incidental, de asemenea instrumente, conex cu prezentarea unor exemple de generatoare de aplicaţii. În lucrarea de faţă, problematica mediilor **CASE** este interesantă în

contextul facilitării, de către majoritatea acestor instrumente, a reutilizării software, prin unele posibilități ale acestor instrumente, puse la dispoziția utilizatorului (a se vedea spre exemplu § 1.7, b - mediul **CODE/ROPE**). Acest paragraf își propune să prezinte câteva trăsături generale ale mediilor **CASE** și a modului cum acestea contribuie la reutilizarea programelor. De asemenea, sunt prezentate pe scurt câteva exemple de astfel de medii.

Instrumentele **CASE** au o logică de folosire bazată pe un anumit model al ciclului de viață al programelor sau pot să fie adaptate chiar mai multor tehnologii de programare. Orice mediu **CASE** general trebuie să cuprindă următoarele cinci componente [Spi95] :

- *capacități grafice* : mediul trebuie să permită reprezentarea grafică a diverselor aspecte ale procesului de dezvoltare de programe, adică trebuie să poată genera, edita și afișa diferitele feluri de reprezentări grafice asociate unor etape din ciclul de viață și unei anumite paradigme de abordare a proiectării.

- *unelte de prototipizare și de specificare* : instrumentele de prototipizare pot să fie folosite împreună cu capacitățile grafice ale sistemului pentru a genera automat ecrane de interfață sau pentru a modela un sistem. Trebuie să fie incluse posibilități de verificare automată a consistenței și completitudinii.

- *suport de proiectare* : include instrumente pentru asistarea proiectării și analizei unui sistem. Analiza trebuie să verifice inconsistențele, ambiguitățile și omisiunile de proiectare.

- *suport de programare și testare* : include instrumente de creare, depanare și testare de cod sursă. Într-un mediu ideal, codul sursă va fi **generat direct** din specificațiile de proiectare (deci, iată aici atribute de **generare de programe**, care sunt de obicei prezente mai mult sau mai puțin, chiar dacă mediul nu este ideal).

- *enciclopedie* : conține cunoștințe despre program (structura sa, proceduri, funcții, date, procese, etc.). Rolul enciclopediei este de a facilita gestiunea informațiilor, controlul și întreținerea (deci, gestiunea informațiilor despre o **bază software**, în înțelesul din § 1.5).

În fig. 1.14 este reprezentat printr-o schemă un mediu **CASE**, cuprinzând cele cinci componente și interacțiunile dintre ele. Dacă sunt adecvat realizate, cele trei componente instrumentale (cele din mijlocul fig. 1.14, întrucât celelalte două componente oferă doar suport pentru acestea) contribuie direct la creșterea *gradului de reutilizare*, prin posibilitatea acordată utilizatorului, de a refolosi nu numai cod sursă ci și modele din etapele de la început ale ciclului de viață (spre exemplu, specificații și modele grafice, împreună cu conexiunile și interfețele

asociate unor întregi proiecte). Regăsirea componentelor software, care constituie o dificultate majoră în cazul **SGBS**, este uşurată de facilităţile oferite de enciclopedia ataşată.

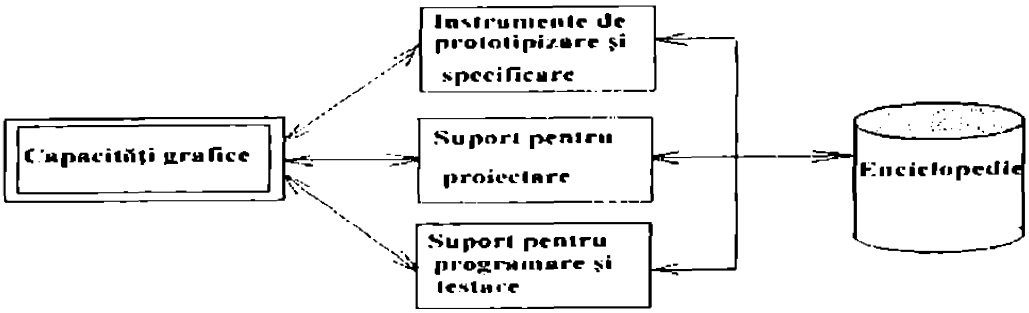


Fig. 1.14. Componentele unui mediu **CASE**

O interesantă evaluare a cererii de instrumente **CASE** pe piaţă este prezentată în [Ble94]. Se estimează astfel că piaţă aplicaţiilor **CASE** pentru grupe de lucru ("*workgoups*") va cunoaşte o importantă creştere în **1996**, cu o extindere a capabilităţilor acestor instrumente pentru asistarea proiectării aplicaţiilor concepute în tehnologii **4GL** (generaţia a 4-a de limbaje) şi o **îmbunătăţire a posibilităţilor de modelare** (cu probabilitate 0.8). Se estimează de asemenea o consolidare a pieţelor de instrumente **CASE** pentru grupe de lucru şi formarea unor nişe de piaţă (segmente specializate) în acest domeniu, până în 1997 (probabilitate 0,7). Acest studiu ne determină să afirmăm cu mare siguranţă că viitorul ingineriei programării este legat în mare măsură de dezvoltarea puternică a instrumentelor **CASE** din ce în ce mai performante, care să permită o dezvoltare rapidă de aplicaţii, pe de o parte, prin **uşurarea şi fluidizarea efortului de proiectare** datorită facilităţilor pe care le oferă suportul oferit de aceste instrumente, pe de altă parte prin **favorizarea puternică a reutilizării software**, prin facilităţile disponibile în aceste programe (spre exemplu, dicţionarele de module sau de clase, disponibile la majoritatea instrumentelor **CASE**).

Multe instrumente **CASE** sunt legate de tehnici şi concepte ale **POO** (v. cap. 2). Unele din acestea sunt însă mai generale. În cele ce urmează, enumerăm câteva astfel de instrumente :

a) **C++ Designer** [Spi95] rulează sub **Windows**, este folosit pentru construirea diagramelor obiect, conform metodologiei **OMT** [Rum91][Nie95][Spi95], care este o tehnologie orientată pe obiecte. Nu implementează toate conceptele **OMT**, dar permite modelarea claselor, atributelor, operaţiilor, relaţiilor de tip asociere, agregare şi generalizare, a calificatorilor şi rolurilor, fiecare dintre acestea constutuind o entitate specifică (articol, *item*), care se găseşte într-un aşa-numit *dicţionar de date*.

Fiecare aplicație modelată corespunde unui proiect. Un astfel de proiect conține o mulțime de diagrame, legate între ele prin relații de egalitate sau de ierarhie și prin descrierile dicționarului de date asociat. Dicționarul este parte a bazei de date asociate unui proiect. Prezentarea articolelor din dicționarul de date se realizează prin utilizarea unor formate (*templates*) standard.

Principalele facilități pe care le pune **C++ Designer** la dispoziția utilizatorului sunt :

- *manipularea unor proiecte* (creare, informații despre proiecte, ștergere, creare de versiuni, clonare, arhivare, restaurare, trecere în stare inactivă) ;
- *lucrul cu fișiere diagramă* (creare, deschidere, ștergere, redenumire) ;
- *desenarea diagramelor* (definirea unei clase - editarea clasei, a atributelor și operațiilor, etc., definirea unei relații, **reutilizarea** claselor existente în biblioteca de clase) ;
- *operații cu dicționarul* de date (căutarea - *browsing*, adăugarea de articole, editarea unui articol, ștergerea unui articol, generarea de rapoarte, etc.) ;
- ***generarea de cod*** C++ permite, pe baza modelelor-diagramă și a atributelor claselor definite, obținerea codului clasei dorite, care constituie un cadru (o capsulă de cod) pentru clasa respectivă și care conține un loc marcat "*Body to be coded here*", pentru inserarea codului specific. Deși, practic, nu a fost obținută întreaga sursă pentru clasa dorită, crearea cadrului constituie deja un mare ajutor și o importantă creștere a productivității.
- *legarea cu un mediu integrat de dezvoltare* (**IDE**) de tipul **Borland C 3.1** sau **Microsoft Visual** C++ pentru a obține codul executabil și a testa programul.

În fig. 1.15 este prezentat aspectul consolei în situația în care este afișată o diagramă.

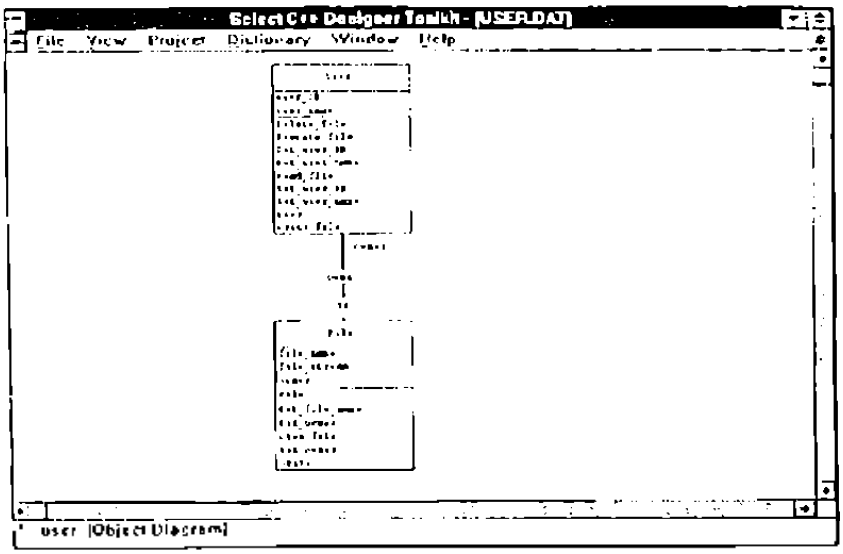


Fig. 1.15. Exemplu de diagramă afișată în **C++ Designer**

b) Select Yourdon [Coa92] este un instrument CASE care foloseşte diagrame suportate şi de tehnologii clasice, care nu sunt orientate pe obiecte : *diagrame de flux de date* (de tip *Yourdon* [Nie95] sau *Hartley*), foarte potrivite pentru reprezentări în sisteme pentru conducerea proceselor, diagrame de tranziţie a stărilor, diagrame entitate-legătură şi diagrame structurate **Constantine**. O diagramă de flux de date (**DFD**) poate fi traversată ierarhic, prin folosirea butonului *mouse*-ului. Facilităţile oferite sunt, în rest, similare celor pe care **C++ Designer** le pune la dispoziţia utilizatorului.

În fig. 1.16 este prezentat un exemplu de afişare a unei **DFD**.

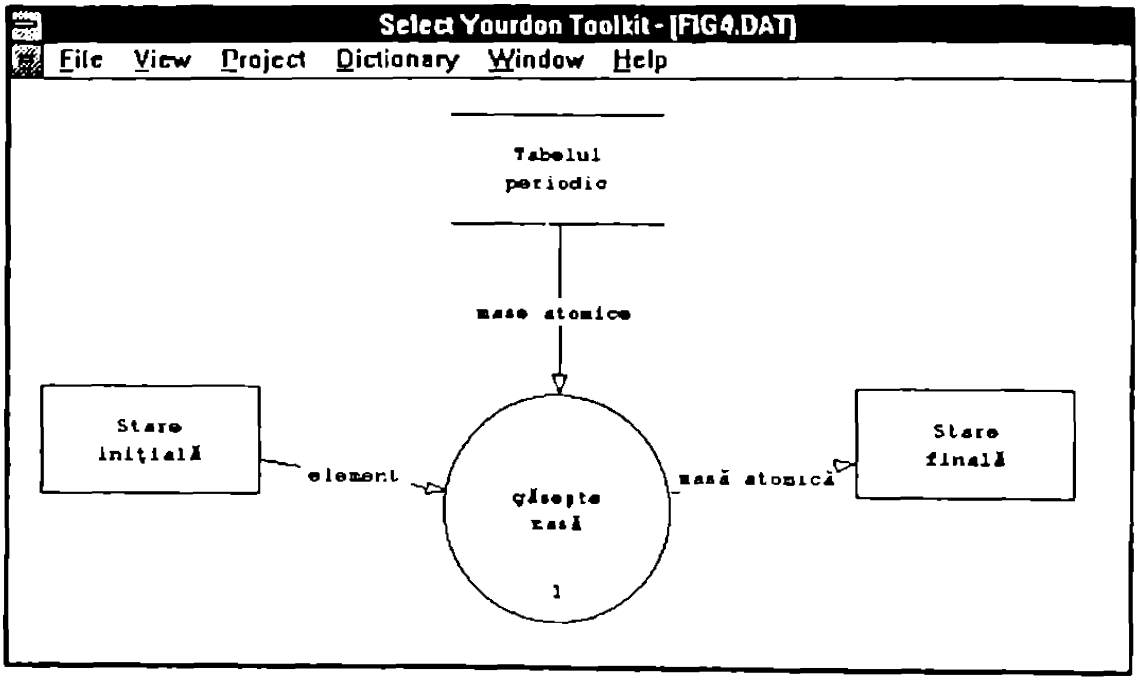


Fig. 1.16. Exemplu de afişare a unei **DFD** în **Select Yourdon**

c) **Rose** [Nie95] este conceput pentru a oferi suport metodei Booch [Boo93], utilizabilă în cadrul tehnologiilor orientate pe obiecte. **Rose** suportă *diagrame de clase şi obiecte, diagrame de arhitectură şi diagrame de proces*. Instrumentul distinge între diagramele de clase şi obiecte, şi poate fi folosit pentru etapele de analiză şi proiectare, orientate pe obiecte.

d) **AM for Windows** [CPN95] este un instrument software de dezvoltare de aplicaţii, de tip **CASE**, special conceput pentru aplicaţii pe 32 biţi, exploatând facilităţile oferite de **Windows API** (*Applications Programming Interface*). Rulează sub **Windows 95**. Este prevăzut cu multe facilităţi comune instrumentelor **CASE** - mediu grafic de interfaţă (**GUI**) incluzând dicţionare, posibilităţi de depanare, un editor înalt structurat, cu facilităţi de hipertext, şi o facilitate specială, numită **Hyperlogic**, pentru reguli şi aplicaţii din domeniul afacerilor (deci un mic sistem expert). Este potrivit, datorită instrumentelor de dezvoltare încorporate, bazate pe

filosofia **POO**, pentru dezvoltarea proiectelor mari, permițând accesul client/server pentru o gamă largă de situații precum și conexiuni cu baze de date standard și calculatoare *mainframe*.

e) **IE:Advantage** [IES93][Mcc93][Ble94] este un instrument **CASE** definit drept "universal" de proiectanții săi, în sensul că permite acoperirea cu "*unelte*" potrivite a etapelor unei întregi metodologii de proiectare, fiind conceput pentru a permite dezvoltarea de aplicații pentru diverse platforme, începând de la planificare strategică până la generarea schemei aplicației, incluzând și modelarea procesului. Instrumentul nu permite generare de cod, dar are o interfață pentru un "constructor de aplicații" disponibil pe piață, **Power Builder**. Este un instrument pentru poriectare orientată pe obiecte. Deși aria de utilizare cuprinde aplicațiile de afaceri, o sumară trecere în revistă a capabilităților programului este utilă studiului nostru.

Funcțiile de bază ale sistemului sunt :

- **gestiunea proiectelor**, care constă din două module, unul pentru întreținerea modelului ierarhic, celălalt, pentru inițierea unei sesiuni de editare ;
- **planificarea** permite introducerea descrierilor care definesc aplicația, descrieri referitoare la aspectele de afaceri pentru care este proiectată aplicația (scopuri, strategii, obiective, factori critici, și lista este extensibilă pentru utilizator) : descrierile pot fi asociate ("legate") una cu alta, precum și cu alte obiecte din *enciclopedia* programului ;
- **modelarea datelor** permite crearea și vizualizarea modelului logic al datelor, cu ajutorul unui dicționar-text sau cu o diagramă grafică de relații (hartă de date) ,
- **modelarea procesului** permite dezvoltarea unei ierarhii interne a procesului și o modelare detaliată a logicii procesului, cu ajutorul unei diagrame grafice de interdependențe ;
- **proiectarea bazei de date** care permite generarea, editarea și punerea în legătură a structurii bazei de date cu modelul logic al datelor ;

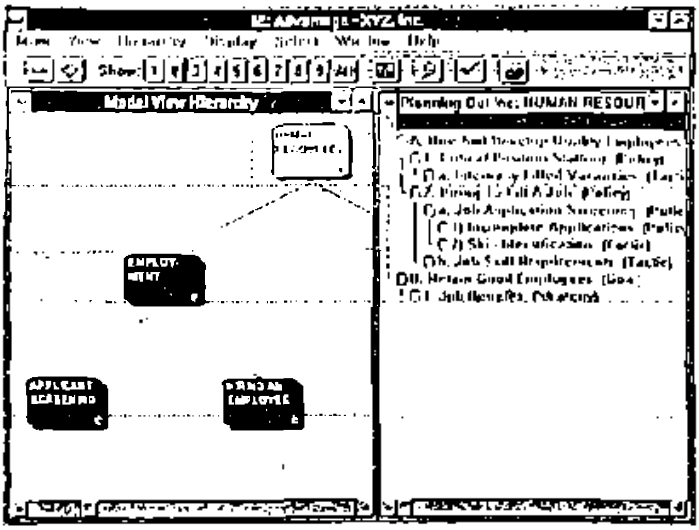


Fig. 1.17. Model ierarhic și aspect al unei descrieri din **planificare**, în **IE:Advantage**

- **construirea sistemului** ajută la generarea **SQL** (*Structured Query Language-Limbaj structurat de Interogare*) adică la realizarea componentelor de interacțiune cu utilizatorii produselor proiectate.

Instrumentul **CASE** prezentat pune la dispoziția utilizatorului o serie de facilități adiționale, precum opțiuni de raportare, **analiză calitativă a modelelor**,

utilități de import/export. Există separat un utilitar care construiește (prin "**inginerie inversă**") modelul logic al datelor pornind de la o bază de date existentă. În Fig. 1.17 este prezentat un exemplu de activare a unei opțiuni a programului.

Putem trage câteva concluzii, legate de cele câteva elemente prezentate mai sus :

- majoritatea instrumentelor **CASE** prezintă facilități în vederea creșterii gradului de **reutilizare** a programelor, prin posibilitatea oferită de a **refolosi** și schimba proiecte sau clase și ierarhii de clase, respectiv **DFD** existente ;
- de cele mai multe ori instrumentele **CASE** au posibilități de căutare după diverse criterii, în **baze software** sau baze de informații legate de proiecte și programe realizate.

Prin aceste facilități, utilizarea instrumentelor **CASE** devine foarte interesantă și importantă pentru organizațiile software, deoarece contribuie nu numai la creșterea productivității activității de proiectare ci și la creșterea gradului de reutilizare al programelor, într-un mod implicit. Cheltuielile de instruire a personalului pentru introducerea unei strategii organizaționale coerente de reutilizare se reduc astfel, întrucât același mediu de programare asigură atât integrarea operațiunilor de proiectare corespunzătoare ciclului de viață al programelor cât și a unor facilități de reutilizare. Deci proiectanții, dacă au învățat să lucreze cu instrumentul **CASE**, pe care oricum trebuie să știe să-l folosească, automat vor ști să folosească și facilitățile de **reutilizare**.

Tabelul 1.4. Atitudinea organizațiilor software față de folosirea instrumentelor **CASE**

Atitudinea față de implementarea CASE	**Procentaj [%]**
Nu a fost evaluată	11
În curs de evaluare	14
A fost evaluată, dar nu există intenții pentru implementare	12
Există dorință de implementare dar încă nu a fost realizată	14
Intenții de implementare în următoarele 24 luni	1
Intenții de implementare în următoarele 12 luni	4
Implementat, utilizat pe scară restrânsă	30
Implementat, utilizat pe scară largă	14

Dezvoltarea instrumentelor **CASE** este deosebit de rapidă. Un studiu la nivelul anului 1991 [BRG91] denotă că pentru un eşantion de 200 de organizaţii, distribuţia opticii faţă de implementarea acestor instrumente în practica organizaţională, este prezentată în Tabelul 1.4.

În prezent, instrumentele **CASE** sunt folosite curent, cu atât mai mult cu cât medii puternice de programare au înglobate caracteristici **CASE**, iar multe organizaţii software au implementate strategii coerente de inginerie a programării, care presupun şi folosirea intensivă a acestor instrumente.

1.9. Reutilizarea în sistemele în timp real

Programele în timp real sunt dificil de realizat, datorită cerinţelor specifice. Evident, dificultatea de proiectare duce la aspecte deosebite şi dificultăţi specifice, în eventualitatea abordării proiectării prin prisma reutilizării **software**. O analiză a acestor aspecte se impune în lucrarea de faţă, întrucât este vorba de acele aplicaţii care sunt preponderent urmărite pe parcursul lucrării.

Programele în timp real sunt o categorie de programe a căror corectitudine depinde nu numai de aspectele *funcţionale* ci şi de cele *temporale* (timpul de obţinere a rezultatelor prelucrărilor). O analiză pertinentă a dificultăţilor, problemelor şi provocărilor, cu referire şi la domeniul de interes al lucrării de faţă, este realizată în [Sta88]. Din analiza efectuată se desprind unele concluzii, prezentate în continuare. Primele concluzii de menţionat se referă la **dificultăţi:**

- Dezvoltarea sistemelor în timp real costă mult şi restricţiile de timp sunt verificate de obicei prin tehnici ad-hoc sau prin simulări extensive şi costisitoare ;
- Integrarea componentelor este extrem de dificilă şi deci măreşte o dată în plus costul sistemului .
- Schimbări mici în sistem conduc spre noi etape de testare extensivă
- Cerinţa de **reconfigurabilitate** a numeroase aplicaţii din această categorie, care sunt concepute pentru medii evolutive, nu statice. Cerinţele de reconfigurabilitate duc la soluţii asemănătoare cu cele adoptate pentru creşterea gradului de reutilizare.

"Tehnicile curente ale «forţei brute» nu satisfac cerinţele de garantare a constrângerilor de timp ale sistemelor generaţiilor viitoare". Textul citat din lucrarea [Sta88] datează din 1988. De atunci, atât tehnica de calcul cât şi ingineria programării, prin apariţia pe piaţă a instrumentelor **CASE** relativ performante, au evoluat. Dar dificultăţile de dezvoltare ale sistemelor în timp real nu au dispărut, cu atât mai mult cu cât evoluţia tehnicii de calcul şi a

cerinţelor, mai ales prin necesităţile de înglobare în sistemele de conducere în timp real a unor aspecte de inteligenţă artificială, nu au făcut decât să crească aceste dificultăţi.

Următorul aspect interesant în legătură cu subiectul abordat se referă la **direcţiile viitoare** de dezvoltare a abordărilor de creştere **a reutilizabilităţii** în domeniul aplicaţiilor în timp real :

- În domeniul **limbajelor-suport** ale sistemelor în timp real, o posibilă îmbunătăţire a reutilizabilităţii este obţinerea modulelor pentru programe în timp real reutilizabile, domeniu caracterizat drept "subiect important", întrucât reduce preţul de cost şi creşte calitatea programelor care rezultă. Practic, dezvoltarea specificată înseamnă un efort de a defini sau extinde limbaje de programare astfel încât acestea să faciliteze efortul de obţinere **a modulelor reutilizabile** pentru sisteme în timp real. Această dezvoltare este însoţită de inerente dificultăţi, deoarece modulele reutilizabile în sisteme în timp real sunt mai dificil de obţinut şi de readaptat, datorită restricţiilor diferite de timp pentru aplicaţii diferite, care trebuie să fie respectate de diversele module de program. Încrederea în tehnicile uzuale de codare inteligentă "cu mâna" şi restricţiile de timp greu de urmărit sunt surse majore de erori în programele în timp real, mai ales la modificarea sistemelor mari.

- Un obiectiv primar al cercetărilor în ingineria programării sistemelor în timp real este de a **automatiza sinteza de cod** eficient şi adaptarea la cerinţele clientului a mecanismelor de timp real, mai ales a specificaţiilor şi aspectelor ce privesc restricţiile de timp. Cu alte cuvinte, e vorba de dezvoltarea unor **GA** specifice, respectiv a unor interfeţe care să permită o rapidă şi facilă descriere de către **UF** a specificaţiilor şi restricţiilor.

- În ceea ce priveşte **programarea în limbaj de asamblare**, programarea handlerelor de întreruperi şi a driverelor de dispozitive, care de asemenea sunt părţi integrante din sisteme în timp real, constituie probleme ştiinţifice rezolvate. Aici mai e loc de dezvoltări viitoare doar în ceea ce priveşte automatizarea proiectării, deci din nou este vorba de dezvoltarea unor **GA** specifice.

- O direcţie în care se pot realiza în continuare investigaţii o constituie găsirea unor convenţii, **forme de exprimare a dimensiunii timp**, inclusiv pentru etapele de specificare, verificare şi analiză, în ceea ce priveşte restricţiile de timp.

Direcţiile enumerate constituie tot atâtea posibilităţi de contribuţii originale în domeniu, deci posibile deschideri pentru demersuri teoretice şi practice, care să ţină însă cont şi de dificultăţile enumerate.

1.10. Modele în proiectarea pentru reutilizare

În acest paragraf, ne concentrăm demersul asupra analizării câtorva abordări teoretice ale modelării procesului de proiectare software în sine, cu accent pe proiectarea pentru **reutilizare**. Este desigur o strânsă legătură între modelarea procesului de proiectare şi modelele pentru ciclul de viaţă al programelor. Legătura este subliniată spre exemplu în [Mat84], prin aceea că, plecând de la o abordare şi o clasificare "încetăţenită" începând cu lucrarea [Dij72], referitoare la modele ale ciclului de viaţă, autorul propune o întreagă metodologie de **proiectare pentru reutilizare** (Anexa 1.1, 5). Întrucât clasificările şi nomenclatura aferentă descrierii ciclului de viaţă al programelor sunt încetăţenite şi uzuale în ingineria programării, în cele ce urmează nu prezentăm decât modelele referitoare strict la proiectarea ca şi proces. De altfel, o legătură sugestivă între reutilizare în proiectare şi ciclul de viaţă al programelor este prezentată printr-o diagramă sugestivă, în lucrarea [Kru90] (v. fig. 1.2). Această diagramă va sugera de altfel şi modelul descris în Cap. 5 al acestei lucrări.

Să trecem deci în revistă unele modele ale proiectării :

a) un prim model este cel menţionat anterior, fiind prezentat în [Dij72] (Anexa 1.1, 5). Reamintim câteva elemente : un program **P(i)** pe nivelul abstract **i** şi conceput pentru a fi rulat pe maşina **M(i)** suferă prin transformare spre un nivel mai jos **i+1**, o modificare în **P(i+1)** care poate fi executat pe maşina **M(i+1)**. O fază a procesului de rafinare este un pas care transformă **P(i)** în **P(i+1)**. Procesul de proiectare software peste **N** nivele abstracte poate fi reprezentat prin relaţii de tipul :

Form (i) := Trans(i, Form(i-1))

(rel. (A1.1), cu notaţiile respective, v. Anexa 1.1, 5)

Această modelare subliniază caracterul **dinamic** al proiectării software, iar rel. (A1.1) sugerează un model discret de "*stare*", sub forma cunoscută din teoria sistemelor.

b) În articolul [Che84], analizat anterior (Anexa 1.1, 9), care se ocupă de prototipizarea rapidă, autorul încearcă şi formalizarea tehnicii propuse.

Fiind dat un program abstract, pentru o anumită aplicaţie, trecerea acestuia la programul concret este realizată urmărind două mecanisme : definire şi transformare. Trecerea ("rafinarea") se poate realiza în mai multe etape, obţinându-se reprezentări "tot mai concrete". Prin **definire** se înţelege atribuirea unor constrângeri (sau valori) pentru proceduri, tipuri, date, etc. Prin **transformare** se înţelege înlocuirea unor construcţii de nivel înalt prin construcţii "*mai concrete*" care realizează funcţia dorită. O regulă de transformare constă dintr-o formă

sintactică, urmată eventual de predicate semantice şi o parte de înlocuire. Un **instrument de transformare** este alcătuit dintr-o *colecţie de entităţi abstracte de program, o mulţime de transformări şi o secvenţă de instrucţiuni*. Acestea din urmă determină instrumentul să aplice diverse transformări asupra entităţilor de program **P**, într-o nouă mulţime de entităţi **P'**:

$$\mathbf{P \times T \longrightarrow P'} \qquad (1.2)$$

Programul final, concret, rezultă după mai multe etape de transformare. Instrucţiunile pot să se refere la transformări asupra tuturor entităţilor de program, sau pot să specifice o secvenţă de transformări locale asupra unor entităţi sau submulţimi de entităţi.

c) Articolul [Lit84] este dedicat fundamentării formale a proiectării pentru reutilizare, folosind limbajul **ADA**. După cum s-a arătat în Anexa 1.1, 8, reutilizarea în viziunea autorilor trebuie divizată sub aspect formal în aspecte **intermodul** şi aspecte **intramodul**. Formalismul prezentat în articol se ocupă cu predilecţie de aspectele privind modelarea intermodul, aici fiind centrul de greutate al proiectării pentru reutilizare, în viziunea autorilor.

Mediul unei componente poate fi exprimat ca o structură algebrică constând într-o mulţime de **sorturi** (tipuri de identificatori) şi operaţii asociate. Acestea reprezintă aspecte ale sistemului de care componenta are nevoie pentru a-şi îndeplini funcţiile. Într-un sistem anume, fiecare componentă este legată de un specific local. Ea poate fi reutilizată într-un alt context asigurat de mediul respectiv, dacă conţine un sub-mediu care poate realiza mapările (corespondenţele) în mediu cerute de componentă. Articolul citat formalizează această abordare pentru reutilizare, prin folosirea **teoriei categoriilor**. Iată câteva dintre conceptele de bază folosite, preluate din [Lan71] :

- **Categoria** este definită ca o clasă de obiecte asociată cu o mulţime de corespondenţe numite **morfisme**, definite pentru fiecare pereche de obiecte din categorie (definiţia riguroasă este mai generală, dar pentru studiul nostru, cea dată este potrivită). Pentru o categorie **C** , notăm o clasă de obiecte prin **obj(C)**, iar morfismele prin **mor(C)**. Fiecare morfism **F** din **mor(C)** are o sursă **s(F)** şi o ţintă **t(F)**, ambele din **obj(C)**. Morfismele pot fi compuse oridecâteori ţinta unuia este sursa altuia. Morfismul identitate este notat cu 1_X. Categorii uzuale sunt :
  - S : obj(S) = clasă de mulţimi ; mor(S) – corespondenţe de mulţimi
  - V : obj(V) = clasă de spaţii vectoriale reale ; mor(V) – transformări liniare
  - G : obj(G) = clasă de grupuri ; mor(G) = homomorfism de grupuri

- Reprezentarea standard pentru vizualizarea relaţiilor în cadrul categoriilor este **diagrama**, un graf orientat ale cărui noduri corespund obiectelor din **obj(C)**, iar arcele corespund morfismelor din **mor(C)**. Se spune că o diagramă **comută** dacă pentru oricare obiecte **A** şi **B** din **obj(C)**, oricare două compuneri ale morfismelor de la **A** la **B** sunt morfisme egale.

- **Coprodusul** a două obiecte **A** şi **B** din **obj(C)** este un element **P** din **obj(C)** şi morfismele I_A : **A** --> **P** şi I_B : **B** --> **P** care satisfac următoarea *proprietate generală* : pentru orice **Q** din **obj(C)** şi morfismul J_A : **A** --> **Q** şi J_B : **B** --> **Q**, există un unic morfism **U** : **P** --> **Q** astfel încât diagrama din Fig. 1.18, a) comută (adică $U \circ I_A = J_A$ şi $U \circ I_B = J_B$).

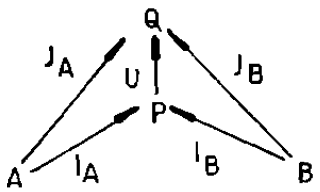


Fig. 1.18, a. Coprodusul

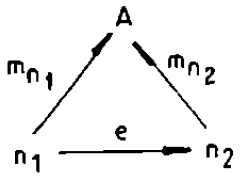


Fig. 1.18, b. Con peste arc al diagramei

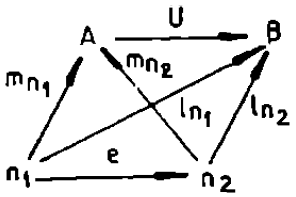


Fig. 1.18,c. Colimita

Intuitiv, imaginile posibilelor morfisme în **Q** sunt tipurile şi operaţiile în **A** şi **B**. De aceea, imaginile I_A şi I_B în **P** trebuie să reprezinte trăsăturile definitive pentru **A** şi **B**, luate împreună. Bineînţeles că noţiunea "trăsături definitive" este relativă la categoria referită.

- Să presupunem că avem diagrama **D** = (**N**, **E**), unde **N** este mulţimea nodurilor şi **E** este mulţimea arcelor. Un **con** pe **D** constă dintr-un obiect **A** şi o mulţime de morfisme m_n : **n** --> **A**, indexate după nodurile **n** din **D**, astfel încât pentru oricare arc **e** din **E**, diagrama din Fig. 1.18, b) comută (în figură, arc de la n_1 la n_2).

- **Colimita** lui **D** = (**N**, **E**) este un con (**A**, **M**) unde **M** constă în m_n : **n** --> **A**, indexat după nodurile **n** din **N**, care satisfac următoarea "proprietate generală" : pentru oricare con (**B**, **L**) unde **L** constă în l_n : **n** --> **B**, există un unic morfism **U** : **A** --> **B**, astfel încât pentru oricare arc **e**, diagrama din Fig. 1.18, c) comută (în figură, arc de la n_1 la n_2). Deci, colimita este conul cel mai "apropiat" de diagramă. Intuitiv, colimita este, asemenea coprodusului, o mulţime a trăsăturilor definitive ale diagramei. În acest caz, într-adevăr, corespondenţele din diagramă identifică aspecte ale structurilor care sunt reprezentate printr-o singură imagine în colimită.

Pe baza acestor noţiuni, putem contura concepte importante pentru formalizarea care ne interesează. Astfel, un **functor** este o corespondenţă de la obiecte şi morfisme ale unei categorii la obiecte şi morfisme ale alteia, astfel încât morfismele sursă şi destinaţie şi operaţiile de identitate şi compunerile sunt păstrate (deci se păstrează relaţiile între obiecte). În consecinţă, diagramele de comutare corespund la diagrame de comutare, coprodusele corespund la coproduse, etc.

Modelul propus în articol consideră **Unităţile Generice din ADA ca nişte functori**. Clasa mediilor permise pentru o componentă este definită ca o clasă de izomorfisme de structuri algebrice multi-sort. Categoria de **structuri algebrice multi-sort** constă în morfisme care sunt corespondenţe care conservă structura ; adică, se păstrează corespondenţa sort la sort, respectiv, identificatori de operaţii la identificatori de operaţii. Includerea unei componente într-un sistem poate fi văzută ca un morfism de la mediul existent înainte de includerea într-un nou mediu. Un tip parametrizat de date, sau constructor de tip de date, poate fi deci gândit ca o încorporare a unui functor între categoriile de structuri algebrice având substructuri (parametrii actuali de tip de date) determinate de izomorfism. Pentru **ADA**, un astfel de tip de date parametrizat poate fi implementat prin pachete generice. Când are loc instanţierea, este practic creat un nou mediu conţinând structurile algebrice făcute disponibile prin specificaţiile pachetului generic.

Un pachet generic **ADA** poate fi implementat ca un modul conţinând instanţieri ale pachetelor generice de nivel mai scăzut. Un întreg sistem **ADA** poate fi implementat ca o ierarhie încuibată a unor astfel de instanţieri generice. Fiecare nivel al sistemului poate fi conceput ca o compunere de morfisme corespunzând fiecărei instanţieri generice la acel nivel. Fiecare nivel poate fi reprezentat ca o diagramă. Fiecare din aceste morfisme şi compunerea lor sunt de fapt instanţe de functori. Aceşti **functori constructori algebrici** corespund componentelor independente de nivel ale sistemului **ADA**. Descompunerea unui pachet generic **ADA** într-o secvenţă de instanţieri generice de nivel scăzut este ca însăşi un morfism, numit **morfism de descompunere**. În Anexa 1.1, 7 sunt prezentaţi operatori ai acestei modelări, capabili să ofere un suport teoretic pentru proiectarea pentru reutilizare.

d) În lucrarea [Sav96] este propusă o schemă care modelează procesul de proiectare în general Am considerat că prezentarea acestei scheme este utilă în contextul lucrării de faţă, întrucât schema insistă asupra procesului iterativ de ajustare/reluare a proiectării, prin compararea rezultatelor (performanţelor) produsului obţinut, cu cerinţele. Această schemă sugerează un model sistemic, cu reacţie, care de altfel va fi dezvoltat în Cap. 5. Schema respectivă este prezentată în Fig. 1.19.

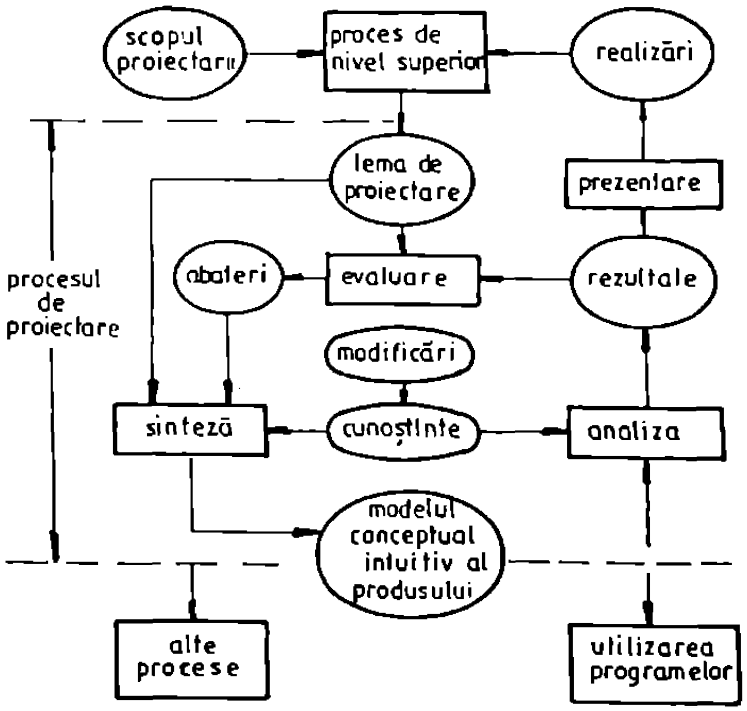


Fig. 1.19. Schema procesului de proiectare cu luarea în considerare a abaterilor

Schema din figură este oarecum asemănătoare cu schema din Fig. 1.2 şi reflectă modul în care abaterile de la tema de proiectare, rezultate în urma procesului de evaluare a rezultatelor proiectării, duc la sinteza elementelor necesare reluării procesului de proiectare, în vederea eliminării acestor abateri.

Modelele prezentate mai sus constituie un bun punct de plecare pentru abordările teoretice ale proiectării pentru reutilizare, dezvoltate în Cap. 5. Astfel, primul model subliniază caracterul ***dinamic*** *al proiectării software (pentru reutilizare), al doilea model propune* ***transformarea*** *ca denumire pentru etapele succesive ale proiectării, următorul model propune o* ***fundamentare formală*** *a procesului proiectării folosind noţiuni de* ***teoria categoriilor*** *iar aspectul interesant pentru lucrarea de faţă, din ultimul model prezentat, constă în reprezentarea* ***comparării*** *rezultate-lor proiectării cu specificaţiile iniţiale, în cadrul unei* ***scheme cu reacţie.*** *Nu toate modelele analizate sunt specifice strict proiectării pentru reutilizare, dar pot furniza idei pentru construirea unui* ***model sistemic*** *pentru acest proces de proiectare.*

1.11. Aspecte generale ale reutilizării

Întrucât prezentul capitol este dedicat realizării unei baleieri a bibliografiei referitoare la reutilizarea software (în vederea mai bunei încadrări a preocupărilor proprii) deci are în primul rând un scop de luare de contact şi explorare de domeniu, este firesc să consacrăm un spaţiu mai aplu extins decât uzual, unor aspecte generale legate de reutilizare, unele din acestea sub forma unor concluzii care să prefigureze astfel sau pe alocuri chiar să contureze direcţiile de dezvoltare în cadrul tezei, ale subiectului abordat. Astfel, ne referim aici la avantaje şi dezavantaje ale reutilizării, clasificări ale modalităţilor de reutilizare, concluzii referitoare la utilizarea **SGBS**, factori care influenţează reutilizarea, orientări, concluzii referitoare la reutilizarea pentru sistemele în timp real.

1.11.1. Clasificarea modalităţilor de reutilizare software

Pentru ordonarea conceptelor de bază ale **IPR**, este necesară o clasificare a modalităţilor de reutilizare. Clasificarea aceasta este utilă şi pentru poziţionarea propriilor contribuţii în contextul general. Prin analiza unui număr mare de abordări, precum cele prezentate în paragrafele precedente ale prezentului capitol, admiţând diferite criterii, putem realiza diverse clasificări. Două dintre acestea sunt prezentate mai jos : unul analitic, **metodologic**, celălalt sintetic, **generic**.

A. Criteriul **metodologic** este un criteriu de clasificare natural, cu fundamente empirice, rezultă din prelucrarea unui mare număr de abordări şi din ordonarea acestora prin gruparea după caracteristicile comune şi după metodele folosite. Din acest punct de vedere, cea mai potrivită clasificare se poate considera cea prezentată în Tab. 1.5 după [Big84] - *clasificarea Biggerstaff* - realizată în bună măsură prin analizarea articolelor prezentate în Anexa 1.1. Sunt prezentate sintetic diversele aspecte ale reutilizării, din perspectiva metodologică.

Diversele abordări ale reutilizării software pot fi încadrate în două mari grupe : folosirea **blocurilor de construcţie reutilizabile - BCR** ("*reusable building blocks*") şi folosirea **modelelor reutilizabile - MR** ("*reusable patterns*").

a) În cadrul grupei **BCR**, componentele ce vor fi refolosite sunt organizate şi combinate după reguli bine definite. **Atomicitatea** componentelor se referă la conservarea identităţii acestora în diverse aplicaţii. Exemple posibile sunt "scheletele" de cod, subrutine, funcţii, clase, etc. Compunerea se poate realiza prin interconectare serială (mecanisme "**pipe**") sau prin mecanisme **POO** : pasarea de mesaje şi moştenirea, adică un mecanism de dispecerizare a mesajelor, care determină funcţia apelată în momentul execuţiei, prin asocierea unui mesaj la un anumit obiect. În literatura de specialitate există abordări care pun accent pe acumularea de componente, în care caz apar probleme de gestiune a acestora (s. ex. **browsere** de module, clase, etc.), a comprehensibilităţii şi a constrângerilor, precum şi alte abordări care pun accentul pe regulile de compunere şi organizare (un capitol al reutilizării extrem de vast şi de dinamic, **POO** fiind o ilustrare exemplară a acestui concept).

b) A doua categorie de abordări - **MR** - este mai dificil de caracterizat, întrucât aici reutilizarea constă în a folosi elemente active, care ele însele generează programul-ţintă. Deci refolosirea este aici o chestiune mai degrabă de execuţie decât de manipulare. Fie că modelele sunt conţinute în instrumentul (programul) folosit (s.ex. **generatoarele de aplicaţii**), fie că instrumentul conţine doar reguli de transformare (**sistemele de transformare**), este efectiv dificilă identificarea directă a componentelor reutilizate. Aspectul de reutilizare este aici "global" şi "difuz". Aici sunt cuprinse **metalimbajele** - limbaje de nivel foarte înalt, pentru scopuri generale, şi **limbaje orientate pe problemă** - pentru domenii mai restrânse (care sunt de fapt **limbaje de specificaţii**), **generatoarele de aplicaţii**, care genercază cod (sursă sau executabil) pe baza unui dialog cu utilizatorul, respectiv **sistemele de transformare**, care conţin reguli de transformare atât asupra programelor cât şi asupra datelor (s.ex **translatoarele de cod**). Clasificarea prezentată trebuie să fie întregită cu referiri la instrumentele **CASE** (v. § 1.8), utilizarea cărora, după cum am arătat în paragraful respectiv, contribuie direct la creşterea **reutilizabilităţii** programelor.

Instrumentele **CASE** includ, de cele mai multe ori (după cum s-a arătat la § 1.8, facilităţi de manipulare a mai multor concepte şi tehnici din tabelul de mai jos, de aceea, într-o eventuală

completare a acestuia, instrumentele **CASE** ar trebui să ocupe o întreagă linie, sub toate coloanele. Numim această clasificare completată, clasificarea **Biggerstaff+**.

Tabelul 1.5. Clasificarea **Biggerstaff+**

Componente refolosite	**Blocuri (BCR)**		**Modele (MR)**		
Natura componenetelor	Atomice şi nemodificabile Pasive		Difuze şi modificabile Active		
Principii de reutilizare	Compunere		Generare		
Accentul	Biblioteci de componente de aplicaţii	Principii de compunere şi organizare	Generatoare bazate pe limbaje	Generatoare de aplicaţii	Sisteme de transformare
Sisteme tipice	Biblioteci de subrutine	- Arhitecturi "pipe" - **POO**	-Limbaje de nivel foarte înalt -Limbaje orientate pe probleme	Generatoare de interfeţe	Translatoare de cod
	← CASE →				

B. Criteriul **generic** de clasificare a metodelor de reutilizare, propus în [Fre94], este structurat pe aspecte de evoluţie şi complexitate. Această clasificare porneşte de la constatarea prealabilă că diversele abordări ale reutilizării sunt influenţate atât de evoluţia limbajelor de programare cât şi de evoluţia metodelor de dezvoltare a produselor-program (adică de dezvoltarea ingineriei programării însăşi). În lucrarea citată posibilităţile de reutilizare sunt divizate în 3 categorii, iar în cadrul acestora, putem pune în evidenţă, conform clasificării propuse, mai multe **posibilităţi** de reutilizare, constând în utilizarea entităţilor software notate cu litere de la de la a) la d).

Clasificarea propusă este următoarea :

1. Reutilizare de **speţa 1** - care înseamnă refolosirea entităţilor software în alte programe, fără posibilitatea unor modificări :

a) **biblioteci de rutine** ; în această abordare, utilizatorul este un simplu client al resurselor disponibile, neputând modifica algoritmii care caracterizează funcţionalitatea modulelor software; caracteristica esenţială a acestei metode este că pune accentul pe algoritmi (operaţii) ; un exemplu al acestui gen de reutilizare software îl constituie "bătrânele" biblioteci de subprograme **FORTRAN**, pentru diverse calcule matematice ; tot acest concept îşi are codificarea teoretică în rel. (1.1,a).

b) **pachetele (modulele)** grupează, pentru o structură de date considerată, operaţiile proprii acestei structuri ; acestea sunt specifice limbajului **ADA** (numite aici *pachete*), limbajului **MODULA** (*module*), **Turbo Pascal** (*units*), etc. ; în sine, acestea sunt unităţi sintactice independente, conţinând atât reprezentarea, cât şi operaţiile unui tip de date utilizator şi pot fi compilate separat.

2. Reutilizare de **speţa a II-a** - care delimitează acele categorii de reutilizare care presupun folosirea de instanţe ale unui program generic, prin instanţierea după un set de parametri, dar fără modificarea propriu-zisă a pachetului generic :

c) **pachete generice** - sunt pachete parametrizate, de obicei parametrul fiind tipul de date supus prelucrării ; asemenea abordări sunt uzuale în **ADA** şi **C++**.

3. Reutilizare de **speţa a III-a** - referitoare la acele tehnici care presupun crearea de clase derivate, pe baza celor existente :

d) **clasele** - constituie un pas uriaş înainte în reutilizare (de aceea, în economia lucrării de faţă fac obiectul unui capitol separat), prin posibilitatea oferită de **POO** de a folosi moştenirea pentru a crea ierarhii de module (clase) în care caracteristicile sunt moştenite şi posibil îmbunătăţite la nivelul descendenţilor. Utilizatorul poate astfel crea noi descendenţi (clase derivate) pentru care el trebuie să precizeze doar acea parte a comportamentului care diferă de cel al părinţilor. Pe de o parte, clasele derivate moştenesc elementele specifice comportamentului părinţilor, iar pe de altă parte, noua ierarhie, îmbogăţită, este total compatibilă cu vechile utilizări, întrucât clasele deja existente nu au fost modificate. Astfel, utilizatorul are un rol activ, adaptându-şi ierarhia existentă la necesităţile proprii. Aspecte asociate acestui gen de reutilizare vor fi prezentate într-un capitol separat al lucrării, de aceea aici nu mai aprofundăm acest subiect.

Se observă că această din urmă clasificare a reutilizării este privită mai ales prin prisma evoluţiei tehnicilor de **POO**. Întrucât lucrarea de faţă nu îşi propune să contribuie cu realizări proprii în acest domeniu, vom utiliza cu predilecţie clasificarea **Biggerstaff**. Un argument în plus îl constituie faptul că această clasificare permite o mai bună localizare a contribuţiilor proprii în contextul general al reutilizării.

1.11.2. Factori ce influenţează reutilizabilitatea

Reutilizabilitatea este influenţată, în mod pozitiv sau negativ, de factori de natură organizatorică, metodologică sau legaţi de limbajul de programare folosit [Fre94]. Astfel, putem enumera aici :

1. factori de natură **organizatorică** :

a) disponibilitatea codului modulelor software implicate în efortul organizaţional pentru reutilizare (cod sursă, obiect, biblioteci);

b) existenţa, la nivelul firmelor implicate în activităţi de proiectare software, a unei politici coerente de realizare a depozitelor de componente reutilizabile;

2. factori de natură **metodologică** (care sunt determinaţi de metodele şi tehnicile folosite în proiectare) :

c) facilităţile puse la dispoziţie de limbajul de programare folosit pentru dezvoltarea aplicaţiilor, în ceea ce priveşte **modularizarea** şi **abstractizarea**;

d) existenţa mecanismelor de parametrizare a tipurilor de date (trăsătură numită **genericitate**);

e) folosirea orientării spre obiecte, implicit trăsăturile respective (**moştenire, polimorfism**);

3. factori **subiectivi** :

f) experienţa şi abilitatea proiectantului.

Se observă că abordarea implementării conceptelor de bază ale reutilizării, la nivel organizaţional, presupune luarea în considerare a multor factori.

1.11.3. Avantaje şi aspecte asociate ale reutilizării

O politică organizaţională coerentă de reutilizare software are mari avantaje. Acest lucru este cunoscut de toate firmele producătoare de software şi se poate pune în evidenţă o mare implicare a acestora în această orientare. De aceea considerăm şi în lucrarea de faţă ca fiind importantă sublinierea câtorva avantaje ale reutilizării. Astfel, putem justifica eforturile de a aduce contribuţii originale în acest domeniu de activitate.

Refolosirea componentelor software "amplifică" capabilităţile proiectantului software. O serie de avantaje, unele aproape subânţelese, atrag după sine eforturi însemnate de a dezvolta

metode, tehnici şi instrumente ale ingineriei programării, care să ducă la creşterea gradului de reutilizare al programelor. Aceste avantaje sunt [Fre94] .

- costuri de dezvoltare ale produselor software, scăzute (dacă ne raportăm la o perioadă rezonabilă de timp) ;
- calitate îmbunătăţită a produselor software obţinute (întrucât se presupune că modulele reutilizate au fost deja testate, deci funcţionează corect);
- întreţinerea programelor, mai puţin costisitoare;
- reducerea complexităţii programelor întrucât trebuie realizate numai componentele specifice domeniului particular de utilizare.

Există însă o serie de motive pentru care reutilizarea pe scară largă nu este de loc simplă. Iată mai jos câteva dintre aceste motive :

1 - realizarea componenetelor reutilizabile este mai dificilă,
2 - în proiectarea pentru reutilizare este nevoie de o mare capacitate de abstractizare şi generalizare şi de multă experienţă pentru conceperea componenetelor reutilizabile ;
3 - avantajele reutilizării, în ceea ce priveşte costurile reduse, nu apar decât în timp (de aceea, la avantajul care se referă la aceste costuri reduse, de mai sus, apare în formulare "perioadă rezonabilă de timp"), deoarece în proiectarea software pentru reutilizare există cheltuieli suplimentare de dezvoltare a produselor, cheltuieli care se vor amortiza doar după realizarea mai multor produse în care componentele sunt refolosite (astfel, preţul de dezvoltare pentru astfel de produse este cu 30 - 200 % mai ridicat decât cel al aceloraşi programe, scrise fără a se urmări reutilizarea) : cu alte cuvinte, reducerea de cost se manifestă doar într-un timp suficient de lung pentru ca să se amortizeze cheltuielile iniţiale mari ;
4 - este necesară reorganizarea proiectării, a muncii în echipă, schimbarea stilului de programare;
5 - sunt necesare noi instrumente pentru gestionarea depozitelor de componente reutilizabile.

Astfel de depozite, care conţin ierarhii de module reutilizabile, există în practică sub formă de cadre (în literatură **"frameworks"** - acestea fiind ierarhii de clase pentru interfeţe utilizator - spre exemplu **TurboVision** pentru **Pascal** sau **C++**) şi de instrumente asociate - **"tool-kits"** - care permit integrarea acestora în procesul proiectării.

Alte câteva concluzii interesante, de factură empirică, în privinţa proiectării pentru reutilizare sunt sintetizate în [Tra88] :

- înainte de a dezvolta software reutilizabil, trebuie să-l fi dezvoltat şi folosit de cel puţin 3 ori, respectiv trebuie să fi dezvoltat programe asemănătoare (din aceeaşi clasă) după metodologii şi tehnici convenţionale, fără a se ţine cont de cerinţele de reutilizare (a se vedea şi observaţia 2 de mai sus);
- beneficiile reutilizării apar după cel puţin 3 refolosiri (v. observaţia 3 de mai sus).

1.11.4. Orientări strategice referitoare la folosirea reutilizării

Din cele prezentate rezultă următoarele orientări pentru folosirea curentă a tehnicilor de reutilizare şi organizarea activităţii de proiectare pentru reutilizare [Fre94] :

- organizarea componentelor reutilizabile în biblioteci organizate ierarhic şi bazate pe abstractizarea datelor;
- limbajul de programare folosit trebuie să posede cel puţin facilităţi de modularizare;
- programatorii trebuie formaţi în ideea utilizării abstractizării datelor şi reutilizării software;
- la nivel organizaţional, reutilizarea trebuie privită ca politică generală şi administrată în ideea recuperării costurilor promovării ei;
- sunt necesare noi metodologii şi instrumente de gestionare a depozitelor de componente reutilizabile, precum şi tehnici şi instrumente de partajare a depozitelor între diferiţi utilizatori.

Cele câteva aspecte ale organizării activităţii software pentru reutilizare privesc în primul rând nivelul *managerial* al entităţii organizatorice de proiectare software (firma, colectivul, etc.), adică nivelul la care se iau decizii privitoare la organizarea activităţii şi gestiunea resurselor. Este clar, din cele câteva orientări prezentate mai sus, că implementarea unei politici coerente de abordare a proiectării pentru reutilizare, la nivel organizaţional, implică în primul rând decizii şi orientări în acest sens, la nivelul conducerii acestor organizaţii.

1.11.5. Concluzii referitoare la reutilizarea în sistemele în timp real

Din analiza particularităţilor sistemelor în timp real (§ 1.9), rezultă unele concluzii importante pentru abordarea aplicării tehnicilor de reutilizare în acest domeniu. Prezentăm în continuare câteva astfel de concluzii :

- proiectarea sistemelor în timp real prezintă dificultăți datorită restricțiilor de timp și cerințelor de siguranță în funcționare ;
- o direcție de dezvoltare de perspectivă o constituie proiectarea pentru reutilizare, cu un plus de dificultate rezultat din cerințe diferite de restricții temporale, pentru sisteme diferite ;
- un obiectiv important în domeniu îl constituie automatizarea proiectării ;
- conex cu problema studiată, se poate lua în considerare și reconfigurarea dinamică a programelor ;
- cerințele de proiectare ale sistemelor în timp real conduc la necesitatea definirii de noi instrumente notaționale și alte instrumente asociate proiectării (tehnici, medii de programare, elemente de limbaj, etc.) pentru integrarea restricțiilor de timp în toate etapele ciclului de viață al programelor.

1.12. Încadrarea lucrării în perspectiva problematicii reutilizării

Acest prim capitol a realizat o trecere în revistă a domeniului reutilizării **software**, foarte vast și de aceea greu de cuprins într-o prezentare (relativ) concisă. Reamintim în continuare principalele subiecte abordate pe parcursul său, respectiv, câteva concluzii și comentarii succinte pe marginea acestora :

1. Definiții (**reutilizabilitate**, **IPR**, **clase de aplicații**, etc.) absolut necesare unei abordări unitare a domeniului - § 1.1 - 1.4 ;
2. Teoria **claselor de aplicații** și a **SGBS**, ca fundamentare teoretică a posibilităților de reutilizare software, folosită din plin în lucrarea de față - § 1.4 și 1.5 ;
3. Stabilirea de contacte cu mai multe **puncte de vedere** asupra reutilizării software (§ 1.6), în vederea încadrării mai ușoare a eforturilor proprii din punctul de vedere al contribuțiilor originale ale lucrării ;
4. **Generatoare de aplicații** ca instrumente puternice de proiectare software pentru reutilizare (împărțite în prezenta lucrare în **GA** dezvoltate în mediile academice, disponibile comercial, respectiv, o clasă specială de **GA**, folosite în instrumentație - și după cum s-a arătat în § 1.7, folosite și în sistemele de telemecanică) ;
5. Instrumente **CASE** care implementează, printre alte facilități, atât *metodologii* cât și *instrumente* pentru refolosire software (**SGBS**) - § 1.8 ;
6. Aspecte particulare ale reutilizării în **sistemele în timp real** care constituie obiectul lucrării de față (restricțiile de timp, interfața cu procesul, siguranța în funcționare, etc.) ; necesitatea dezvoltării în acest domeniu (desigur, din perspectiva reutilizării software), a unor limbaje specializate și a unor **GA** specifice - § 1.9 ;

7. **Modele** ale procesului de proiectare software pentru reutilizare (§ 1.10) care pot constitui puncte de plecare pentru dezvoltare de noi modele, pornind de la aspectele de *dinamică a procesului de proiectare, reprezentare formală şi comparare reacţie* ;

8. **Clasificarea** posibilităţilor de reutilizare software (§ 1.11.1) ; pentru perspectiva lucrării de faţă s-a considerat utilă clasificarea **Biggerstaff+** ;

9. **Factorii care influenţează reutilizabilitatea** şi strâns legate de aceştia, **avantaje şi probleme asociate** reutilizarării software, respectiv, **orientări strategice** care privesc mai ales activitatea organizaţiilor de proiectare software (§ 1.11.2-1.11.4) ; aceste aspecte trebuie avute în vedere de orice abordare a proiectării pentru reutilizare, întrucât orice tehnică, metodologie sau instrument software folosit în acest scop trebuie să aibă în vedere aspecte *practice*, de eficienţă ;

10. Formularea unor **concluzii** notabile (§ 1.11.5) referitoare la reutilizarea în sistemele în timp real : se conturează posibilităţi de dezvoltare referitoare la instrumente şi tehnici relativ la exprimarea restricăiilor de timp, la **GA** din acest domeniu, la implementarea posibilităţilor de reconfigurare dinamică, etc. ;

11. **Mediul ştiinţifico-tehnic şi suportul logistic** pentru domeniul abordat : colective de lucru, manifestări ştiinţifice internaţionale prestigioase, adrese **INTERNET**, eforturi de standardizare, lucrări de specialitate, etc. ; s-a realizat o prezentare succintă în Anexa 1.3, care nu face decât să sublinieze în ce măsură subiectul abordat este de **actualitate** şi stârneşte interesul justificat al comunităţii ştiinţifice.

Apreciem că prin sinteza prezentată, referitoare la **domeniul reutilizării software** s-au realizat :

- **fundamentarea teoretică** a tezei, care pe tot parcursul ei foloseşte drept concept de bază **clasa de aplicaţii** ;
- **situarea, încadrarea şi interfaţarea corectă** în domeniu, prin analiza unor puncte de vedere, a avantajelor, a problemelor specifice, prin încadrarea în clasificările găsite, etc. ;
- depistarea unor **domenii de investigare** :
  - formalizarea **procesului de proiectare pentru reutilizare** ;
  - construirea unor **GA** şi a unor instrumente **CASE** specifice **sistemelor în timp real** ;
  - definirea unor noi **tehnici şi structuri de program** pentru creşterea gradului de reutilizare pentru **sisteme în timp real**, mai ales pentru programe scrise în limbaj de asamblare ;
- prefigurarea a patru **direcţii de dezvoltare pentru teză** :

- abordarea teoretică de o manieră nouă a **formalizării** procesului de proiectare pentru reutilizare ;
- dezvoltarea unor **tehnici specifice** pentru sistemele **mici** de conducere sau de instrumentație de proces (pentru aplicații scrise în limbaj de asamblare) ;
- construirea unor **GA** pentru domeniul aplicațiilor mici de instrumentație (pentru care nu au fost dezvoltate multe asemenea instrumente) ;
- realizarea unor programe **CASE** care să implementeze metode și tehnici și să înglobeze instrumente specifice pentru reutilizare.

*Abordările **teoretice** ale lucrării de față pot fi împărțite în : **utilizarea logicii fuzzy** (pentru definirea unor **măsuri software** și pentru investigarea unor proprietăți structurale) și **utilizarea tehnicilor și modelelor sistemice** pentru a contura un formalism al proiectării pentru reutilizare în ipoteza folosirii conceptului de **clasă de aplicații**.*

*Realizările **practice**, ca și continuări firești ale aspectelor teoretice dezvoltate, urmăresc consecvent două "piste" : **aplicațiile mici de conducere de proces sau instrumentație - cu microprocesoare sau microcontrollere** (lucrarea propunând pentru acest gen de aplicații un set de tehnici și structuri de programe, două **GA** și un instrument **CASE**), respectiv **aplicațiile multitasking în timp real** pentru conducere de proces (pentru care lucrarea de față propune un set de măsuri software pentru evaluarea unor aspecte ale reutilizării, respectiv un instrument **CASE** care implementează acest gen de evaluare). Instrumentele **CASE** descrise în lucrare urmăresc cu consecvență implementarea cât mai multor facilități pentru creșterea gradului de reutilizare. Astfel, ambele instrumente au facilități de **baze software**, unul din instrumente are înglobate facilități de manipulare de variante în cadrul unei **clase de aplicații**, iar celălalt instrument are inclus un **GA**.*

1.13. Structura lucrării

Conform cu prezentarea din paragraful anterior, structura lucrării respectă direcțiile enunțate, adăugând încă două capitole, necesare pentru buna structurare și fundamentare a lucrării.

Teza este structurată în 9 **capitole, anexe și bibliografie**. Structura lucrării este prezentată în Fig. 1.20.

Capitolul 1 este **introductiv**. Justificarea prezenței acestui capitol este dată chiar prin paragraful precedent.

Din considerente de parcurgere completă a domeniului reutilizării și întrucât domeniul respectiv necesită o tratare distinctă, a fost considerată necesară prezența **Capitolului 2** care se

referă la **tehnicile orientate pe obiecte**. Unele aspecte specifice acestei abordări vor fi extrapolate la tehnicile şi structurile de program prezentate în Capitolul 6 (relaţia aceasta este reprezentată prin linia întreruptă de pe figură, care uneşte cele două capitole în cauză).

Capitolul 3 prezintă **măsuri software** folosite în prezent pentru evaluarea reutilizării, atât în domeniul pradigmelor clasice cât şi în domeniul tehnicilor orientate pe obiecte (cele două săgeţi dinspre primele capitole spre acest al treilea capitol sugerează această legătură ; deoarece conceptele **POO** nu constituie principala preocupare a lucrării, adoua legătură a fost realizată prin linie întreruptă). În Capitolul 3 propunem şi câteva extinderi de sens pentru măsuri software, pentru a putea fi utilizate în evaluarea gradului de reutilizare. Capitolul mai analizează şi inconvenientele măsurilor existente, pregătind astfel următorul capitol.

Capitolul 4 propune noi **grade de reutilizare** definite printr-o logică fuzzy, elimi-nând astfel inconvenientele subliniate anterior. În acest capitol este construită şi o mică teorie a **relaţiilor fuzzy** induse de aplicaţii care folosesc gradele de reutilizare definite aici.

Capitolul 5 propune **un model şi un formalism original pentru procesul proiectării pentru reutilizare**, în cadrul unei clase de aplicaţii. Aici sunt definite şi măsuri software utile în **evaluarea calităţii reutilizării**.

Capitolul 6 este o continuare firească a aspectelor teoretice propuse, prin aceea că, în formalismul propus, ocupă o anumită nişă : cea a tehnicilor numite în capitolul 5, de **speţa a II-a**.

Capitolul 7 prezintă **GA (tehnici de speţa a III-a)** pentru sisteme mici de instrumentaţie (conform clasificării (5.37) din cap. 5).

Capitolul 8 realizează descrierea a două instrumente **CASE** (conturate succint de asemenea în paragraful precedent). Deoarece aceste instrumente **CASE** implementează atât **GA** cât şi concepte de speţa a II-a, în fig. 20 au fost reprezentate legături între capitolele anterioare şi acest capitol. De asemenea, deoarece ambele instrumente folosesc conceptele teoretice dezvoltate anterior, pe schemă au fost reprezentate prin linie punctată şi aceste conexiuni.

Anexele completează conţinutul şi au fost realizate pentru a nu întrerupe firul dezvoltărilor. Ele conţin exemplificări şi prezentări *in extenso* ale unor calcule sau programe, precum şi completări la prezentare.

În text au fost folosite caractere **îngroşate** sau <u>sublinate</u> pentru a scoate astfel în relief anumite cuvinte-cheie, considerate importante, sau pentru a sublinia ideile centrale ale unor afirmaţii. Au fost de asemenea utilizate caractere *înclinate* (*italice*) pentru a evidenţia corpul unor *observaţii* sau *definiţii* precum şi unele *atribute* considerate demne de interes. În anumite paragrafe sau chiar capitole, cuvinte corespunzând unor concepte sau idei centrale, au fost constant scoase în evidenţă prin caractere *înclinate*, pentru a sublinia astfel ideea urmărită pe parcursul respectivului fragment.

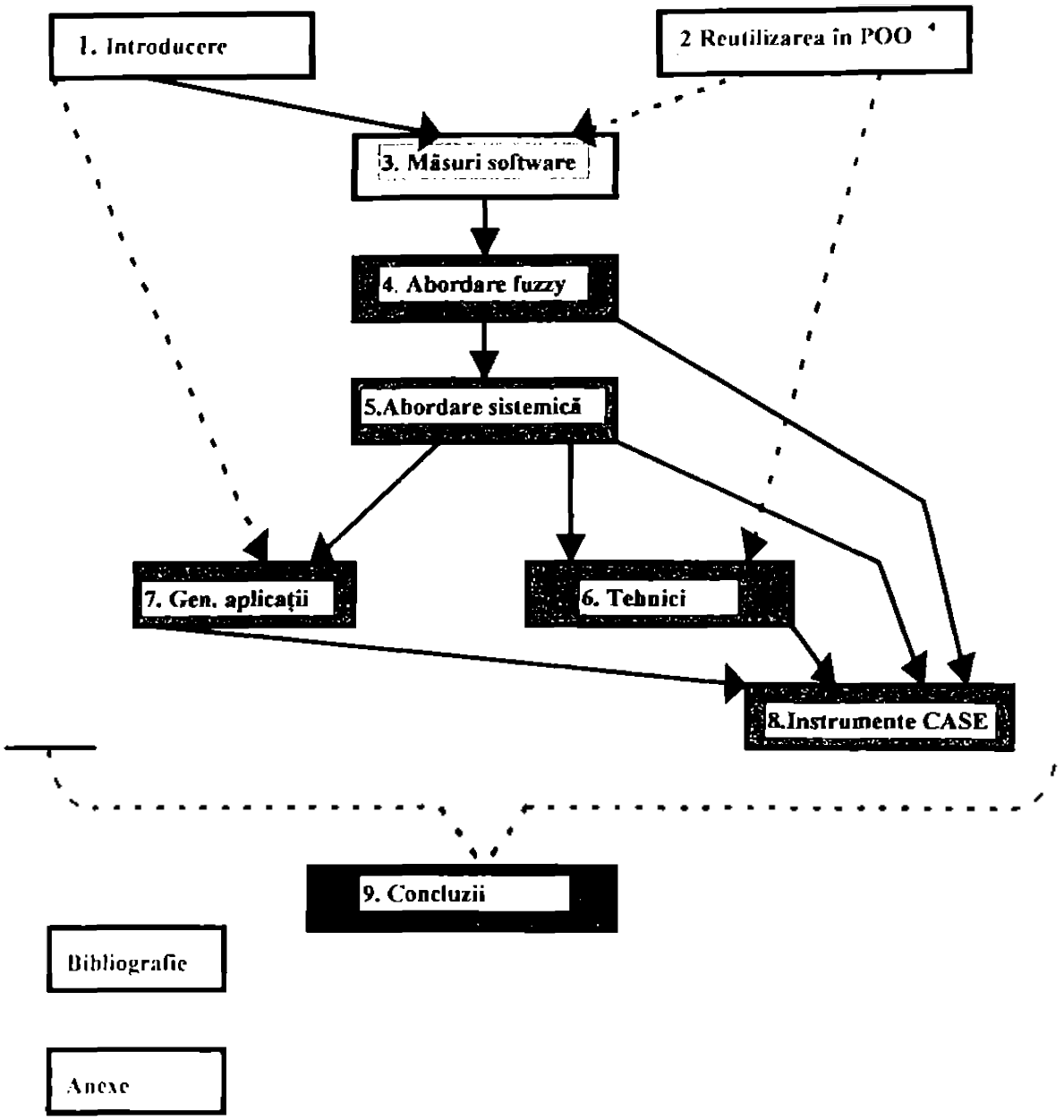


Fig. 1.20. Structura lucrării

CAPITOLUL 2
PROGRAMAREA ORIENTATĂ PE OBIECTE ŞI REUTILIZAREA

Folosirea proiectării şi programării orientate pe obiecte (**POO**) este una din realizările de succes ale ingineriei programării ultimului deceniu. Folosirea **tehnologiilor orientate pe obiecte** (**TOO**) necesită însă anumite precauţii în practica proiectării software [Nie95][Uni94]. Doar aplicarea corectă, consecventă şi pe întreaga durată a ciclului de viaţă al programelor a acestor **TOO** duce la evidenţierea efectivă a numeroaselor avantaje ale utilizării sale. O înţelegere corectă a conceptelor sale de bază poate să ajute la evitarea unor "capcane" asociate acestei tehnologii.

Extinderea **TOO** la etapele de analiză şi proiectare s-a realizat de-abia în ultimii ani şi una din problemele dificile în acest domeniu este alegerea metodei care va fi aplicate pentru realizarea proiectului.

Întrucât lucrarea de faţă se referă la *reutilizare* şi deoarece majoritatea conceptelor fundamentale ale **TOO** conţin în sine elemente de *reutilizare* (v. § 1.11.1. B - reutilizarea de speţa a III-a), este necesară trecerea în revistă a acestora. De aceea primul paragraf al acestui capitol va prezenta paradigma **TOO**. În cadrul acestei prezentări vor fi subliniate în mod special aspectele care interesează *reutilizarea*. O lucrare de referinţă pentru domeniul urmărit este [Nie95], din care sunt preluate unele formulări din cuprinsul capitolului.

Scopul capitolului de faţă este de a sublinia principalele aspecte legate de *reutilizare* în ceea ce priveşte folosirea **TOO**. De aceea, prezentarea are o factură specifică de trecere în revistă a problematicii generale a acestui domeniu, fără a se intra în prea multe detalii. Dat fiind tematica lucrării de faţă, am considerat potrivit să realizăm o baleiere completă a domeniului *reutilizării* în ingineria programării, pentru a putea încadra mai corect contribuţiile originale. Din acest motiv, prezenţa unui capitol de sinteză asupra aspectelor de *reutilizare* în **TOO** este justificată. Pe de altă parte, vom vedea că multe alte tehnici de programare şi abordări specifice *reutilizării*, chiar în cadrul paradigmei proiectării şi programării structurate fac apel (sau pot fi puse în legătură) la concepte ale **TOO** (de fapt la acele principii care sunt generale în ingineria programării, dar sunt puse în valoare la maximum în cadrul **TOO.**

În cuprinsul capitolului am ţinut să subliniem (prin folosirea caracterelor *italice*) termenul de *reutilizare* (şi înrudite) pentru a evidenţia astfel aspectele urmărite cu consecvenţă în toată lucrarea.

*Capitolul acesta își propune în primul rând să treacă în revistă elementele caracteristice ale paradigmei **TOO**, în vederea evidențierii posibilităților de exploatare a acestor caracteristici în vederea îmbunătățirii **reutilizării** software. Pe acest fundament, vom prezenta în continuare nivelele **reutilizării** în proiectarea prin **TOO**, respectiv modul cum aceasta poate fi realizată în diverse etape ale **TOO** (analiză, proiectare, implementare). Câteva considerații pe marginea utilizării **TOO** completează perspectiva generală asupra subiectului vizat. În încheiere, sunt prezentate câteva particularități ale limbajului **C++** (ca limbaj tipic - și cel mai folosit - pentru aplicarea paradigmei **TOO**), care pot contribui la creșterea posibilităților de **reutilizare** ale programelor scrise în acest limbaj, precum și câteva concluzii.*

2.1. Paradigma TOO

Abordarea orientată pe obiecte [You79] este considerată o importantă realizare în ingineria programării, după definirea elementelor caracteristice ale programării structurate. **Programarea orientată pe obiecte (POO)** a fost introdusă ca un concept inițial al **TOO**, și a fost propulsată odată cu limbaje ca **Simula** [Dah70], **Smalltalk-80** [Gol83], **C++** [Str91] și **Eiffel** [Mey88]. Proiectarea orientată pe obiecte (care folosește **POO**) a fost promovată odată cu lucrările [Abb83], [Boo86], [Lis86], etc.

Care sunt avantajele imediate ale folosirii **TOO** ? Cele mai evidente (și importante) sunt:

- la nivelul **managementului** organizațiilor software :
  - gradul ridicat de **reutilizabilitate** care poate fi atins prin folosirea acestor tehnici;
  - o mai bună comunicare cu clientul ("obiectele" pot descrie pentru acesta lumea reală mai bine);
  - timp de dezvoltare a aplicațiilor, mai scăzut (putând realiza și prototipizare rapidă) ;
  - ușurința întreținerii
- la nivelul **specialiștilor** (proiectanților, teoreticienilor) :
  - un înalt grad de modularitate, asigurându-se astfel o slabă cuplare între module ;
  - prezența unui mecanism de încapsulare care asigură respectarea principiului ascunderii informației ;
  - abstractizarea datelor pentru a realiza astfel obiecte care se referă la un anumit tip de date și operații asociate ;
  - existența mecanismelor de clasificare și moștenire care asigură **reutilizabilitatea** și

extensibilitatea.

Prin extinderea **TOO** către faza de analiză [Boo93], [Jac92], acestea acoperă practic întregul proces de dezvoltare software.

Principalele concepte şi elemente constitutive ale paradigmei **TOO** sunt :

a) Clasele şi obiectele

Un obiect este un "*lucru*" sau un concept utilizat pentru descrierea unor entităţi specifice care sunt în relaţie cu sistemul care trebuie realizat. Fiecare obiect are asociate un set de atribute şi operaţii care definesc comportarea acestuia. O clasă este un cadru ("*template*") sau prototip care descrie atributele comune şi operaţiile care se aplică asupra mulţimii obiectelor corespunzştoare care aparţin acestei clase. Un anumit obiect este considerat o **instanţă** a unei clase şi este identificat prin nume sau identificator numeric. Când este creat un obiect, acestuia i se asociază un set de atribute şi operaţii. Clasele şi obiectele sunt concepte vitale pentru **TOO**, din mai multe motive :

- **descompunere** : un sistem mare poate fi descompus într-un număr de clase abstracte. Aceste clase, precum şi obiectele asociate au relaţii directe cu entităţile lumii reale şi pot fi utilizate pentru a stăpâni complexitatea sistemului ;
- **comunicare şi documentare** : deoarece clasele abstracte şi obiectele sunt în relaţie directă cu entităţile lumii reale, pot fi folosite în discuţiile cu clientul sau utilizatorul final pentru a creşte gradul de înţelegere asupra cerinţelor sistemului ;
- **reutilizabilitatea** : strategia de clasificare implică o ierarhie şi o potenţială moştenire în cadrul relaţiilor superclasă/subclasă. Subclasele pot moşteni atribute şi operaţii de la superclase şi eventual pot căpăta atribute şi operaţii adăugate la subclase, deci de fapt realizăm *reutilizare* în cel mai înalt grad, folosind **TOO** ;
- **întreţinere uşoară** : potenţialul crescut de *reutilizabilitate* implică extensibilitate, uşurinţa reconfigurării pentru sisteme viitoare şi efort redus de întreţinere.

b) Încapsularea şi ascunderea informaţiei

Caracteristica generală a **TOO** este folosirea extensivă a abstractizării. O abstractizare este o descriere de nivel înalt (sau un model) - care ignoră detaliile considerate nesemnificative pentru perceperea unitară - a unui concept detaliat sau complex. Bineînţeles, abstractizarea nu se foloseşte numai în **TOO** ci şi în programarea structurată. La aceasta din urmă, se foloseşte abstractizarea procedurală (funcţională), respectiv, implementarea unor detalii este înlocuită cu apeluri de proceduri. La un nivel mai înalt de abstractizare, se includ procesele sau taskurile şi bibliotecile de legături dinamice (**DLL**) care conţin implementări **GUI** (Graphical User Interface - interfaţă grafică cu utilizatorul).

O abstractizare care reprezintă o entitate software poate fi descrisă printr-o notație unitară, prin **încapsulare**. Aceasta trebuie să fie însoțită de mecanisme care să constituie singura cale de acces la informațiile dorite, pentru abstractizarea dată. Un obiect bine proiectat încapsulează o entitate bine precizată a lumii reale care are un set de atribute și operații. O încapsulare corectă a unui modul software are câteva caracteristici importante :

- detaliile de implementare ale structurii de date și ale algoritmilor folosiți pentru operații, sunt ascunse pentru utilizator. Aceasta asigură o slabă cuplare între diversele module ale aplicației care utilizează obiectul, iar schimbările în implementarea obiectului pot fi realizate fără modificări care să afecteze întregul sistem ;
- partea vizibilă a modulului asigură o interfață unică și neambiguă cu obiectul încapsulat.

Această interfață este singura cale de a accesa structurile de date reprezentând abstractizarea. Aceasta asigură un **grad ridicat de reutilizare** deoarece obiectele încapsulate pot fi utilizate în sisteme diferite fără a schimba interfețele.

Am văzut că încapsularea implică ascunderea informației, de aceea câteodată aceste două concepte sunt folosite ca sinonime. Cu un limbaj de programare precum C++, care asigură un suport direct pentru ascunderea informației, o mare parte a efortului de proiectare va fi direcționată spre găsirea și asigurarea unui nivel potrivit de ascundere a informației.

c) Abstractizarea datelor

Se referă la modul de implementare a încapsulării și ascunderii informației la nivelul proiectării și programării. Această poate fi descrisă ca și procesul creării unor tipuri definite de utilizatori, care să implementeze clasele și obiectele lumii reale care să creioneze proiectarea sistemului. Aceste noi tipuri reprezintă o extensie a tipurilor
"built-in", care sunt în mod normal disponibile într-un limbaj de programare dat.

Conceptul de abtractizare a datelor este bazat pe o combinare a abstractizării procedurale și abstractizare prin specificare [Lis86]. O structură de date, declarată ca tip, și operația necesară de manipulare a obiectelor tipului respectiv, sunt specificate împreună și adunate într-un modul de program. Operațiile devin o parte a unui tip nou. Modulul reprezintă o încapsulare a tipului și operațiilor, iar operațiile sunt considerate interfețe pentru afectarea evoluției (comportării) obiectelor tipului. O caracteristică a unui **tip abstract de date (TAD)** este că poate încapsula doar o singură abstractizare. Dacă nu se respectă acest principiu, crește cuplarea modulelor și corespunzător se complică modificările de realizat în cazul operării unor schimbări undeva în program. Suportul oferit de limbaj pentru crearea **TAD** este de cea mai mare importanță în Proiectarea Orientată pe Obiecte. Posibilitatea de implementare directă poate fi un factor semnificativ în crearea unor sisteme software mari care să prezinte caracteristicile dorite ale unei coeziuni puternice și ale unui cuplaj slab între module.

d) Responsabilităţile

Putem vedea o clasă C++ ca o specificare de tip client/server. Interfaţa vizibilă include un număr de operaţii care reprezintă un server, spre un număr de clienţi care utilizează operaţiile asociate instanţelor specifice (obiecte) ale clasei. Unele dintre aceste operaţii pot fi ascunse de clienţi şi pot fi utilizate doar de obiectele în sine. Operaţiile asociate (funcţii membre în C++) obiectelor determină comportarea acestui obiect şi sunt referite ca responsabilităţi ale obiectului.

Un punct de vedere uşor diferit relativ la mecanismul client/server este prezentat în [Boo93] unde interfaţă vizibilă a clasei asigură un **contract** cu clienţii. Acesta conţine responsabilităţile obiectului, adică elementele referitoare la comportarea pentru care este responsabil.

e) Colaborările şi transmiterile (pasările) de mesaje

O colaborare poate fi definită ca un serviciu asigurat de un obiect pentru a fi utilizat de alte obiecte. Acestea, în comun, definesc comportarea aşteptată la nivelul sistemului. Colaborările sunt strâns legate de conceptul de **transmitere (pasare)** a mesajelor, care este în fapt descrierea modului în care modulele software comunică. Transmiterea "mesajelor" de la un obiect la altul este implementată uzual prin combinarea trecerii unor parametri şi returnarea valorii rezultatelor. Spre exemplu, în C++ parametrii sunt pasaţi prin valoare sau referinţă pentru apeluri de funcţii, iar rezultatele sunt returnate chiar prin valorile de retur sau prin modificarea variabilelor referenţiate.

Pasarea mesajelor în C++ este de fapt doar realizarea apelurilor de funcţii asupra funcţiilor membre vizibile, declarate astfel în specificarea clasei. Direcţia pasării mesajelor poate fi indicată printr-un graf, ca în fig. 2.1. Datele pasate ca parametri pot fi schimbate în ambele direcţii (săgeţile cu cercuri goale). Modul de retur al unei date este dependent de limbajul de implementare. Un cerc umplut poate simboliza pasarea unui indicator (*flag*).

Pentru sistemele în timp real, este necesară luarea în considerare a colaborării şi pasării mesajelor într-un mediu concurent

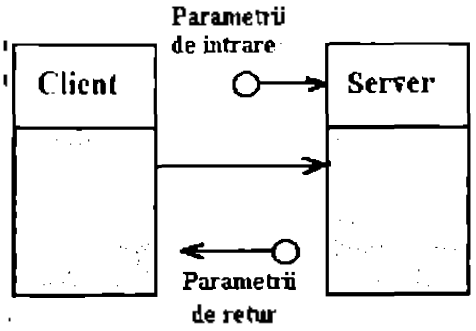


Fig.2.1. Pasarea mesajelor

f) Moştenirea

Conceptul de moştenire este utilizat pentru descrierea creării unei subclase pornind de la una sau mai multe superclase. Fiecare subclasă moşteneşte structura şi comportarea superclasei sale, incluzând atributele şi operaţiile specificate. Dacă se permite ca o subclasă să moştenească de la mai mult de o superclasă, limbajul de programare folosit suportă moştenirea multiplă.

Relaţia de moştenire este exemplificată în reprezentarea grafică din fig. 2.2, unde

săgeata este orientată spre antecesor. Superclasa este **Fereastra** şi subclasele de la primul nivel sunt **FerPrinc, Control** şi **Dialog**. Subclasa **Control** are subclase de nivel scăzut : **Editare, CutiaDeListare** şi **BaraDeDefilare**.

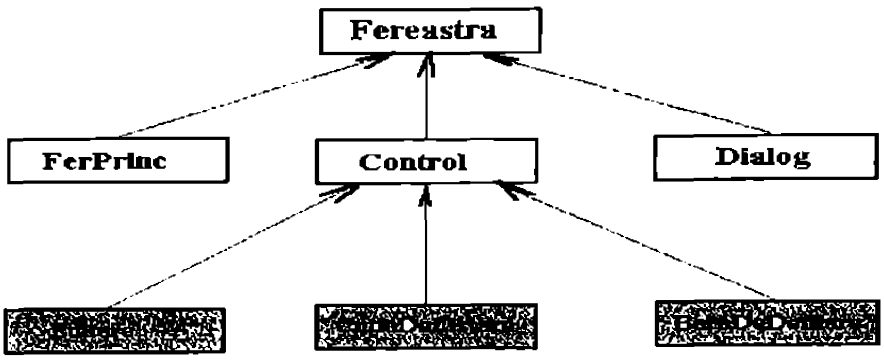


Fig. 2.2. Exemplu de relaţii de moştenire

Atributele (caracteristicile) şi operaţiile superclaselor sunt moştenite de fiecare subclasă ca un subset. Operaţiile adiţionale care sunt unice pentru o subclasă dată pot fi adăugate ca extensii. Săgeţile din figură care arată dinspre osubclasă spre o superclasă, implementează expresia "**este un/o**", spre exemplu, pentru fig. 2.2, **Control** este o **Fereastra**. O altă faţetă a moştenirii, prezentată în [Boo93] este descrisă de expresia corespondentă săgeţii "**este un fel de**". Spre exemplu, **Dialog** este un fel de **Fereastra**.

Trăsătura de moştenire permite o redefinire a claselor (copil) derivate din clase părinte. Această asigură un instrument pentru **organizarea şi construcţia claselor reutilizabile**, pe baza modulelor existente. Dacă o operaţie din clasele de pe nivelul ierarhic superior este modificată, schimbările sunt automat moştenite de clasele de pe nivelul inferior (după recompilare). Fără trăsătura de moştenire, fiecare clasă trebuie dezvoltată ca o entitate independentă, într-un stil **bottom-up** (de jos în sus). Consistenţa cerută poate fi astfel asigurată doar din disciplina de programare, nu din tipurile şi interfeţele verificate de compilatorul care suportă moştenirea.

Efectul net, imediat, al moştenirii constă în reducerea codului care trebuie realizat pentru a efectua o eventuală *reutilizare*. O problemă însă rămâne codul, mai complicat, în absenţa unui browser disponibil pentru navigarea prin diversele nivele ierarhice de clase.

g) Polimorfismul

Această trăsătură se referă la acea caracteristică a unui obiect de a exista ca instanţă a diferite clase, apartenenţa stabilindu-se în momentul rulării. Într-un limbaj de programarecare suportă polimorfismul, operaţiile au tipuri statice şi dinamice asociate, iar referinţele tipurilor dinamice pot să se schimbe în timpul execuţiei. Un exemplu simplu este prezentat în fig. 2.3, în

care **Poligon** este superclasa, iar **Triunghi**, **Dreptunghi** şi **Paralelogram** sunt subclasele. Polimorfismul poate fi aici ilustrat prin calculul ariei unui obiect, care poate fi o instanţă a oricărei subclase din figură.

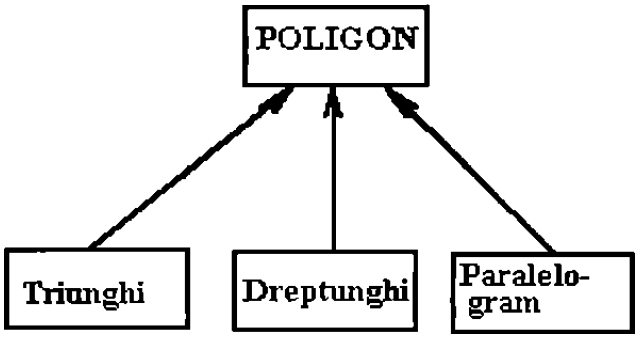


Fig 2.3. Exemplu de moştenire care ilustrează conceptul de polimorfism

h) Legătura

Conceptul, utilizat în interconexiune cu un limbaj de programare, constă în legarea unui mesaj (adică, apelarea unei operaţii asociate unui obiect) de codul care va fi executat ca urmare a apariţiei mesajului. Într-un limbaj de programare cu legare statica toate referinţele sunt determinate la compilare. Legarea dinamică, sau întârziată, înseamnă că selectarea execuţiei codului ca răspuns la un mesaj nu poate fi realizată înainte de execuţie. Adesea, astfel se implementează meniurile "pull-down" unde acţionarea unor "butoane" este asociată cu mesaje care activează funcţii corespunzătoare.

Moştenirea, polimorfismul şi legătura sunt interdependente, iar executarea unui apel la o operaţie este asociată unei referinţe polimorfe, care depinde de tipul referinţei. În timpul execuţării unui program scris în limbajul C++, pentru exemplul din fig. 2.3, referinţa potrivită la o funcţie membră **Aria** va fi determinată la compilare sau în timpul execuţiei. Utilizarea cuvântului-cheie **virtual** anunţă compilatorul că referinţa este rezolvată în momentul execuţiei.

i) Modularitatea

Se referă la modul în care "împachetăm" sistemul în entităţi software manevrabile pentru a reduce complexitatea de sistem mare. Aceste entităţi pot fi **TAD**, operaţii individuale, biblioteci întregi şi programe necesare creării unei imagini executabile. Entităţile software pe care le-am delimitat astfel se numesc **module**, şi crearea acestor module are un efect profund asupra modului cum putem profita de potenţialele beneficii ale **TOO**, mai ales în ceea ce priveşte *reutilizarea*.

O importantă raţiune în crearea modulelor este aceea de a fi unităţi de compilare. Aceasta poate reduce puternic volumul de recompilat pentru sisteme mari, după operarea unor modificări, adică are un efect imediat asupra creşterii productivităţii *reutilizării*. Conceptul de modul implică stabilirea unor limite identificabile pentru module. De aceea, vom încerca obţinerea unor module care să implementeze încapsularea şi ascunderea informaţiei. Principiul general de proiectare pentru sisteme modulare nu a fost schimbat prin introducerea TOO : coeziune ridicată şi cuplare slabă între module.

O deosebire fundamentală faţă de modularizare în programarea structurată este aceea că **TOO** se bazează mai ales pe **TAD**. Modulele proiectate structurat includ un set de operaţii şi

structuri de date asociate, dar acestea din urmă nu pot fi ascunse şi nici modificate, rezultând o puternică cuplare cu această structură.

Un modul C++ specific constă dintr-un fişier header (cu extensia .h) care conţine specificările unei clase C++ ca şi **TAD**. Acest fişier header este importat (cu directiva #include a preprocesorului) şi reprezintă specificarea ("contractul") pentru programul client care foloseşte acest **TAD**. Un fişier separat conţine implementarea **TAD**. Un alt fişier - fişierul program - conţine construcţiile de programe care implementează trăsăturile specifice sistemului. Un exemplu simplu (referitor la implementarea unui mecanism de tip stivă) prezintă în continuare aceste aspecte ale modularităţii :

```
// fisier header (stack.h)

class stack { // specificare
public:
     void stackInit();
     long topOfStack();
     long pop();
     void push(long);
private:
     long stackItems[50];
     int stackPtr;
};

// fisier sursa (stack.cpp)
#include stack.h

// implementare
     void Stack::stackInit() { ... };
     long Stack::topOfStack() { ... };
     long Stack::pop() { ... };
     void Stack::push(long) { ... };
};

// fisier program (usestack.cpp)
#include stack.h

// aici, declaratiile globale

void main() {
```

```
    newStack Stack;
    ...
    push( ... );
    ...
    pop();
    ...
);
```

Posibilitatea de modularizare este foarte importantă pentru atingerea unui grad ridicat de *reutilizare*, întrucât pe de o parte sistematizează eforturile de combinare a diferite module existente, iar pe de altă parte permite o mai uăoară construire a unor browsere de module, deci regăsirea componentelor.

j) Genericitatea

Se referă la crearea unor entităţi software cu tipuri parametrizate care pot fi instanţiate cu diferite tipuri, la momentul compilării. Acestea se numesc clase **parametrizate**, sau clase **generice** (a se vedea în acest sens şi Anexa 1.1, 7 şi lucrarea [Gog84]). Clasa generică este creată ca un **cadru** (**template**) cu parametri formali, şi instanţele clasei sunt create prin furnizarea parametrilor actuali necesari. Acest concept oferă **ultimul nivel de reutilizare**, acela în care aceeaşi clasă (care, după cum am văzut, implementează în sine un mare potenţial de *reutilizare*) poate fi utilizată pentru diverse tipuri fără a afecta relaţiile de moştenire.

Genericitatea este suportată în **C++** prin folosirea cadrelor. Mai jos este prezentat un exemplu simplu (referitor la aceeaşi problemă ca şi exemplul precedent) :

```
template <class TYPE>
class Stack {
public:
    void reset();
    void push(TYPE);
    TYPE pop();
    TYPE topOf();
private:
    TYPE *stptr;
    int top;
    int length;
);
```

Tipul parametrizat pentru clasa generică **stack** este **TYPE**. Vom putea declara stive de diverse tipuri :

```
Stack<int> intStack;          // stiva de intregi
Stack<float> floatStack;      // stiva de numere in virgula
                              //  mobila
Stack<complex> complexStack;  // stiva de numere complexe
Stack<chr> chrStack;          // stiva de caractere
```

Clasa generică reprezintă un nivel de abstractizare mai mare decât **TAD** şi deci suportă direct *reutilizarea*, o mulţime de instanţieri putând fi realizate prin simpla schimbare a tipului sau valorii parametrilor actuali fără rescrierea clasei pentru fiecare tip. Structura unei clase generice este mult mai simplă decât ierarhiile de clase utilizate în relaţiile de moştenire.

*În concluzie, paragraful a urmărit ca, prin trecerea în revistă a elementelor constitutive ale paradigmei **TOO** (concepte, filosofii de abordare a proiectării, etc.), să evidenţieze în ce mod utilizarea acestor elemente poate contribui la creşterea gradului de reutilizare al programelor. Studiul respectiv a arătat că toate atributele paradigmei **TOO** (utilizarea claselor, ascunderea, abstractizarea, responsabilităţile, legătura, moştenirea, colaborările, polimorfismul, genericitatea) contribuie la îmbunătăţirea posibilităţilor de reutilizare.*

2.2. Nivele ale reutilizării în proiectarea prin TOO

Formele de *reutilizare* în diversele etape ale dezvoltării proiectelor software prin **TOO** sunt prezentate în fig. 2.4.

Forma de nivelul cel mai scăzut pentru *reutilizare* este practica foarte întrebuinţată a extragerii unor linii de cod individuale ("*cut-and-paste*") din surse de programe realizate anterior şi refolosirea în proiecte noi. Mai departe, prin construcţia adecvată a claselor individuale, se pot folosi mecanismele specifice ale **POO** pentru asigurarea *reutilizării*, la un nivel mai avansat. Aşa cum s-a arătat în paragraful precedent, pentru extinderea *reutilizării* claselor sunt extrem de potrivite mecanismele disponibile pentru implementarea moştenirii (crearea subclaselor şi superclaselor) şi genericităţii (utilizarea cadrelor pentru specificarea tipului unui obiect particular).

O formă şi mai evoluată de *reutilizare* este crearea unor **frameworks** (ierarhii de clase, v. § 1.11.3). Fiecare dintre acestea constă dintr-un grup de clase independente de domeniu (spre exemplu, interfeţe cu utilizatorul specifice lucrului cu ferestre sub **Windows**: **Microsoft Foundation Classes - MFC** - sau **ObjectWindows Library** al firmei **Borland - OWL 2.0**) sau specifice unui domeniu. Cele specifice unor domenii sunt dezvoltate pentru un segment mai restrâns de aplicaţii (spre exemplu pentru clase de aplicaţii în sensul din [Mit87]) : reţele,

control de trafic, instrumentaţie şi supraveghere de proces, aplicaţii bancare, etc.

Reutilizarea nu este transpusă în viaţă prin simplul fapt al utilizării unui limbaj care suportă mecanismele **POO**, ci doar printr-un efort specific, distribuit pe întreg ciclul de viaţă al

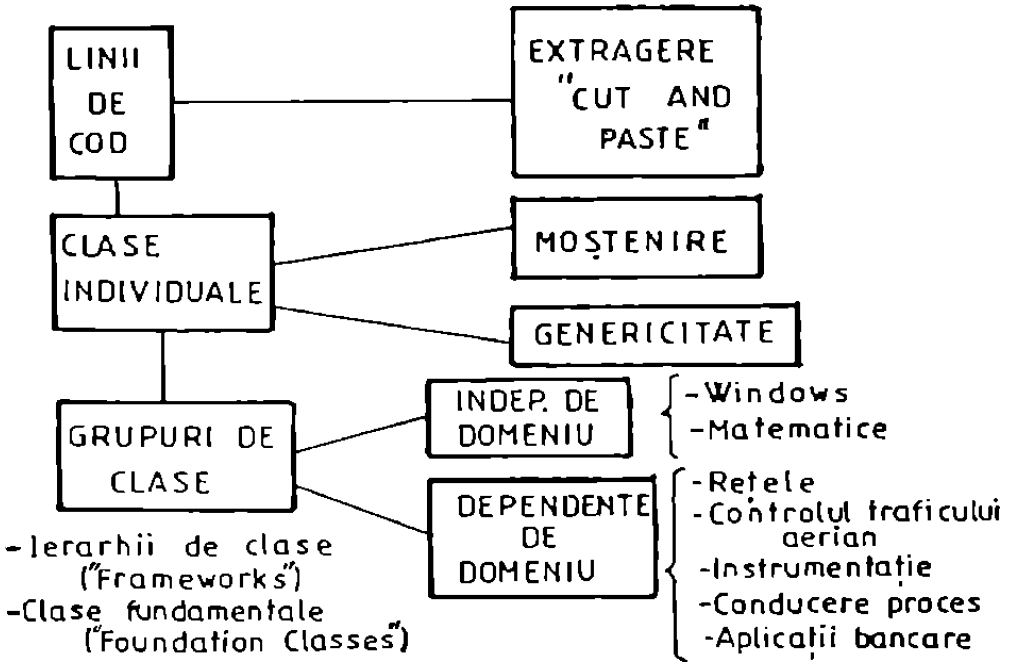


Fig. 2.4. Nivele ale reutilizării în **TOO**

programelor. În cadrul acestui efort trebuie neapărat să fie incluse şi aspecte manageriale ale activităţii, precum decizii de achiziţionare a unor biblioteci sau **frameworks** existente în alternativă cu dezvoltare de cod nou. De asemenea, activitatea de proiectare trebuie să fie ghidată de linii directoare pentru evaluarea codului existent şi al celui care trebuie realizat şi pentru evaluarea volumului şi dificultăţii proiectării prin folosire de cod existent sau cod nou.

*Deci, reutilizarea la nivel de linii-sursă, moştenirea şi generecitatea, respectiv utilizarea ierarhiilor de clase (universale sau specializate) - **frameworks** - constituie nivele succesive, din ce în ce mai rafinate, de aplicare a conceptului de **reutilizare**, în ingineria programării prin **TOO**.*

2.3. Reutilizarea în diverse etape ale TOO

Scopul acestui paragraf este de a schiţa unele aspecte ale *reutilizării*, specifice etapelor distincte ale ciclului de viaţă al programelor, specifice **TOO**. Vom trata deci pe rând fiecare dintre aceste etape, subliniind acele aspecte care sunt conexe sau privesc direct *reutilizarea* software.

a) Analiza orientată pe obiecte

Nu vom prezenta aici conținutul unor metodologii de proiectare, ci doar câteva observații specifice domeniului abordat :

- *Reutilizarea* în sens lărgit (deci mai mult decât sensul comun al *reutilizării* unor linii individuale de cod, altfel spus, mai sus decât *nivelul elementar, în schema din fig. 2.4) poate fi realizată prin transformarea entităților lumii reale în module software sau chiar, dacă este posibil, în subsisteme întregi. Altfel spus, reutilizarea în sens lărgit se referă la reutilizarea nu numai a rezultatelor etapei de programare* ci și a celor ale etapelor de analiză și preoiectare. Paradigma **TOO** (încapsularea, ascunderea, clase, obiecte, moștenirea, etc.) pot fi portate de la analiză orientată pe obiecte la proiectare orientată pe obiecte și apoi la **POO**.

- Crearea claselor și bibliotecilor suportă *reutilizare* și poate îmbunătăți creșterea calității și a productivității, deoarece trebuie testat și depanat mai puțin cod, iar cel deja depanat este refolosit.

- Întreținerea este simplificată prin înlocuirea unor obiecte întregi atunci când se schimbă cerințele sistemului.

- Dezvoltarea obiectelor poate să fie făcută prin tehnica prototipizării, pentru a avea reacția utilizatorului relativ la cerințele sistemului.

b) Proiectarea software

Tranziția între etapa de analiză și cea de proiectare este fuzzy [Nie95]. Pentru sisteme mari în timp real, tranziția de la analiza orientată pe obiecte la proiectarea orientată pe obiecte include determinarea unei mulțimi de elemente concurente care vor reprezenta un model al unei arhitecturi de proces pentru execuție paralelă. Pentru a obține un software robust și **reutilizabil**, în program trebuie încorporate elemente de toleranță la defectare. Spre exemplu, acestea pot fi implementate mai degrabă printr-o strategie consistentă de tratare a excepțiilor, decât printr-o simplă tratare a erorilor.

În implementarea unor structuri software **reutilizabile** în sistemele în timp real, de cea mai mare importanță sunt mecanisme eficiente de **comunicații interprocese (CIP)**. Pentru a crește posibilitatea *reutilizării*, trebuie acordată în egală măsură atenție comunicațiilor interprocese ca și schimburilor de date interprocesoare. Pentru a asigura transparența mecanismelor de **CIP**, cel mai adesea se folosește un model de proiectare care presupune **stratificarea** modulelor software și utilizarea protocoalelor standard de comunicație. Stratificarea asigură ascunderea informației în așa fel încât înlocuirea unui strat nu afectează decât stratul următor. Obiectivul implementărilor **CIP** este de a izola software-ul de aplicație de detaliile mecanismelor de comunicație. Aceasta asigură **portabilitatea** aplicațiilor și permite

modificări ale implementărilor mecanismelor **CIP**, fără ca proiectarea majorității modulelor de aplicație să fie afectate. Cea mai promițătoare abordare pentru atingerea acestui obiectiv o constituie crearea unor straturi pentru servicii software. Câteva din abordările **CIP** cu nivele descrescânde de cuplare între modulele de aplicații și straturile de **CIP**, sunt următoarele :

- apelurile directe ale unor primitive de comunicație de nivel scăzut ;
- pasarea de mesaje ;
- utilizarea mecanismelor de apel de la distanță [Nie92].

Câteva considerații interesante pentru subiectul nostru se referă la interfețele claselor și la moștenire, ca atribute de bază a *reutilizării* acestora. Interfața unei clase reflectă modul în care un client vede clasa, respectiv poate utiliza serviciile oferite de aceasta. Interfața include și reflectă operațiile disponibile și nivelul de ascundere a informației. Criteriile de interfațare a claselor sunt:

- **ascunderea informației** . operațiile disponibile clienților trebuie să fie vizibile (în C++ vor fi implementate ca *public*) iar elementele de date trebuie să fie ascunse (implementate în C++ ca *private*) ;
- **evaluarea operațiilor** în termenii următoarelor categorii :
  - funcții de administrare (inițializări, asignări, conversii de tip, managementul memoriei, constructorii și destructorii) care de obicei sunt vizibile pentru utilizator ;
  - funcții implementoare : manipulează obiectele și reprezintă o porțiune din interfața "**contract**" pentru clasă și deci sunt vizibile ;
  - funcții de sprijin : au atribuții auxiliare, cerute de alte funcții și din acest motiv sunt de obicei ascunse ;
  - funcții de acces : asigură accesul la datele membre, ascunse, prin furnizare de informa-ții despre acestea și diferă de funcțiile implementor prin aceea că nu modifică datele , constituie, împreună cu funcțiile implementor, interfața "*contract*" cu clienții clasei.

Moștenirea este un element esențial al *reutilizării*, dar în utilizarea acesteia, trebuie ținut cont de unele precauții : structura de moștenire nu trebuie să fie prea adâncă, de asemenea nu se va folosi o singură clasă rădăcină din care să fie derivate toate clasele. Relația **este un/o** trebuie să fie satisfăcută pentru toate subclasele. Clasele părinte trebuie să fie stabile, cu seturi bine definite de operații care pot fi **reutilizate** sau lăsate nefolosite.

Se recomandă utilizarea a maxim trei nivele icrarhice, cel puțin pentru primele câteva proiecte care folosesc **TOO**. Dacă se folosește moștenirea multiplă, proiectanții trebuie să

justifice riguros opțiunea. Utilizarea acesteia trebuie evitată la primele proiecte. În lucrarea [Nie95] este citat după [Rac94] un mers "natural" al evoluției implementării **TOO** din punct de vedere organizațional :

- primul proiect va fi folosit pentru a înțelege o parte restrânsă a paradigmei **POO** utilizând un experiment de prototipizare ;
- al doilea proiect presupune aprofundarea restului paradigmei **POO** ;
- al treilea proiect va include tehnicile cele mai subtile pentru dezvoltare de programe în stil mare și în consecință va "copleși" pe proiectanți ;

De aceea se recomandă ca de abia la al patrulea proiect să se folosească moștenirea multiplă, întrucât introduce un cuplaj mai strâns între structurile de clase derivate și deci efortul de reproiectare (implicit, de *reutilizare*) și de testare va fi mai mare, în cazul oricăror schimbări în cadrul unor clase părinte.

c) Implementarea

În procesul de dezvoltare a aplicațiilor software, această fază este ultima. Arhitectura aplicației este obținută și clarificată în etapa precedentă. Aici se cuvin subliniate unele dificultăți specifice testării programelor scrise în limbaje orientate pe obiecte (spre exemplu **C++**). Se știe că testarea în general se realizează folosind facilitățile de depanare ale mediilor de programare. Utilizarea unui depanator nu reprezintă nimic nou pentru un programator care trece la **POO**. Dar depanarea unui program scris spre exemplu în **C++** reprezintă o provocare specială deoarece multe facilități puse la dispoziție de limbaj nu pot fi decelate decât în timpul rulării. Astfel, funcțiile pot fi supraîncărcate (deci depanatorul trebuie să determine care obiect este referențiat prin mecanismul polimorfismului), obiectele sunt create de constructori din clase din moșteniri și ierarhii de agregare complexe, iar **TAD** sunt asociate cu valori temporare și obiecte, în mod dinamic. Toate acestea complică depanarea și determină strategii de testare specifice. Implementarea cu succes, implicit **refolosirea** programelor, a claselor și ierarhiilor, este strâns legată de existența unui bun depanator, care trebuie să ajute la descurcarea subtilităților inerente la un program orientat pe obiecte. Multe medii actuale dispun de bune facilități de depanare în acest sens : **Microsoft Visual C++**, **Borland C++**, **SunPro WorkShop**, etc.

*În cele de mai sus, au fost trecute în revistă câteva particularități ale **reutilizării** folosind **TOO**, dar, spre deosebire de paragraful precedent, dintr-un punct de vedere deosebit, respectiv, referitor la modalități de realizare ale **reutilizării** în diversele etape ale ciclului de viață al programelor (analiză, proiectare, implementare).*

2.4. Consideraţii despre utilizarea TOO

Utilizarea **TOO** nu garantează o realizare automată a beneficiilor aşteptate în ceea ce priveşte *reutilizarea*, implicit în ceea ce priveşte calitatea şi productivitatea. Pentru a beneficia de acestea, este necesar un efort destul de susţinut, atât în ceea ce priveşte managementul activităţii de proiectare, cât şi în ceea ce priveşte proiectarea propriu-zisă. Ba chiar apar unele **probleme** specifice utilizării **TOO**, care fac mai dificilă implementarea unei metodologii unitare şi coerente de proiectare :

- instrumente software de tip **CASE** incomplete ("imature"), care nu suportă întreaga paradigmă şi toate etapele caracteristice **TOO** ;
- ierarhii de clase, extrem de complexe : o utilizare nedisciplinată a ierarhiilor de clase poate face proiectul greu de înţeles şi relaţiile de moştenire mai greu de urmărit, iar o schimbare în clasa părinte atrage după sine retestarea în clasele descendente ;
- accent pe abstractizarea datelor : pentru **sistemele în timp real**, metodele de proiectare orientate pe obiecte pot genera câteodată unele dificultăţi în determinarea elementelor concurente, adică a abstracţiilor procesului, întrucât aceste metode se bazează în întregime pe **TAD** [Nie95] ;
- cerinţele specifice de pregătire a personalului : în [Nie95] se apreciază că implementarea la nivel organizaţional a unei metodologii specifice **TOO** şi instruirea pentru utilizarea unui limbaj adecvat **POO** durează de la 6 la 12 luni ;
- lipsa unor instrumente software adecvate : nu numai că instrumentele **CASE** disponibile sunt incomplete, dar unele aspecte ale **TOO** nici nu sunt acoperite cu instrumente adecvate, spre exemplu pentru implementarea eficientă a *reutilizării* claselor, sunt necesare browsere sofisticate care să localizeze rapid ierarhia sau calea unei entităţi conţinute într-o subclasă de nivel scăzut (a se vedea în acest sens şi observaţiile de la § 1.11.1) ;
- lipsa unui suport pentru programare concurentă : cele mai multe instrumente şi limbaje-suport nu oferă şi cadrul adecvat pentru implementarea concurenţei, de aceea pentru sistemele în timp real apar dificultăţi în realizare, trebuind făcute nu o dată "improvizaţii" care scad calitatea programelor obţinute şi în ultimă instanţă scad şi posibilităţile de *reutilizare* ;
- metodologii diferite de proiectare : dacă în cadrul unei organizaţii, diferite echipe folosesc metode diverse de proiectare, posibilităţile de *reutilizare* scad.

Există şi **factori de risc** ai utilizării **TOO** :

- timp de dezvoltare mai mare : primele aplicaţii dezvoltate cu aceste tehnici pot să coste mai mult şi durata realizării proiectului să fie mai mare decât prin metode tradiţionale (v. şi

observațiile de la § 1.11.1, perfect valabile și aici) ;

- întreținere mai dificilă : utilizarea moștenirii și a polimorfismului în C++ crește complexitatea programelor și face mai dificilă testarea, depanarea, modificarea[1] ;
- înrăutățirea unor performanțe : timpii de compilare mai ridicați datorită complexității mai mari a programelor precum și utilizarea legării dinamice, care determină o creștere a timpului de execuție, ambele sunt creșteri defavorizante ;
- recuperarea mai dificilă a investiției : din pricina costurilor ridicate ale pregătirii personalului, a instrumentelor software necesare, a dezvoltării primelor produse, recuperarea acestor cheltuieli se întinde pe o perioadă mai mare de timp (v. § 1.11.1) ba chiar aplicarea **TOO** ar putea duce la creșterea generală a costurilor dacă implementarea acestor tehnici se realizează cu jumătăți de măsură ;
- disensiuni la nivel organizațional : într-o firmă mare care se ocupă cu dezvoltare de produse software, pot să apară conflicte de promovare a unor metodologii sau instrumente software asociate, diferite, după preferințele sau interesele diferitelor colectivități componente.

În ciuda problemelor și riscurilor, utilizarea **TOO** poate îmbunătăți foarte mult activitatea de proiectare. Iată ce argumente se pot aduce (câteodată circumstanțial !) de aceea, în sprijinul celor care doresc să adopte ca strategie de firmă, trecerea de la proiectarea și programarea structurată la **TOO** și **POO** :

- solicitarea expresă a clienților (deși, în acest caz, este de dorit o clarificare a ceea ce înțelege clientul prin **POO** !);
- trecerea la un limbaj cu mecanisme ale **POO** ;
- definirea unei strategii proprii de *reutilizare* care să ducă spre exemplu și la crearea de biblioteci de clase dependente de domenii bine stabilite de aplicații ;
- **POO** oferă suport adecvat pentru **prototipizare** (versus utilizarea diagramelor de flux de date - **DFD** - care tind să orienteze reprezentările folosite spre proiectare iar nu spre client, deci îngreunează dialogul cu acesta) ;
- disponibilitatea unor scule **CASE** adecvate, care mărese foarte mult productivitatea proiectării și **reutilizabilitatea** aplicațiilor ;
- ușurința identificării subsistemelor ca reprezentări abstracte ale lumii reale.

Avantajele folosirii **TOO** precumpănesc asupra problemelor și riscurilor. De aceea, aplicarea acestor tehnologii este o orientare importantă în activitatea de proiectare software actuală, pe plan mondial.

---

[1] Este adevărat că acest argument este considerat subiectiv de către adepții **POO**, în sensul că acest "dezavantaj" ar fi de fapt un avantaj., prin însăși filosofia **POO** care prin moștenire încurajează preluarea de componente deja testate (deci de fapt viteza de testare se îmbunătățește astfel). Totuși, mulți programatori continuă să considere că depanarea / testarea unui program orientat pe obiecte este mai dificilă.

*Paragraful evidenţiază problemele specifice, induse de utilizarea **TOO**, factorii de risc aferenţi, dar şi argumentele care pledează pentru utilizarea acestora.*

2.5. Aspecte ale utilizării limbajului C ++

Scopul acestui paragraf este de a prezenta câteva din argumentele pentru care **C++** este foarte mult folosit în proiectarea programelor, chiar şi pentru sistemele în timp real. Limbajul C++ suportă paradigma **POO** prezentată în § 2.1. Suportă de asemenea şi alte facilităţi, pe care le vom enumera mai jos, mai puţin concurenţa. Dar există extensii specifice pentru a o implementa şi pe aceasta.

C++ este un superset al limbajului **C** şi constituie o progresie naturală de la un limbaj procedural la un limbaj orientat pe obiecte. De aceea, trecerea la C++, în cadrul unei politici organizaţionale de trecere pe **TOO**, este cea mai bună opţiune, mai bună decât spre exemplu trecerea de la **C** la **Smalltalk**. Programatorii **C** trebuie să se concentreze, în trecerea lor la noul limbaj, doar asupra diferenţelor şi a noilor facilităţi.

C++ oferă mecanisme puternice de verificare a utilizării corecte a funcţiilor (tipuri şi număr de parametri). Fişiere separate de declaraţii (fişiere header) pentru definirile interfeţelor şi a fişierelor de implementare (fişiere sursă) permit o utilizare flexibilă, în sensul *reutilizării*. Compilatorul face verificările necesare, de consistenţă a apelurilor de funcţii între fişierele corespunzătoare aplicaţiei şi fişierele de implementare.

Construcţia claselor şi supraîncărcarea operatorilor din **C++** permit construcţia unor tipuri de date concrete. Acestea sunt definite de utilizator şi pot fi utilizate la fel ca şi tipurile existente (ca şi *int* sau *float*). Dintr-o clasă pot fi create obiecte (instanţe) asupra cărora pot fi efectuate operaţii ca "+" sau "||", deci rezultă o uniformizare a stilului de tratare pentru toate tipurile utilizate în program.

C++ este compatibil la nivelul editării de legături cu alte limbaje (**C**, **FORTRAN**, etc.). Utilizarea **C++** este benefică mai ales la aplicaţiile utilizând **GUI** [Pow95] pentru **Windows** sau **XWindow System**. Funcţiile de interfaţă **GUI** pot fi apelate direct din aplicaţia **C++** sau pot fi încapsulate în biblioteci de clase specifice.

Mecanismul de editare de legături care permite legarea cu module în alte limbaje asigură un suport potrivit pentru o strategie de dezvoltare adecvată trecerii de la **C** la **C++**, prin **refolosirea** modulelor existente scrise în **C** şi adăugarea codului nou (partea modificată a aplicaţiei) scris în **C++**. Codul în **C** moştenit de la aplicaţiile precedente poate fi treptat rescris în **C++**, la fiecare actualizare, astfel încât după un timp întreaga aplicaţie va fi transferată în **C++**.

Reutilizarea poate să fie realizată şi aici (v. şi § 1.11.2) în mai multe moduri. La nivelul

cel mai scăzut, se pot prelua din aplicații vechi spre exemplu doar funcții membre, prin **copiere** (*cut and paste*) și adăuga noi funcționalități acestora, în aplicațiile în curs de dezvoltare. Multe produse **GUI** reutilizează software în acest mod. La un nivel mai ridicat, se pot refolosi funcții din **biblioteci** linkeditate cu aplicația noastră sub dezvoltare (spre exemplu funcțiile **Windows API - Applications Programming Interface**). Nivelul următor de reutilizare este importarea unor tipuri individuale de **clase**. Cel mai înalt nivel de reutilizare folosind **TOO** îl constituie includerea în aplicație a unui întreg strat între aplicația propriu-zisă și mecanismul aflat sub aplicație (incluzând aici și sistemul de operare). Un astfel de strat poate fi implementat spre exemplu de **Microsoft Foundation Classes - MFC** - sau **Borland's ObjectWindows Library - OWL** [Bar94].

Reutilizarea unei clase poate fi îmbunătățită dacă în ea sunt incluse **funcții inverse** (complemetare) celor identificate ca fiind necesare. Asemenea funcții pereche sunt de exemplu *adăugare ștergere, ascundere prezentare* sau *desenare ștergere fereastră*. Astfel, interfața rezultată este mai bună pentru un potențial client al serviciilor clasei. Funcția adăugată inițial oricum urmează a fi adăugată în momentul, foarte probabil de altfel, al apariției necesității noii funcții. Dezavantajul unui volum ceva mai mare de cod și deci de efort inițial de proiectare, merită totuși de cele mai multe ori a nu fi luat în considerare.

Una din cele mai importante posibilități de *reutilizare* în limbajul C++ este oferită de mecanismul suport al **moștenirii**. O proiectare îngrijită a claselor de bază care să aibă grijă de vizibilitatea în raport cu utilizatorul și de asemenea de modul de realizare și desemnare a funcțiilor virtuale poate să asigure un mare grad de *reutilizare* a claselor și bibliotecilor de clase. Este important de conștientizat că acest tip de *reutilizare* este realizabil printr-un efort conștient de planificare. **Documentarea** pentru modulele de program scrise deja, ușurința **regăsirii** și ușurința **utilizării** sunt cele mai importante trăsături ale unei **strategii de reutilizare** de succes.

*Limbajul C++, se arată deci în acest paragraf, este potrivit pentru a fi folosit cu succes pentru implementarea **TOO**, oferind suport paradigmei respective și deci implicit posibilităților de reutilizare evidențiate în § 2.1. Prin ceea ce oferă, limbajul și suportul instrumental disponibil (medii de programare, biblioteci, clase și ierarhii de clase) sunt potrivite pentru implementarea unei strategii organizaționale eficiente pentru **reutilizare**.*

2.6. Concluzii

S-a arătat în acest capitol că **TOO** sunt indicate pentru a implementa o strategie coerentă de *reutilizare*, în cadrul unei organizații care derulează activități de proiectare software. Majoritatea atributelor specifice **TOO** vin direct în ajutorul eforturilor de reutilizare software.

În special modularitatea aferentă structurii programelor realizate sub semnul acestei paradigme, mecanismele de moştenire, dar şi încapsularea contribuie efectiv la realizarea reutilizarii. În şi mai mare măsură, genericitatea determină posibilitatea refolosirii unor module de program. Dacă au mecanisme eficiente asociate de regăsire, bibliotecile de clase (**frameworks**) oferă posibilităţi largi de dezvoltare rapidă a unor părţi mari de aplicaţie (spre exemplu interfeţe). Utilizarea **TOO** presupune însă şi anumite dificultăţi, care nu contracarează avantajele oferite, decât în anumite cazuri.

Un limbaj foarte potrivit utilizării **TOO** este C++. În § 2.5 am arătat de ce acest limbaj este perfect utilizabil pentru a putea proceda la o abordare coerentă a *reutilizării*, din punct de vedere organizaţional.

În fig. 2.5 am sintetizat şi subliniat încadrarea aspectelor de *reutilizare* software în contextul problematicii **TOO.** Figura evidenţiază conexiunile care se pot realiza între

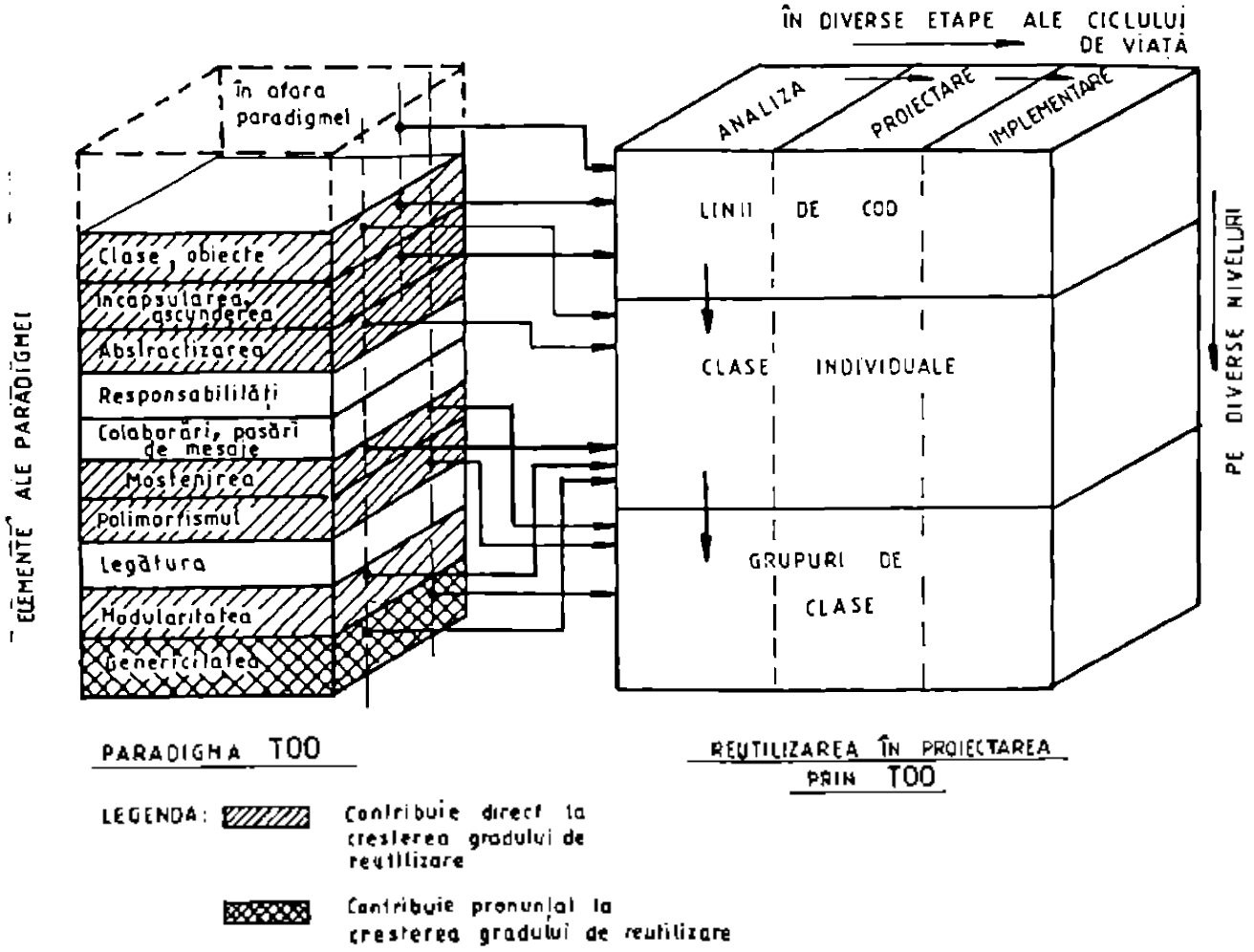


Fig. 2.5. Aspecte de *reutilizare* în cadrul **TOO**

elementele paradigmei **TOO** (§ 2.1) şi diversele aspecte ale *reutilizării*. În figură au fost evidenţiate doar conexiunile evidente. În realitate, sunt greu de realizat delimitări asupra

dependenţelor diverselor aspecte de *reutilizare* de elementele paradigmei **TOO**, întrucât întreaga paradigmă **TOO** este o abordare unitară a filosofiei proiectării software, în cadrul căreia componentele se intercondiţionează strâns. Aspectele *reutilizării* sunt structurate în două planuri : în cel al nivelurilor de *reutilizare* § 2.2), respectiv în cel al etapelor ciclului de viaţă al programelor (§ 2.3). Din acest motiv, spaţiul *reutilizării* a fost reprezentat tridimensional. Figura rezumă structura capitolului de faţă şi exprimă un rezultat esenţial al aplicării paradigmei **TOO**: contribuţia directă şi importantă a acesteia la creşterea gradului de *reutilizare* al programelor, atât la nivel individual (de proiectant) cât şi la nivel organizaţional.

În fine, scopul acestui capitol a fost de evidenţiere a aspectelor de *reutilizare* în folosirea **TOO**, de unde a rezultat structura actuală a capitolului, cuprinzând mai multe prezentări de avantaje şi dezavantaje, clasificări, ş.a.m.d., în detrimentul prezentărilor exhaustive. Considerăm că acestea au fost importante pentru a contura trăsături ale *reutilizării* folosind **TOO**, atât pentru a avea *criterii contrastative pentru alte abordări*, cât şi pentru a putea *extrapola anumite atribute*, în cadrul unor tehnici de programare care folosesc alte paradigme şi care vor fi dezvoltate în capitolele aplicative ale acestei lucrări.

CAPITOLUL 3

MĂSURI SOFTWARE UTILE ÎN PROIECTAREA PENTRU REUTILIZARE

Un demers teoretic asupra reutilizării software trebuie să includă și analiza posibilităților de cuantificare numerică a aspectelor de reutilizare. Cuantificarea este necesară din mai multe motive:

- estimarea eficienței diverselor filosofii și tehnici de reutilizare ;
- fundamentarea teoretică a unor aspecte de ingineria programării ;
- măsurarea eficienței activității individuale (la nivel de proiectant) ;
- estimarea eficienței unei anumite politici organizaționale și urmărirea realizării acesteia.

De aceea considerăm important să includem în structura lucrării de față prezentul capitol, ca fiind acela care fundamentează contribuțiile originale referitoare la măsurarea *gradului de reutilizare*, această măsură fiind urmărită cu precădere în lucrarea de față.

În prezent există multe preocupări care privesc aceste cuantificări numerice, respectiv aspecte de **metrici software**. Diverse abordări teoretice sau care pleacă de la considerente empirice au generat multe definiții de metrici pentru estimarea complexității programelor, a efortului de programare și a gradului de reutilizare, a unor aspecte care sunt cunoscute sub numele de **calitate software**.

*Capitolul de față își propune să treacă în revistă principalele abordări utile în studiul problemei enunțate, cuantificările numerice fiind de fapt aspecte de metrică software. Aspectele de metrică software analizate pot fi puse în legătură cu evaluarea complexității programelor. În literatură sunt prezentate câteva abordări uzuale referitoare la evaluarea complexității programelor, de asemenea mai multe accepțiuni pentru gradul de reutilizare, inclusiv pentru **TOO**. Cele mai importante dintre acestea sunt prezentate în acest capitol, subliniindu-se dezavantajele metodologice ale diferitelor abordări. Diverse abordări asupra fundamentării teoretice a unor aspecte ale programării furnizează criterii de validare a măsurilor software. Un asemenea criteriu este de asemenea prezentat în acest capitol. O prezentare a câtorva standarde care se impun în practica organizațională și academică, a unor medii de programare care implementează unele din măsurile prezentate și un grup de concluzii încheie capitolul.*

3.1. Complexitatea unui program[1]

Măsurile software referitoare la reutilizare sunt exprimări cantitative ale unor proprietăţi ale programelor, care sunt fundamentate pe *metrici software*. Ceea ce interesează, legat de subiectul analizat în prezenta lucrare, sunt exprimările cantitative ale complexităţii programelor. Acestea pot duce la o conexiune intuitivă cu *reutilizarea*, deoarece programele, sau modulele de program reutilizate sunt legate între ele printr-un raport de transformare asupra structurilor de program, a codului-sursă în ultimă instanţă, deci prin reutilizare se efectuează transformări, intervenăii asupra complexităţii programelor. De aceea un demers corect asupra măsurilor software referitoare la reutilizare, trebuie să înceapă cu o analiză asupra exprimării cantitative a complexităţii programelor, chiar dacă, de cele mai multe ori, aceste măsuri sunt folosite în domeniul fiabilităţii software (a se consulta în acest sens [Mih95] şi [Wey90]).

A. Principalii indicatori numerici utilizabili pentru cuantificarea complexităţii programelor sunt **indicatorii Halstead** [Fen91][2], indicatori folosiţi şi la exprimarea gradului de reutilizare. Aceştia sunt formulaţi în legătură cu proprietăţile şi structura programului. Ei permit o evaluare a complexităţii şi a timpului necesar unui programator mediu pentru a implementa un anumit algoritm. Ansamblul indicatorilor Halstead dă o idee asupra **dificultăţilor** legate de un anumit program. Ei pot fi calculaţi pornind de la următoarele mărimi:

η_1 - numărul de operatori distincţi din program,
η_2 - numărul operanzilor distinşi din program;
N_1 - numărul de incidenţe ale tuturor operatorilor;
N_2 - numărul de incidenţe ale tuturor operanzilor.

Folosind aceste notaţii se pot defini următorii **indicatori** :

a) **vocabularul** programului : $I = \eta_1 + \eta_2$; (3.1,a)

b) **lungimea observată** a programului : $L = N_1 + N_2$; (3.1,b)

c) **lungimea estimată** a programului : $\hat{L} = \eta_1 \bullet \log_2 \eta_1 + \eta_2 \bullet \log_2 \eta_2$ (3.1,c)

d) **volumul** programului : $V = L \log_2 I$ (3.1,d)

[1] In cele ce urmează ne referim la complexitatea structurală a unui program, nu la complexitatea algoritmului ; deşi între aceste complexităţi există o interdependenţă, delimitarea trebuie totuşi operată.
[2] [Fen91] poate fi considerată ca şi lucrare de referinţă în domeniul metricilor software.

e) **dificultatea** programului : $D = \frac{\eta_1 N_2}{2\eta_2}$ (3.1,e)

f) **nivelul** programului : $L_1 = \frac{I}{D}$ (3.1,f)

g) **efortul** : $E = \frac{V}{L_1}$ (3.1,g)

h) **numărul de erori** : $B = \frac{V}{3000} = \frac{E^{\frac{2}{3}}}{3000}$ (3.1,h)

i) **timpul** : $T = \frac{E}{S}$ (3.1,i)

unde S = 18 operații elementare de comparație pe secundă.

Ansamblul acestor indicatori Halstead permite o evaluare a dificultății programelor, care nu trebuie să depășească un anumit prag pentru ca fiabilitatea acestora să nu fie periclitată. Acest prag depinde de limbajul de programare folosit [Mih95].

Un exemplu (simplu) de calcul pentru indicatorii Halstead este prezentat în Anexa 2.1.

B. *Indicatorii Halstead* oferă o informație despre dificultatea de realizare a unui program. Complexitatea acestuia este exprimată prin alți indicatori. În lucrarea [Fen91] sunt prezentați feluriți asemenea indicatori, niciunul însă satisfăcător din toate punctele de vedere, există patru **indicatori ai complexității** bazați pe teoria grafurilor:

a. **Complexitatea statică** descrie programul ca pe o rețea de noduri interconectate. Fiecare nod reprezintă un modul care poate fi sau nu executat în paralel cu un alt modul. Fiecare arc reprezintă o apelare (și returul corespunzător) între module. Fie E numărul de arce și N numărul de noduri (module). Atunci, prin definiție, complexitatea statică este cantitatea :

$$C = E - N + 1 \qquad (3.2)$$

b. **Complexitatea statică generalizată** consideră, alături de modulele interconectate, resursele (de memorie, de timp) alocate atunci când controlul rulării programului trece de la un modul la altul.

Fie matricea resurselor R(K,E) având elementele :

$$r_{ki} = \begin{cases} 1 \text{ dacă resursa k este necesară la arcul i} \\ 0 \text{ în caz contrar} \end{cases}$$

unde : k = 1,2,...,K;
i = 1,2,...,E.

Fie d_k o măsură a complexității asociate alocării resursei k (de exemplu, complexitatea asociată procedurii folosite pentru a avea acces exclusiv la date comune). Fie c_i ponderea în complexitatea programului, asociată apelării și revenirii pe arcul e_i ; i = 1,2,...,E. Atunci complexitatea statică generalizată C_g este :

$$C_g = \sum_{i=1}^{E} c_i + \sum_{i=1}^{E}\sum_{k=1}^{K} d_k * r_{ki} \qquad (3.3)$$

c) **Complexitatea dinamică** se calculează folosind formula complexității statice la diferite momente de timp din rularea unui program, pentru a lua astfel în considerare faptul că întreruperile datorate apelărilor și revenirilor pot modifica în timp numărul de arce din graf. O complexitate dinamică medie pe un interval de timp pune în evidență frecvențele diferite de execuție ale modulelor și întreruperilor în timpul execuției lor.

d) **Complexitatea ciclomatică (măsura McCabe)** se utilizează pentru evaluarea complexității structurale a unui modul realizat la nivel de cod. Modulul este reprezentat printr-un graf având ca noduri instrucțiunile și ca arce trecerile controlului de la o instrucțiune la alta. Un nod din care pornesc mai multe arce se numește nod despicat. Fie **SN** (*"splitting node"*) numărul nodurilor despicate. Se numește **regiune**, un domeniu mărginit de arce care nu se intersectează. Fie **RG** numărul regiunilor. Atunci, complexitatea ciclomatică a modulului de program poate fi exprimată în trei moduri :

C = E - N + 1 (ca la complexitatea statică) (3.4,a)
C = RG (numărul de regiuni) (3.4,b)
C = SN + 1 (numărul de noduri bifurcate + 1) (3.4,c)

O altă definire a complexității ciclomatice este dată în [Ram88], după care

$$C = E - N + P \qquad (3.4,d)$$

unde P este numărul de componente conectate între ele (care pentru un modul este totdeauna 1, deci se regăsește definiția (3.4,a)).

Un inconvenient al *măsurii McCabe* este acela că nu poate diferenția programe relativ diferite, nu este adecvat pentru orice complexitate software, nedecelând întotdeauna aspectele

de lungime ale programului. Astfel, orice program compus exclusiv din structuri secvențiale are aceeași complexitate ciclomatică cu altele din aceeași categorie. Evident, astfel nu există o corespondență foarte clară între structura unui program și măsura sa, în sens McCabe. Din acest motiv, dacă măsura astfel definită este folosită mai departe în calculul gradului de reutilizare, spre exemplu, valoarea rezultată poate să reflecte situații din practică într-un mod foarte echivoc.

Măsurile software prezentate se pot calcula numai pentru grafuri conectate (în care orice nod este accesibil din oricare alt nod).

Din motive practice, se consideră că pentru un program complexitatea statică maxim admisibilă este 10 (un program mai complex este greu de urmărit, depanat, etc.). Un sistem foarte complex poate fi compus din mai multe subsisteme de complexitate statică mai mică sau cel mult egală cu 10, la fel pentru complexitățile ciclomatice ale modulelor componente [Mih85]. Complexitatea statică generalizată nu trebuie să depășească 20 * E, unde E este numărul de arce. Complexitatea dinamică este în general mai mare decât complexitatea statică. Minimizarea frecvenței întreruperilor în timpul execuției reduce complexitatea dinamică medie.

Exemple de calcul pentru complexitățile definite mai sus, sunt prezentate în [Mih95] (Anexa 2.2).

C. În [Ram88] sunt prezentate măsuri software deduse din indicatorii Halstead, folosind însă o definire mai bună a efortului. contribuția mai importantă a articolului citat o constituie însă definirea așa-numitelor măsuri **ponderate**. Acestea sunt obținute prin ponderarea numerelor N_1 și N_2. Scopul definirii unor asemenea măsuri este de a atribui ponderi în relațiile de definire pentru diversele variabile care intervin, în așa fel încât măsurile de lungime, volum și efort de programare să poată surprinde aspectele de complexitate ale structurilor de control nesecvențial. În articolul citat prezentarea se limitează la programe care îndeplinesc anumite condiții :

- fiecare modul are o singură intrare și o singură ieșire;
- graful programului este tare conex;
- structurile de control folosite sunt doar secvența, **IF ...THEN...ELSE, DO WHILE, DO UNTIL**;
- în program nu sunt predicate cu teste multiple (s. ex. **CASE**).

Înainte de a defini măsurile **ponderate**, trebuie clarificate aspectele de numerotare a nivelelor de încuibare. Astfel, un program cu structuri strict secvențiale este de nivel de încuibare zero. În același sens, este enunțată următoarea :

Teoremă : numărul *ciclomatic al grafului conținînd* ***IF...THEN...ELSE, DO WHILE*** *sau* ***DO UNTIL*** *la nivelul* **L** *este* **L+1** *dacă una din următoarele condiții este întrunită :*

1) structura de control la nivelul **L** *este cea mai încuibată structură din program;*
2) la fiecare nivel de încuibare este mai mult decât o structură de control nesecvențial;
3) programul conține ***doar*** *secvențe,* ***IF...THEN...ELSE, DO WHILE, DO UNTIL.***

Partea 2) a teoremei spune că spre exemplu nu pot fi construcții **DO WHILE** și **IF...THEN...ELSE** la același nivel de încuibare. De asemenea, rezultă că pentru fiecare construcție **IF...THEN...ELSE**, doar una din ramuri poate conține secvențe de control nesecvențial.

În continuare descriem modul de a introduce măsurile ponderate. Întâi, se definesc câteva notații. Fiind dat un modul de program, definim **T** ca fiind mulțimea operatorilor din modul. Fiecare operator trebuie să se găsească în **T** cu aceeași incidență ca și în modul. Deoarece prin definiție mulțimea nu admite repetiție, trebuie să modificăm ("*alterăm*") operatorul înainte de a-l introduce în mulțimea **T**. O metodă de modificare a operatorilor este de a marca diversele apariții ale aceluiași operator cu culori diferite. Astfel fiecare operator poate să apară de mai mult decât o dată și totuși putem să îi recunoaștem individual apariția. O tratare matematică riguroasă ar considera fiecare operator ca o pereche ordonată alcătuită din operatorul în sine ca și primă coordonată și din poziția în cadrul programului ca și a doua coordonată. Definim în continuare **R** ca fiind mulțimea tuturor operanzilor din modul. Fiecare operand are în mulțimea **R** aceeași incidență ca și în modul. Asociem fiecărui element al reuniunii **T U R**, valoarea 1 sau 0, dacă un operator sau un operand este parte a unei structuri de control, i se asociază valoarea 1; altfel, i se asociază valoarea 0. Notăm valoarea asociată fiecărui element x din **T U R**, cu δ(x). De asemenea asociem fiecărui element din mulțimea **T U R** nivelul de încuibare pe care îl notăm cu l(x). În cadrul măsurilor ponderate propuse, fiecare din cele două valori mai sus indicate interacționează asupra valorii măsurilor astfel încât fiecare operator și fiecare operand are un nivel de încuibare care afectează măsurile software ponderate doar dacă operatorul sau operandul este parte a unei structuri de control. Cele două funcții sunt definite pe **T U R** cu valori în mulțimea {0, 1}, respectiv, în mulțimea întregilor non-negativi:

$\delta(x) : T \cup R \rightarrow \{0,1\}$
$l(x) : T \cup R \rightarrow N \cup \{0\}$

Pentru a sublinia caracterul de măsură **ponderată**, se folosește indexul inferior **p** (*pondere*). Distingem următoarele măsuri ponderate:

- **totalul ponderat al operatorilor** :

$$N_{p1} = \sum_{x \in T} \left[1 + \delta(x) * l(x)\right] \qquad (3.5,a)$$

- **totalul ponderat al operanzilor** :

$$N_{p2} = \sum_{x \in R} \left[1 + \delta(x) * l(x)\right] \qquad (3.5,b)$$

- **lungimea ponderată** :

$$N_p = N_{p1} + N_{p2} \qquad (3.5,c)$$

- **volumul ponderat** :

$$V_p = N_p * \log(\eta_1 + \eta_2) \qquad (3.5,d)$$

- **efortul ponderat** :

$$E_p = \frac{(V_p)^2}{V^*} \qquad (3.5,e)$$

unde V^* este volumul de program care implementează cel mai eficient algoritmul pe care îl realizează modulul. Acest volum poate fi doar estimat. Pentru aceasta, în [Ram88] se propune relația :

$$\hat{E}_p = V_p * \left[\frac{(\eta_1 * N_{p2})}{(2 * \eta_2)}\right] \qquad (3.5,f)$$

care poate fi considerată ca o relație de aproximare a expresiei (2.5,e).

Pentru fiecare x din mulțimea T U R, $\delta(x) * l(x)$ este o pondere asociată. dacă x nu este parte dintr-o structură de control, ponderea asociată este 0, întrucât $\delta(x)$ este 0. dacă x face parte dintr-o structură de control, valoarea $|1 + \delta(x) * l(x)|$ este complexitatea ciclomatică a structurii de control. Fiecare incidență a unui operator sau a unui operand contribuie cel puțin o dată la N_{p1} sau N_{p2} și fiecare operator sau operand care face parte dintr-o structură de control contribuie într-adevăr cu valoarea egală cu complexitatea ciclomatică la N_{p1} sau N_{p2}.

În articolul citat sunt prezentate câteva aspecte cantitative rezultate din analiza comparată a unor programe, folosind măsurile propuse. Un exemplu de calcul pentru măsurile ponderate, este prezentat în Anexa 3.3.

D. O foarte interesantă metrică software este descrisă în [She95], cu atât mai interesantă pentru lucrarea de față, cu cât poate aduce sugestii pentru o mai bună definire a măsurilor software pentu estimarea complexității programelor care pot fi reprezentate prin diagrame de flux de date

(**DFD**) - adică programe multitasking, pentru conducere de proces, spre exemplu. Pentru lucrarea de față, este mai interesantă măsura software referitoare la "*diagrame informaționale globale*", adică la scheme ierarhice de reprezentare, chiar **DFD** fiind un caz particular al acestora. Măsura propusă este numită de autor **IF4** (de la "Information Flow" - diagramă informațională). Această măsură ia în considerare structura programului la care se referă, prin aceea că în evaluarea măsurii propuse, sunt înglobate în calcul valorile de **fan-in** și **fan-out** ale fiecărui modul de program. Aceste valori reprezintă numărul de conexiuni în **amonte**, respectiv în **aval**, ale unui modul de program dat. Metrica propusă se calculează după relația :

$$IF4 = \sum_{i=1}^{n} \left(FI_i * FO_i \right)^2 \qquad (3.6)$$

unde **IF4** - metrica de diagramă informațională

FI_i - "fan-in" (numărul de module care sunt conectate la intrare cu modulul i)

FO_i - "fan-out"(numărul de module care sunt conectate la ieșire cu modulul i)

Produsele sunt ridicate la pătrat pentru a crește astfel, în rezultatul final, respectiv în valoarea **IF4**, ponderea modulelor cu multe conexiuni (deci pentru a pondera mai puternic valoarea informațională crescută, în cadrul diagramei, a conexiunilor multiple).

Un exemplu complet de calcul este prezentat în Anexa 3.4. În calcul se iau în considerație și legăturile dintre module, prin intermediul unor structuri globale de date.

*Paragraful a trecut în revistă o serie de indicatori ai complexității programelor, unii încetățenit în practică și teoria software (**măsurile Halstead** și **McCabe**), alții mai elaborați sau mai puțin folosiți (**măsurile ponderate**, **IF4**). Acești indicatori furnizează măsuri ale complexității programelor, utile în demersurile teoretice în continuare, de asemenea folositoare în practica proiectării software.*

3.2. Unele evaluări numerice ale gradului de reutilizare

Exprimarea gradului de reutilizare raportează întotdeauna, sub o formă sau alta, valori caracteristice (măsuri) referitoare la două sau mai multe programe, *exprimând într-o formă numerică gradul în care fragmente de program de la proiecte anterioare se regăsesc într-un program nou.*

Gradul de reutilizare nu are o accepțiune unanim recunoscută. După cum a fost arătat în Anexa 1.3, standarde internaționale încearcă să sistematizeze nomenclatura și definițiile referitoare la măsuri software. După **Standardul IEEE Computer Society nr 1045/1992** numit

"Standard for Software Productivity Metrics" (Standard pentru metrici de productivitate software) [Sta96] : software-ul reutilizat poate fi măsurat prin numărul de **instrucţiuni sursă logice (ISL)** sau **instrucţiuni sursă fizice (ISF)** încorporate fără modificări într-un sistem. Software-ul nou poate fi măsurat prin numărul de **ISL** sau **ISF** care au fost create sau modificate pentru noua aplicaţie.

Indicatorii bazaţi pe contorizarea cazurilor de reutilizare pot fi deficitari în ce priveşte discriminarea între mai multe categorii evaluare a reutilizării : numărul de includeri ale codului în alt cod, numărul de execuţii ale codului, o combinaţie a celor două, un număr care să reflecte valoarea economică a reutilizării. De aceea, inevitabil, apar mai multe accepţiuni care coexistă. În cele ce urmează, ne vom referi strict la prima accepţiune, cea folosită şi în primul alineat din prezentul paragraf. În [Sta96] sunt citate trei categorii de indicatori care sunt utili în evaluarea gradului de reutilizare :

- indicatori de structură ai reutilizării ;
- indicatori de clasificare a reutilizării ;
- indicatori privind valoarea reutilizării.

Astfel, din [Dil93], sunt prezentaţi pentru prima categorie (indicatori de **structură**) :

- **procentul de obiecte noi** : proporţia obiectelor dintr-un sistem, care au fost scrise efectiv pentru acest sistem, exprimată procentual :

```
procent_obiecte_noi = numar_obiecte_noi * 100 / numar_total_obiecte_utilizate
```

- **structura reutilizării** : inversul proporţiei de mai sus (non-procentual) :

```
structura_reutilizarii = numar_total_obiecte_utilizate / numar_obiecte_noi
```

A doua categorie de indicatori propuşi în articolul citat (deci indicatori de **clasificare**) permit calcularea şi compararea reutilizării sistemului prin clasificarea după sursă a obiectului. Reutilizarea este *internă* dacă un obiect creat pentru un sistem este utilizat de mai multe ori în cadrul aceluiaşi sistem. Reutilizarea este *externă* dacă un obiect dintr-un sistem este utilizat cel puţin o dată în cadrul unui sistem nou. Gradul reutilizării interne va depinde de mărimea echipei de proiectanţi şi de calitatea comunicaţiei în cadrul ei. Gradul reutilizării externe depinde de calitatea **SGBS** folosit. Cei doi indicatori propuşi la această categorie de indicatori, sunt :

- **procentul de reutilizare internă** :

procentul_reutilizării_interne=100-procentul_obiectelor_noi-procentul_reutilizării_externe

- **procentul de reutilizare externă** :

procentul_reutilizării_externe = număr_obiecte_ale_altor_sisteme/număr_total_obiecte

A treia categorie nu prezintă interes pentru lucrarea de față. măsurile de mai sus nu prezintă suficientă nuanțare pentru analiza programelor în abordarea [Mit87], folosită de noi. În plus, sunt mai potrivite pentru folosirea paradigmei **POO**, ceea ce nu constituie centrul de interes al tezei. Din aceste motive, vom analiza pe larg alte măsuri software pentru exprimarea cantitativă a gradului de reutilizare. Aceste exprimări vor fi referitoare la prima accepțiune de mai sus, deci vor investiga **structural** programele supuse analizei cantitative relativ la **reutilizare**.

Majoritatea exprimărilor cantitative ale gradului de reutilizare iau în considerare numărul de variante din cadrul unei clase. **Mittermeir și Oppitz** [Mit87] propun un început de formalizare, plecând de la categoriile denumite "**aplicație**", respectiv "**task**".

A. Explorarea exprimării complexității unui program este dezvoltată de **Ramamurthy și Melton** [Ram88], care utilizează conexiuni cu teoria grafurilor, dar metoda este extrem de laborioasă, implicând descrierea exactă a structurii programului. Indicatorii definiți în articolul citat sunt derivați din indicatorii Halstead prezentați în § 3.1. Astfel, folosind definițiile (3.1,a-e) respectiv, vocabularul programului, lungimea observată, lungimea estimată, volumul și dificultatea programului, putem defini (*prin extensie de sens* - adică prin folosirea relației propuse într-un sens **original**, pentru scopurile lucrării de față, desigur, într-o formă riguroasă, acceptabilă) gradul de reutilizare ca fiind raportul dintre două "eforturi" de programare, ale **programului generic** și ale unei **variante** (în sensul din [Mit87]). În această accepțiune, gradul de reutilizare va fi un raport de complexități :

$$r_r = \frac{\hat{E}_r}{\hat{E}_g} \tag{3.7}$$

unde $\hat{E}$ este definit ca "efort" , $\hat{E}_r$ - "efortul" pentru partea reutilizabilă

$\hat{E}_g$ - "efortul" pentru întregul program generic, și

$$\hat{E} = V\frac{\eta_1 N_2}{2\eta_2}; \quad V = N \cdot \log_2(\eta_1 + \eta_2) \tag{3.8}$$

cu $N=N_1+N_2$

și η_1 - numărul operatorilor; η_2 - numărul operanzilor ;

N_1 - numărul de incidențe ale tuturor operatorilor ;

N_2 - numărul de incidențe ale tuturor operanzilor.

Definiția poate fi îmbunătățită prin utilizarea relațiilor (3.5), respectiv a măsurilor software ponderate.

Aplicarea în practică a relației de definire astfel extrapolate este relativ laborioasă (dar se pot proiecta ușor programe de evaluare care să implementeze definițiile propuse). *Dezavantaje* mai importante sunt însă altele:

- *prezentarea din articolul citat se referă la efortul de programare, în general, iar definirea de mai sus este o extrapolare fără o fundamentare decât cel mult intuitivă;*
- *exprimarea se referă la "efortul" de programare, deci rapoarte ale unor elemente de "efort" de programare (așa cum apar în relațiile de definire de mai sus) nu sunt de fapt prin nimic legate de reutilizarea propriu-zisă (ca reluare a unor linii-sursă precis delimitate);*
- *în definițiile prezentate nu există referiri structurale și la ierarhia modulelor programului generic și ale variantelor;*

B. Kitchenham [Kit87] (citat în [Ter94]) dă următoarea relație de definire pentru gradul de reutilizare (numit în lucrarea citată, **metrică a reutilizabilității**) :

$$M_r = \sum(cef_c - eff_n) * n \tag{3.9}$$

unde M_r - metrica reutilizabilității (gradul de reutilizare)

eff_c - efortul pentru a produce componenta **c**

eff_n - efortul de a încorpora componenta într-un sistem nou

n - numărul previzionat de sisteme noi care vor utiliza componenta

Definiția aceasta este generală, nefiind propriu-zis instrumentală (nu oferă o explicație a modului cum se poate calcula efortul de a produce o componentă sau de a încorpora componenta într-o aplicație nouă). Această definiție are o conotație **globală**, întrucât ține cont de aspectele de reutilizare software ale unor componente software, în toate aplicațiile în care această componentă va fi refolosită. Deoarece în relația de definire apare "efortul" de programare, această relație este asemănătoare cu definirea precedentă.

C. Weyuker [Wey90] prezintă o metodă de exprimare a complexităţii programelor folosind un criteriu cu un caracter subliniat empiric. Metoda este folosită pentru estimarea necesarului de activitate pentru testarea unui program. Fiind vorba de un indicator cantitativ legat de numărul de bucle de program de testat, se poate realiza o **extensie de sens (originală)** pentru numărul de variante în cadrul unei clase. Astfel, propunem o utilizare a definiţiei din [Wey90], pentru conturarea gradului de reutilizare. În aceste condiţii, se poate defini un grad de reutilizare ca următoarea valoare medie :

$$E = \frac{1}{n}\sum_{i=1}^{n} s_i \quad , \qquad \text{unde} \quad s_i = \frac{d_i}{t} \tag{3.10}$$

n - numărul variantelor;
d_i - numărul de decizii în programul (varianta)$_i$;
t_i - numărul valorilor de intrare;

Relaţia (3.10) nu poate totuşi exprima corect toate situaţiile din practică, deoarece nu asigură suficientă nuanţare a acestor situaţii. Astfel, în relaţie nu este reflectată apartenenţa la aceeaşi clasă de programe, decât sub forma unei simple ponderări.

Din [Wey90], respectiv din [Ram88] este de reţinut ideea exprimării complexităţii programelor în funcţie de numărul de structuri logice de decizie conţinute.

D. Browne ş.a. [Bro90] prezintă o abordare "clasică" şi foarte simplă pentru calculul gradului de reutilizare. Pornind de la metoda de clasificare numită "*structurată - relaţională*", autorul citat defineşte o metrică relativ la componentele de program reutilizate, timpul de dezvoltare al programelor şi rata erorilor în programele obţinute. Tot în articolul citat sunt introduşi indicatori calitativi care reflectă gradul în care sistemul răspunde cerinţelor impuse de utilizator. Toţi aceşti indicatori sunt evaluaţi experimental. Browne foloseşte reprezentarea programelor prin grafuri care descriu structura acestora, conţinând noduri care reprezintă prelucrări secvenţiale. Corespunzător, se definesc două cardinale, pentru liniile de cod generate :

LOC $_g$ pentru liniile de cod corespunzătoare **structurii** (aceasta este o mărime de "*proiectare*"), respectiv
LOC $_n$ pentru **părţile de program secvenţiale**.

Atunci, numărul total de linii de program este :

$$LOC = LOC_g + LOC_n \tag{3.11}$$

Notăm cu **R (.)** numărul de linii-sursă reutilizate.

Se definesc trei grade de reutilizare :

- gradul de reutilizare al proiectării :

$$r_D = \frac{R(LOC_g)}{LOC_g} \qquad (3.12)$$

Acest grad de reutilizare se calculează ţinând cont de aspecte structurale, întrucât în calcul intervin acele elemente de cod-sursă (chiar dacă gradul de reutilizare se numeşte al **proiectării** şi nu al codului) care definesc structura programului (decizii, ramificări, bucle, etc.). Între aceste linii de cod se pot insera oricâte alte linii care corespund unor procesări *strict* secvenţiale, fără ca logica şi forma grafică de prezentare a programului să se schimbe sensibil.

- gradul de reutilizare al codului :

$$r_C = \frac{R(LOC_n)}{LOC_n} \qquad (3.13)$$

Acest grad de reutilizare poate fi asociat cu la reutilizarea **cantitativă** a liniilor-sursă, surprinzând deci aspectul brut de reutilizare, cel de refolosire în înţelesul imediat, de portare a unor fragmente de program, de cod-sursă, de la un program scris la un program în curs de proiectare.

- gradul de reutilizare al programului este definit ca o sumă ponderată a gradelor definite prin relaţiile (3.12) şi (3.13), adică :

$$r_p = \delta\, r_D + \varepsilon\, r_C \ , \quad \text{cu} \quad \delta + \varepsilon = 1 \qquad (3.14)$$

Definirea aceasta combină practic, prin intermediul unor **factori de ponderare empirici**, ambele grade de reutilizare definite anterior. Astfel rezultă un grad de reutilizare pe care îl putem caracteriza ca **global** pentru un program dat, surprinzând atât aspectele de reutilizare de structură, cât şi cele de reutilizare de cod.

Bibliografia dă valori diverse pentru factorii de ponderare. În [Bro90] se folosesc valorile $\delta = 0.57$ şi $\varepsilon = 0.43$.

Această din urmă reprezentare este o exprimare "naturală" pentru gradul de reutilizare, întrucât în expresiile de mai sus apar direct cardinale de linii-sursă, de bază (adică, ale "programului generic"), respectiv reutilizate. însă nu se poate spune că această reprezentare cuantifică toate aspectele de reutilizare, întrucât în ceea ce privește liniile-sursă reutilizate, putem face următoarele nuanțări :

- unele linii-sursă afectează în mică măsură funcțiile de bază ale modulului reutilizat, deci, pentru o cuantificare exhaustivă, ar fi necesară o ierarhizare a "importanței" unor linii-sursă reutilizate;

- nu e întotdeauna ușor să realizăm o delimitare clară între modificări "formale" și modificări "propriu-zise" [3]*;*

- unele modificări structurale pot aparent să nu determine o modificare sensibilă a gradului de reutilizare definit ca mai sus (spre exemplu, modificarea ordinii liniilor-sursă);

- definițiile de mai sus nu includ sub nici o formă aspectele de ierarhie în program;

Totuși, această din urmă definiție poate fi considerată o apreciere globală, utilă, ușor de folosit, a gradului de reutilizare, fiind din acest motiv folosită în lucrarea de față ca valoare primară pentru calculul unui grad de reutilizare care să exprime mai nuanțat aspectele de reutilizare (Cap. 4). Gradele de reutilizare prezentate la alineatele **A** și **C** constituie încercări de **extindere** a unor definiții folosite pentru alte exprimări cantitative, dar demersul teoretic corespunzător este util întrucât sugerează aspecte utile în găsirea unor definiții pentru măsuri software mai nuanțate (v. Cap. 4). Gradul de reutilizare prezentat la alineatul **B** sugerează aspecte utile referitoare la definirea unor metrici software (v. Cap. 5).

Definițiile de mai sus prezintă mai multe perspective asupra cuantificării exprimării gradului de reutilizare, fiecare perspectivă cu motivații de ordin teoretic sau empiric și cu avantaje și dezavantaje specifice. Doar două dintre gradele de reutilizare prezentate au fost definite în bibliografie ca atare, celelalte reprezentând extinderi de sens sau interpretări originale. Prezentarea a fost direcționată pe analiza avantajelor și dezavantajelor diverselor definiri ale gradului de reutilizare, în vederea găsirii criteriilor pentru conceperea unui nou grad de reutilizare, mai avantajos decât cele găsite în literatură, fie ca atare, fie reinterpretate.

[3] Spre exemplu, modificarea numelui unei variabile, în fragmentul de cod refolosit, nu constituie propriu-zis o modificare, din punctul de vedere al reutilizării software, în înțelesul lucrării de față.

3.3. Gradul de reutilizare în POO

Grupul de studiu de la Universitatea Tehnică din Magdeburg propune în [Dum90] şi [Neu91] mai multe relaţii de exprimare a unor aspecte de metrică software referitoare la reutilizare în **POO**. Evident, metricile utilizate trebuie să se raporteze la aspectele specifice ale acestei abordări (v. cap. 2). Aici, sunt mai puţin importante mărimi precum numărul de linii de cod (ca în exprimările din [Bro90]), dar capătă importanţă majoră aspecte privind virtualitatea, genericitatea, moştenirea, etc.

Mai multe asfel de relaţii sunt prezentate în continuare. În toate aceste relaţii, măsurile propuse sunt raportate la un lot de clase care fac obiectul investigaţiilor de cuantificare, efectuate de relaţiile respective. măsurile propuse, spre deosebire de cele prezentate în paragrafele precedente (unde au fost prezentate măsuri software referitoare fie la un program, fie relativ la două programe puse în relaţie de reutilizare unul cu celălalt), sunt măsuri globale (relativ la mulţimea investigată). Ideea definirii unor măsuri globale pare foarte atrăgătoare, mai ales în contextul definirii şi implementării unor politici organizaţionale coerente pentru reutilizare.

a) **Virtualitatea medie** [Neu91] exprimă măsura în care sunt folosite metode virtuale în cadrul claselor investigate.

$$VIRT = \frac{\sum\left(\frac{vm}{m}\right)}{n} \qquad (3.15)$$

cu : n - numărul claselor investigate
m - numărul metodelor dintr-o clasă
vm - numărul metodelor virtuale

Valoarea astfel definită dă o idee privind răspândirea utilizării metodelor virtuale în cadrul claselor investigate, deci implicit gradul de apelare la elementul paradigmei **TOO** numit legătură (v. § 2.1). Prin conexiunea directă între elementele paradigmei **TOO** şi reutilizare, măsura propusă constituie o cuantificare a uneia din faţetele posibile ale **reutilizării**.

b) Genericitatea medie :

$$GENER = \frac{\sum_{m}^{\cdot} \sum \left(\frac{cgm}{cm} \right)}{n} \qquad (3.16)$$

unde : cm - numărul de componente într-o metodă ;
cgm - numărul de componente generice.

Astfel se poate cuantifica gradul în care se foloseşte în cadrul grupului de clase investigate, elementul de **genericitate** (v. § 2.1) :

c) Reutilizabilitatea medie :

$$REUSE = \frac{\sum (gT)}{n-1} \qquad (3.17)$$

cu gT - numărul de subclase egale

Această valoare furnizează informaţii despre măsura în care se ramifică ierarhiile de **moştenire** în cadrul claselor analizate (deci despre extensia "orizontală" a acestor ierarhii).

d) Complexitatea medie de moştenire este măsura complementară celei de mai sus :

$$ICOMP = \frac{\sum (sd)}{\frac{n}{2} - 2} \qquad (3.18)$$

cu : sd - numărul de superclase directe.

Valoarea de mai sus oferă informaţii despre extensia "verticală" în cadrul ierarhiilor de **moştenire** din setul de clase analizate.

Aceste măsuri sunt definite conform concluziilor cap. 2 referitoare la legătura directă între elementele paradigmei **TOO** şi reutilizare. Evident, analiza este superfluă în cazul grupării ca şi clase de investigat a unor clase disparate, fără legătură între ele. Analiza numerică prin intermediul relaţiilor prezentate îşi are sensul în cazul unor clase grupate în ierarhii, orientate spre deservirea unor cerinţe dintr-o familie clar delimitată.

De aceea, aceste măsuri sunt valoroase pentru programe complexe, dezvoltate cu **TOO**, în care sunt înglobate ierarhii complexe de clase. În această situație, măsurile globale prezentate (măsurile globale în general) își dovedesc utilitatea, cu atât mai mult cu cât aici numărul de linii-sursă spre exemplu, dar și alte măsuri de estimare a complexității programelor bazate pe structurile interne ale programelor, sunt sau irelevante sau inaplicabile.

*Măsurile software pentru reutilizare în **POO** sunt, după cum s-a arătat mai sus, calculabile doar la nivelul **global** al unei **ierarhii de clase**. Ideea folosirii unor măsuri software globale pentru reutilizare este aplicabilă, considerăm noi, de asemenea, în alte paradigme decât **TOO**.*

3.4. Instrumente software pentru evaluarea metricilor programelor

Relațiile de definire pentru măsurile software introduse în § 3.1, 3.2 și 3.3 sunt instrumente de lucru dificil de utilizat în majoritatea cazurilor, deoarece volumul de activitate pentru colectarea sau deducerea datelor necesare aplicării acestora este de cele mai multe ori însemnat. Pentru a putea totuși utiliza aceste relații, se impune automatizarea operațiilor aferente. Eforturi ale unor grupuri de studiu au dus la specificarea și realizarea unor instrumente software destinate automatizării operațiilor de evaluare a unor seturi de măsuri specifice, de genul celor prezentate în paragrafele precedente.

A. *Instrumente software pentru evaluarea metricilor programelor convenționale*

Am numit în formularea de mai sus programe convenționale, acele programe care au fost dezvoltate folosind paradigme diferite de paradigma **TOO**.

Un astfel de program (**"SVS"**) este descris în lucrările [Lei90], [Lei91], [Tie92]. Scopul acțiunilor programului este de a implementa un instrument bazat pe un model de calitate care poate fi aplicat pentru toate fazele procesului de dezvoltare software. Evaluarea propriu-zisă este divizată în factori de calitate, criterii și subcriterii și poate să fie efectuată pentru etape ale ciclului de viață al programelor : analiza, proiectarea, codificarea și testarea. În cadrul subcriteriilor se regăsesc majoritatea măsurilor menționate în § 3.1 (s. ex. măsurile Halstead și McCabe). Aceste măsuri sunt raportate la o scală definită de la 1 la 10.

B. *Instrumente software pentru evaluarea metricilor programelor dezvoltate prin* ***TOO***

Alte instrumente software prezentate în bibliografie sunt destinate implementării determinării unor măsuri ale complexității programelor și a unor măsuri precum cele prezentate în § 3.4, deci pentru **POO**. Rezultatele obținute prin evaluare permit compararea programelor scrise în ideea folosirii paradigmei **TOO**, cu alte stiluri de programe.

B1) În [Neu92] este descris modul în care extensiile **Smalltalk/V** propuse pot fi folosite pentru estimarea următoarelor categorii de metrici software, pentru programe scrise în **Smalltalk** :

- măsuri **generale** : număr de clase, adâncimea ierarhiei, lățimea ierarhiei, numărul mediu de metode pe clasă, numărul mediu de variabile pe clasă, numărul mediu de subclase ;
- măsuri ale **claselor** : numărul metodelor din clasă, numărul instanțelor de metode, numărul de variabile din clasă, numărul de instanțe de variabile, numărul de subclase ;
- măsuri de **metodă** : numărul de linii de cod, măsura McCabe.

B2) Instrumentul **M++** propus de Ines Kuhrau [Kuh93] furnizează valorile metricilor de mai sus pentru programe scrise în **C++**.

Sunt predefinite "valori critice" pentru măsurile software implementate, deci programul semnalează scăderile sau depășirile sub/peste pragurile critice. Programul furnizează fișiere **EXCEL** pentru prezentări grafice ale rezultatelor. Pe baza evaluării unui program nou proiectat, se pot trage concluzii asupra calității proiectării și eventual se poate impune reproiectarea în situația în care indicatorii calitativi nu sunt mulțumitori. Deciziile de încadrare calitativă pot fi parte a efortului individual de proiectare sau mai ales parte a unui efort organizațional coerent de creștere a calității și a reutilizabilității.

B3) În lucrarea [Ter94] este descris un instrument software de tip **CASE**, numit **QDAtool** (de la "**Q**uality **D**riven **A**ssessment **T**ool" - metoda de realizare (a programelor) prin asigurarea calității) care implementează, pentru programe scrise în **Smalltalk** , facilități de instrument **CASE**, având în plus înglobate instrumente de evaluare a metricilor calității software. Un exemplu de aspect al ferestrei de afișare pentru acest program este dat în Fig. 3.1.

Se observă că instrumentul înglobează și o funcție de evaluare a gradului de reutilizare ("**Reusability**") al programelor scrise conform paradigmei **TOO**. Metrica de reutilizabilitate folosită este cea din rel. (3.9).

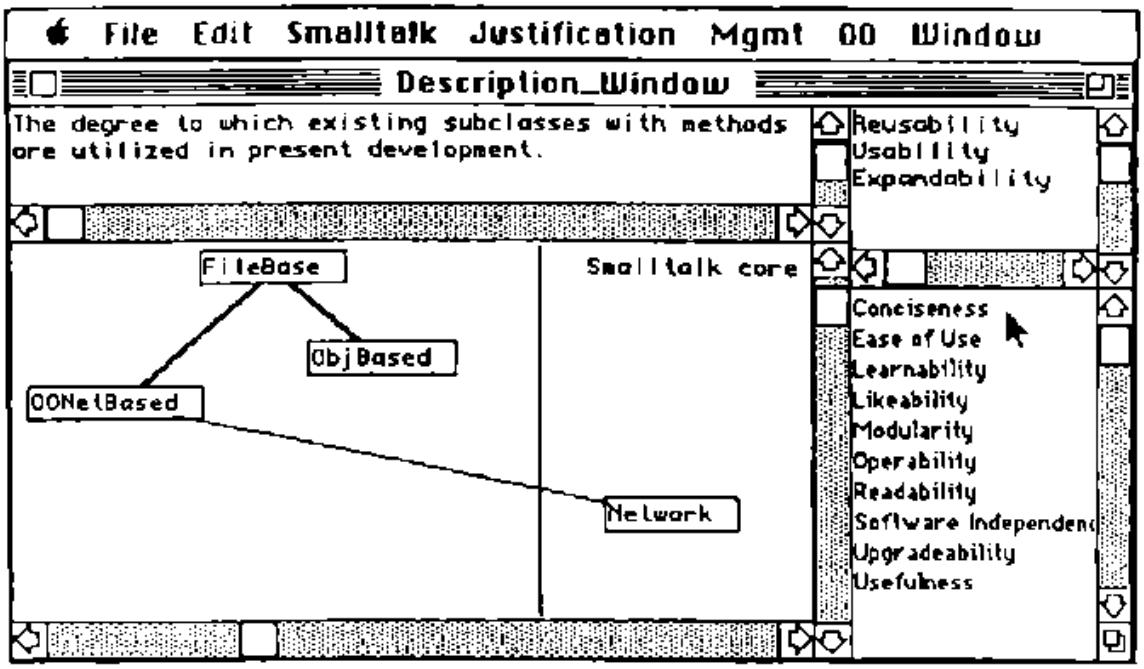


Fig. 3.1. Fereastra **Description_Window** a programului **QDAtool**

B4) **Cadrele de măsură** (*"measurement framework"*) sunt instrumente software care acoperă întregul proces de dezvoltare de software, deci pun la dispoziție instrumente de evaluare pentru toate etapele ciclului de viață al programelor. Prezentări de astfel de medii sunt realizate în lucrările [Dum93-G], [Dum93-S], [Dum94].

În concepția noastră, este indicat, atât din punctul de vedere al efortului de dezvoltare a programelor cât și din acela al efortului de dezvoltare a instrumentelor software în sine, ca instrumentele de evaluare software să fie înglobate în medii mai extinse, de tip **CASE** (v. Cap. 1). Astfel, utilizatorul beneficiază de mai multe avantaje :

- în mediul care asistă procesul proiectării (de tip **CASE**) utilizatorul are la îndemână instrumentele de evaluare astfel încât poate să efectueze corecții în cazul în care rezultatele evaluărilor nu se încadrează în limitele strategic fixate ;
- productivitatea creăte datorită folosirii unui mediu unic atât pentru dezvoltarea propriu-zisă de programe cât și pentru evaluare ;
- funcțiile de asistare a dezvoltării de programe și funcțiile de evaluare fiind integrate în același mediu, utilizatorul va fi mai degrabă determinat să apeleze la funcția de evaluare decât dacă această funcție ar fi înglobată într-un mediu distinct.

Proiectantul mediilor de programare şi de evaluare are şi el sarcina uşurată prin comasarea de mai sus, din următoarele motive :

- va trebui să realizeze un singur mediu integrat ;
- o serie de facilităţi (manipulare de fişiere, funcţii auxiliare, etc.) vor putea fi utilizate atât de funcţiile specifice instrumentelor **CASE** cât şi de funcţiile de evaluare.

Din motivele enumerate, mediile de dezvoltare de tip **CASE** dezvoltate de autor vor îngloba de preferinţă şi facilităţi de evaluare software. Acesta poate să devină, în strategia organizaţională, chiar o necesitate [Sta96][Ban94].

*Din succinta enumerare a unor medii de evaluare a măsurilor software concluzionăm că pe de o parte există unele preocupări de dezvoltare a acestui domeniu, pe de altă parte că în opinia noastră este utilă înglobarea facilităţilor de evaluare software în mediile de tip **CASE**. Pentru evaluarea software, folosirea unor programe de evaluare este o **necesitate**.*

3.5. Concluzii

Măsurile software utilizabile pentru cuantizarea unor aspecte de reutilizare pot fi obţinute plecând de la indicatori cantitativi pentru exprimarea **dificultăţii** sau a complexităţii programelor (indicatorii **Halstead**, **complexitatea ciclomatică**). O exprimare nuanţată a complexităţii o reprezintă măsurile **ponderate** (rel. (3.5)), care oferă premisele unei legături mai puternice între structura unui program şi indicatorii cantitativi de evaluare a complexităţii acestuia.

Plecând de la reprezentările enumerate, se pot defini diverse grade de reutilizare. În § 3.2 sunt prezentate două extrapolări plecând de la abordări existente, referitoare la exprimarea **efortului** de programare, respectiv a necesarului de activitate pentru **testarea** unui program. Este detaliată şi definirea **clasică** a gradului de reutilizare. Se cuvine să subliniem că nu există accepţiuni unanim vehiculate ale gradului de reutilizare, iar identificarea unor metrici specifice este problematică [Ter94].

Asupra celor de mai sus, se pot trage câteva concluzii :

- cu excepţia primului indicator, ceilalţi sunt "empirici" sau "experimentali" (în formularea autorilor);
- aplicarea relaţiilor de definire este câteodată destul de dificilă;
- valorile definite nu reflectă suficient de "fin" aspecte de structură;

- definiţiile de mai sus nu sunt mulţumitoare pentru programe multitasking;
- exprimările prezentate sunt nişte evaluări "amorfe", uniformizante, deci eventual potrivite pentru studii comparative; nu sunt suficient de consistente, suficient de apropiate de o fundamentare teoretică, pentru ca aceasta să le confere un caracter mai puţin relativ;
- definiţiile se referă strict la refolosirea software într-un program şi deci nu pot caracteriza global variantele dintr-o clasă.

Astfel, apare necesitatea formulării unor noi definiţii pentru gradul de reutilizare, care să înlăture dezavantajele menţionate mai sus. Capitolul următor va fi o continuare a investigaţiilor pe această direcţie.

Multă vreme s-au folosit în ingineria software metrici unidimensionale pentru a exprima noţiuni complexe. Experienţa de proiectare arată că, în contrast cu încercarea de a efectua o sinteză între diverşi indicatori, este de dorit o abordare multidimesională, care să ţină seama de diferitele puncte de vedere legate de fazele de dezvoltare ale produsului. Problema unei metrici potrivite, uşor de folosit şi utile, este o problemă sensibilă în ingineria programării. Unele încercări de definire a metricilor potrivite exprimării cantitative a gradului de reutilizare există şi pentru **POO**. Grupul de studiu de la Magdeburg (v. § 3.3) a definit astfel de măsuri, de asemenea a realizat **instrumente software** de investigare a programelor din perspectiva măsurilor software. **Ideea definirii unor măsuri globale** este utilă lucrării de faţă. De asemenea, considerăm potrivită o **înglobare** a funcţiilor de **evaluare** software în medii **CASE**. Avantajele acestei strategii de proiectare a instrumentelor software sunt prezentate în § 3.4

În concluzie, reţinem următoarele aspecte esenţiale, utile în capitolele următoare :

- *metrici folosite actualmente pentru evaluarea calităţii în reutilizarea software prezintă anumite **dezavantaje** care justifică efortul de căutare a noi metrici ; astfel, metricile actuale nu sunt adecvate pentru sistemele multitasking şi nu exprimă suficient de mulţumitor aspectele structurale, iar cele mai multe abordări sunt **empirice** ;*
- *unele propuneri sugerează definirea unor măsuri **globale**, care să ia în considerare aspecte relevante pentru programele cu multe module (e de preluat ideea din § 3.3, folosită acolo pentru **POO**) ;*
- *metrici de software pot fi calculate eficient doar prin **implementarea** unor facilităţi specifice de evaluare, în medii **CASE** ;*

Toate aceste observaţii vor fi folosite intensiv în cadrul următoarelor capitole ale lucrării

CAPITOLUL 4.
ABORDĂRI FUZZY ÎN PROIECTAREA PENTRU REUTILIZARE

S-a arătat în cap. 3 că adesea metricile software utilizate în exprimarea cantitativă a unor atribute ale programelor sunt lipsite de "subtilitatea" necesară exprimării a tot ceea ce se doreşte să se exprime (a se vedea *completitudinea* ca principiu al ingineriei programării [Dav86]). De aceea în prezentul capitol considerăm noi principii de evaluare care să surmonteze dezavantajele menţionate la concluziile cap. 3. Aceste principii vor permite definirea unor măsuri software mai intim legate de natura însăşi a reutilizării software, atât din punct de vedere structural, cât şi din punctul de vedere al volumului de program (de cod) reutilizat. Pentru aceasta, vor trebui conturate noi accepţiuni ale gradului de reutilizare. Propunerea noastră se va referi atât la aceste noi accepţiuni cât şi la un principiu de abordare fuzzy a conceptelor acestui domeniu.

Capitolul prezintă noi accepţiuni ale gradului de reutilizare, respectiv modalităţi de definire şi de calcul folosind logica fuzzy (cu un exemplu de aplicare). O teorie fuzzy referitoare la reutilizare software în cadrul unei clase de aplicaţii întregeşte abordarea fuzzy propusă.

4.1. Noi accepţiuni ale gradului de reutilizare

Vom defini în continuare, din punct de vedere calitativ, două grade de reutilizare, referitoare la programele unei clase de aplicaţii (conceptul de clasă fiind utilizat în accepţiunea din [Mit87]) :

a) gradul de reutilizare al unui program relativ la programul generic al clasei sale

r_p - grad de reutilizare de program

b) gradul de reutilizare al programelor din cadrul unei clase, ca măsură globală a calităţii întregii clase, din punctul de vedere al reutilizării :

r_c - grad de reutilizare de clasă

(4.1) Definiție : ***Gradul de reutilizare de program*** *este o măsură a gradului în care structura și codul programului generic al unei clase, se regăsesc în cadrul structurii și codului unui program oarecare din cadrul acelei clase.*

Se pune problema în ce mod diferă gradul de reutilizare propus în acest capitol de cel definit bunăoară de Browne ([Bro90] - vezi § 3.3). Sau : ce interpretare (utilizare) se poate da gradului de reutilizare astfel obținut. Apreciem că gradul de reutilizare propus aici elimină în bună măsură, dacă nu chiar total, dezavantajele subliniate în concluziile capitolului 3, prin aceea că modul de definire ține cont atât de aspectele subiective (prin includerea acestora în algoritmi *fuzzy* de evaluare) ale aprecierii acestui grad cât și de aspecte structurale propriu-zise ale programelor (conform modului prezentat mai jos) ; astfel se obține un indice care să reflecte mai bine toate aspectele reutilizării.

(4.2) Definiție : ***Gradul de reutilizare de clasă*** *este o măsură globală a gradului de asemănare a variantelor în cadrul unei clase, respectiv a gradelor de asemănare ale acestora cu programul generic.*

Gradul de reutilizare de clasă este un indicator *original*, întrucât el se referă la cuantificarea unor calități de reutilizare, pentru toate programele clasei, privite în ansamblu, aspect neîntâlnit în literatura de specialitate. Acest grad de reutilizare permite aprecierea modului de individualizare ("limitare") a unei clase, măsurând "cât de strâns" sunt alese caracteristicile clasei, care permit încadrarea unor programe ca și variante ale programului generic. Cu cât acest grad de reutilizare este mai ridicat, cu atât programele din cadrul clasei sunt mai asemănătoare între ele, fiind obținute din programul generic cu modificări mai mici. Adică, în cadrul clasei, reutilizarea este mai pronunțată. Evident, în practică trebuie să existe un echilibru între un grad de reutilizare cât mai ridicat - pentru ca productivitatea proiectării să fie cât mai bună și implicit posibilitatea folosirii tehnicilor de reutilizare, mai ridicată - și cerința de a obține aplicații mai diverse, pentru a acoperi o plajă mai largă de specificații posibile. De aceea, din punctul de vedere al strategiei organizaționale este inutil un efort de creștere continuă a gradului de reutilizare de clasă, întrucât peste o anumită limită aceasta se poate realiza doar cu prețul îngustării nepermise a domeniului clasei, dincolo de care tehnicile de reutilizare nu își mai au rostul.

Primul grad de reutilizare propus este similar celor prezentate în § 3.3, în sensul că raportează un program la altul, față de care au fost făcute modificări și preluări de cod-sursă, în sensul reutilizării. Al doilea grad de reutilizare propus este un grad global, caracterizând proprietatea de reutilizabilitate a întregii clase de programe.

4.2. O exprimare fuzzy a gradului de reutilizare

Având în vedere natura preponderent vagă a caracterizării reutilizării programelor, în contextul concret al reutilizării în sine, se propune, în cele ce urmează, un principiu de evaluare bazat pe logica fuzzy [Sto94-D]. Propunerea se referă la calculul gradelor de reutilizare definite în paragraful precedent, care înglobează atât aspecte ale reutilizării codului cât şi ale structurii, în înţelesul din [Bro90]. Spre deosebire însă de sursa citată, structura luată în considerare nu e structura internă a unui modul de program (de altfel, gradul de reutilizare [Bro90] este calculabil mai ales pentru programe seriale, adică non-multitasking) ci a unei aplicaţii multitasking. În lucrarea de faţă se foloseşte diagrama de flux de date pentru a reprezenta astfel de programe. Deoarece programele de conducere de proces sunt de cele mai multe ori multitasking, modul de calcul prezentat mai jos se poate aplica tocmai pentru asemenea programe.

Vom prezenta întâi principiul de calcul pentru r_p. Plecăm de la ipoteza că structurile programului generic şi a unui program oarecare din cadrul unei clase vor semăna, vor putea fi chiar parţial suprapuse. "Măsura" în care cele două grafuri se suprapun sugerează valoarea gradului de reutilizare de program. În absenţa unor criterii uşor cuantificabile evaluarea acestui grad de reutilizare se poate realiza numai printr-o logică *fuzzy* (*vagă*). Pentru a reda principiul, folosim reprezentarea unui program printr-un graf [Ram88], reprezentare echivalentă diagramelor de flux de date (**DFD**)

Fie ramificaţia pe două niveluri, din Fig 4.1, aparţinând programului generic, rezultată prin separare, în care $P_{i,j}$ reprezintă taskuri (procese) - sau module de program altfel delimitate, în raport cu reprezentarea folosită -iar $r_p^{i,j}$ valorile gradelor de reutilizare de cod (grade de reutilizare primare) în sensul definit de rel. (3.6), i fiind numărul de ordine al nivelului pe care se află, iar j - numărul de ordine al taskului pe nivel. Toate nodurile grafului sunt completate similar. Calculul gradului de reutilizare de program se realizează recurent, de jos în sus, asociind pentru fiecare ramificaţie un grad de reutilizare echivalent care va înlocui pe mai departe în calcule întreaga structură până atunci subordonată. De aceea, în cele ce urmează, se prezintă doar calculul pentru structura elementară din Fig. 4.1. Ideea se aseamănă întrucâtva cu abordarea din [Ram88], referitoare la măsurile ponderate. Într-adevăr, astfel se pot lua în considerare toate aspectele structurale, de la toate nivelele, cu o ponderare dictată de importanţa nivelelor ierarhice pe care sunt situate diferite

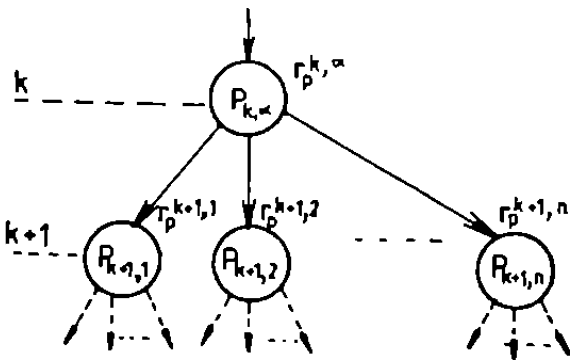


Fig. 4.1. O ramificaţie a grafului unui program

module, aşa cum în abordarea din articolul citat, complexitatea în cadrul unui modul de program era ponderată cu nivelul de încuibare. O altă referinţă poate fi considerată lucrarea [She95] (calculul **IF4** -v. § 3.1, D).

Este potrivit să subliniem, înainte de a prezenta elementele propriu-zise de logică fuzzy, aplicate pentru calculul dorit, motivele care justifică abordarea fuzzy. Motivele acestea îşi află originea în observaţiile de la § 3.7, referitoare la dezavantajele gradelor de reutilizare definite în diverse surse bibliografice. Ceea ce încercăm să rezolvăm prin definirea fuzzy a gradului de reutilizare, este să atribuim acestuia caracteristica de a exprima o mai strânsă legătură între valoarea sa şi structura programului multitasking, dar în aşa fel încât să luăm în considerare şi structura ierarhică a programului. De asemenea, exprimarea prezentată îmbină caracteristicile unei definiri numerice cu flexibilitatea logicii fuzzy, cu posibilitatea acestei logici de a se putea uşor adapta la criterii subiective de apreciere. Deci să schiţăm în continuare elementele fuzzy ale definirii propuse.

Pentru o abordare în logică fuzzy, se porneşte prin desemnarea ca variabilă lingvistică a gradului de reutilizare $r_p^{(\cdot)}$, cu universul de discurs acoperit de următoarele 5 valori lingvistice : **Scăzut (S)**, **Mic (Mi)**, **Mediu (Me)**, **Mare (Ma)** şi **Ridicat (R)**. Acestora li se pot asocia de exemplu funcţiile de apartenenţă din Fig. 4.2

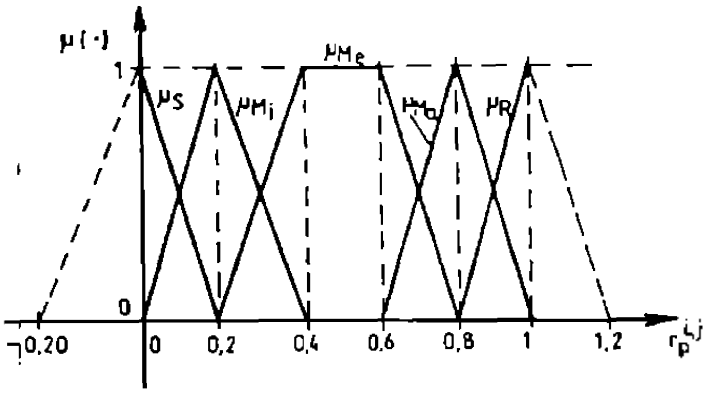


Fig. 4.2. Funcţiile de apartenenţă

Rezultatul evaluării trebuie să ţină cont de toate gradele de reutilizare de la nivelul inferior. Valoarea "1" asociată lui $r_p^{i,j}$ denotă că taskul respectiv coincide cu cel din programul generic. Totodată, pentru ramurile care nu sunt refolosite, se va considera $r_p^{i,j} = 0$ (până la completarea la nivelul programului generic).

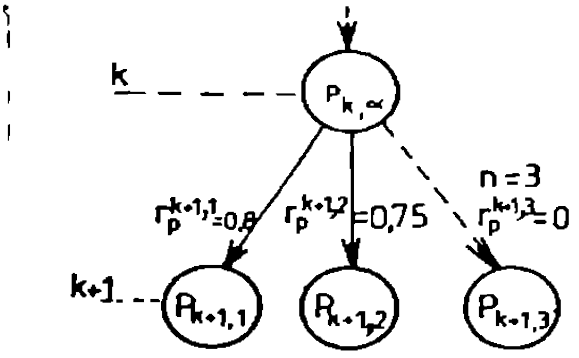


Fig.4.3. Completarea nodurilor dacă există ramuri nefolosite

În Fig. 4.3 se prezintă o situaţie particulară pentru a exemplifica cele de mai sus. În acest exemplu, taskul $P_{k+1,3}$ nu există în programul analizat, în consecinţă, gradul său de reutilizare este nul ($r_P^{k+1,3} = 0$). Gradul de reutilizare asociat nivelului k, nodul α, va fi

calculat luând în considerare acest grad de reutilizare nul.

Pentru **inferență**, propunem ca setul de reguli să se elaboreze plecând de la asocierile de tipul celor din Tabelul 4.1. Modul de lucru cu tabelul este următorul : o linie a tabelului activează o regulă de inferență numai în situația în care este satisfăcută inegalitatea la care se referă operatorii algebrici $<, \leq, =, \geq, >$. Prin Card $\{ j \mid r_P^{ij} \sim V \}$ (vezi Tab. 4.1) se înțelege numărul de grade de reutilizare care intră în calcul la un nivel dat și care satisfac operația de comparare fuzzy $\sim$ (simbolul $\sim$ ține loc de $\lesssim, \lessapprox, \eqsim, \gtrapprox, \gtrsim$) cu valoarea lingvistică V.

Tabelul 4.1. Asocierile pentru generarea regulilor de inferență
(pentru m = n + 1 și j,k ∈ { 1,2,...,m})

Regu-la nr.	Regulile de asociere ale premisei	Valoarea concl.
1	card $\{j \mid r^j \lessapprox S\} \geq [2m/3]$	S
2	card $\{j \mid r^j \lessapprox S\} \geq [m/2] \wedge$ card $\{k \mid r^k \gtrapprox Ma\} < [m/3]$	Mi
3	card $\{j \mid r^j \lessapprox S\} \geq [m/2] \wedge$ card $\{k \mid r^k \gtrapprox Ma\} \geq [m/3]$	Me
4	card $\{j \mid r^j \lessapprox S\} \geq [m/3] \wedge$ card $\{k \mid r^k \gtrapprox Ma\} < [m/3]$	Me
5	card $\{j \mid r^j \eqsim Mc\} \geq [2m/3]$	Me
6	$a =$ card $\{j \mid r^j \eqsim Me\}$; $b =$ card $\{k \mid r^k \eqsim Ma\}$ $a - b \geq [2m/3] \wedge a \geq b$	Me
7	card $\{j \mid r^j \gtrapprox Ma\} \geq [m/2] \wedge$ card $\{k \mid r^k \lessapprox S\} \geq [m/3]$	Me
8	$a - b \geq [2m/3] \wedge a < b$	Ma
9	card $\{j \mid r^j \gtrapprox Ma\} \geq [m/2] \wedge$ card $\{k \mid r^k \lessapprox S\} < [m/3]$	Ma
10	card $\{j \mid r^j \gtrsim Ma\} \geq [2m/3]$	R

Liniile din tabel nu reprezintă propriu-zis reguli de inferență ci mijlocul de generare al acestora. De exemplu, pentru situația : m=3 și ultima linie din Tabelul 4.1, rezultă :

dacă $\{[r^1 \ Ma) \wedge (r^2 \ Ma)] \vee [(r^1 \ R) \wedge (r^2 \ R)]$

$\vee [(r^1 \ Ma) \wedge (r^2 \ R)] \vee [(r^1 \ R) \wedge (r^2 \ Ma)]$

$\vee$ {*similar pentru* r^2 *si* r^3}

$\vee$ {*similar pentru* r^1 *si* r^3}

atunci $\mu r \quad R$

Similar se pot construi reguli de inferență plecând de la celelalte reguli din tabel, pentru situații concrete. Cu privire la modul de obținere a mecanismului de reguli mai sus prezentat, se impune observația că asocierile din prima coloană precum și valorile lingvistice asociate acestora prin a doua coloană se bazează pe experiența utilizatorului în aprecierea structurii, a importanței ierarhice și a dimensiunilor elementelor de program de pe fiecare ramificație care participă la stabilirea gradului de reutilizare.

Agregarea regulilor se poate realiza folosind diferite tipuri de inferență, de exemplu **MAX - MAX**. Pentru defuzzificare și obținerea gradului de reutilizare propunem metoda centrului de greutate. Pentru a atinge întreaga gamă de valori a gradului de reutilizare, trebuie extins corespunzător domeniul funcțiilor de apartenență.

Al doilea gen de grad de reutilizare propus (r_c) se calculează pentru un set de programe (variante) din cadrul unei clase, desemnate *relevante* în cadrul clasei (inclusiv programul generic). Algoritmul de calcul este similar celui precedent, cu următoarele deosebiri :

- structura de program considerată este cea a programului generic ;
- gradele de reutilizare primare asociate nodurilor se consideră egale cu incidentele statistice relative ale modulelor (adică numărul de utilizări al modulului respectiv, în cadrul variantelor relevante ale clasei, raportat la numărul respectivelor variante) .

Evident, nu se va întâmpla niciodată situația din Fig. 4.3, respectiv, nu vom avea grade de reutilizare primare nule, deoarece cel puțin un program (programul generic) folosește fiecare modul.

Aici, principala dificultate constă în a *desemna* acele programe pe care le-am numit *relevante*. Formularea lasă loc interpretărilor pur subiective pentru această selecție. Pentru a selecta aceste programe, se pot adopta diverse criterii, spre exemplu :

- prin selectarea aplicațiilor cel mai des solicitate din cadrul unei clase (acesta fiind deci un criteriu statistic) ;
- prin selectare aleatorie;

- prin anchetă în rândurile echipei de proiectare, solicitând desemnarea unor programe "*reprezentative*" şi acordarea unor punctaje;
- pe baza varietăţii maxime (programele cele mai diferite între ele, din cadrul unei clase);
- prin combinarea unora dintre aceste criterii, sau a tuturor acestora, pe baza unor ponderi definite în acest scop.

Odată desemnate programele *relevante*, o modificare a mulţimii acestora în sensul extinderii sau micşorării mulţimii acestora, nu trebuie să afecteze sensibil valoarea gradului de reutilizare de clasă calculat pentru acele programe. Motivul este necesitatea unei anumite omogenităţi a proprietăţilor clasei, omogenitate care de fapt impune caracterul de clasă, pentru programele aparţinând clasei. Asupra acestor chestiuni vom reveni în cuprinsul capitolului următor.

Abordarea fuzzy prezentată oferă proiectantului un mijloc de evaluare (inclusiv un algoritm) care înlătură dezavantajele prezentate la cap. 3, pentru gradele de reutilizare folosite curent. Algoritmul prezentat este flexibil, extensibil şi pasibil de îmbunătăţiri uşor de implementat, în timp. Deci permite implementarea uşoară a unor criterii diferite, conform obiectivelor organizaţionale stabilite de la caz la caz.

4.3. Exemplu de calcul pentru gradul de reutilizare de clasă

Să luăm spre evaluare partea de prelucrare a informaţiei din proces pentru un sistem de telemecanică având legătură multipunct [Tiv87-N] (Fig. 4.4). Presupunem că în urma unei analize prealabile s-au dovedit "*relevante*" 4 programe ale clasei :

- programul generic (reprezentat în Fig. 4.4);
- o aplicaţie fără imprimantă;
- o aplicaţie care nu realizează funcţii grafice, integrare şi tratare excepţii la avarii proces;
- o aplicaţie lipsită de tratare excepţii atât la avarii proces cât şi la avarii sistem şi la care telesemnalizările nu trebuie confirmate de operator.

Ţinând cont de toate acestea şi "desfăşurând" diagrama de flux de date din Fig. 4.4 astfel încât să nu existe convergenţe de sus în jos, transformând diagrama în graf arbore[1] (deci

[1] Prin echivalarea taskurilor cu nodurile şi a canalelor de comunicaţie inter-taskuri cu ramurile grafului.

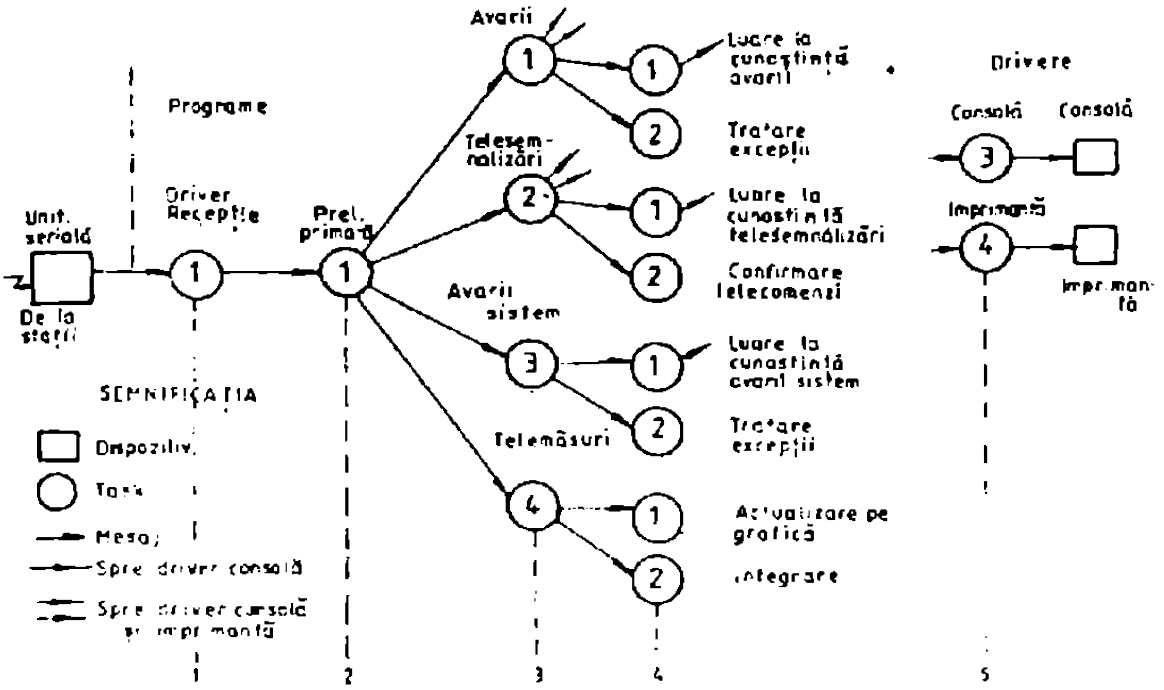


Fig. 4.4. Diagrama de flux de date pentru informația din proces la un sistem de telemecanică (reprezentare simplificată)

trebuind ca driverele de ieșire să apară de mai multe ori, deoarece ele sunt ieșiri pentru mai multe taskuri ; astfel, pentru a evita convergențele, pentru fiecare ieșire de task vom avea câte un nod corespondent, chiar dacă acel nod se repetă), se ajunge la graful din Fig. 4.5.

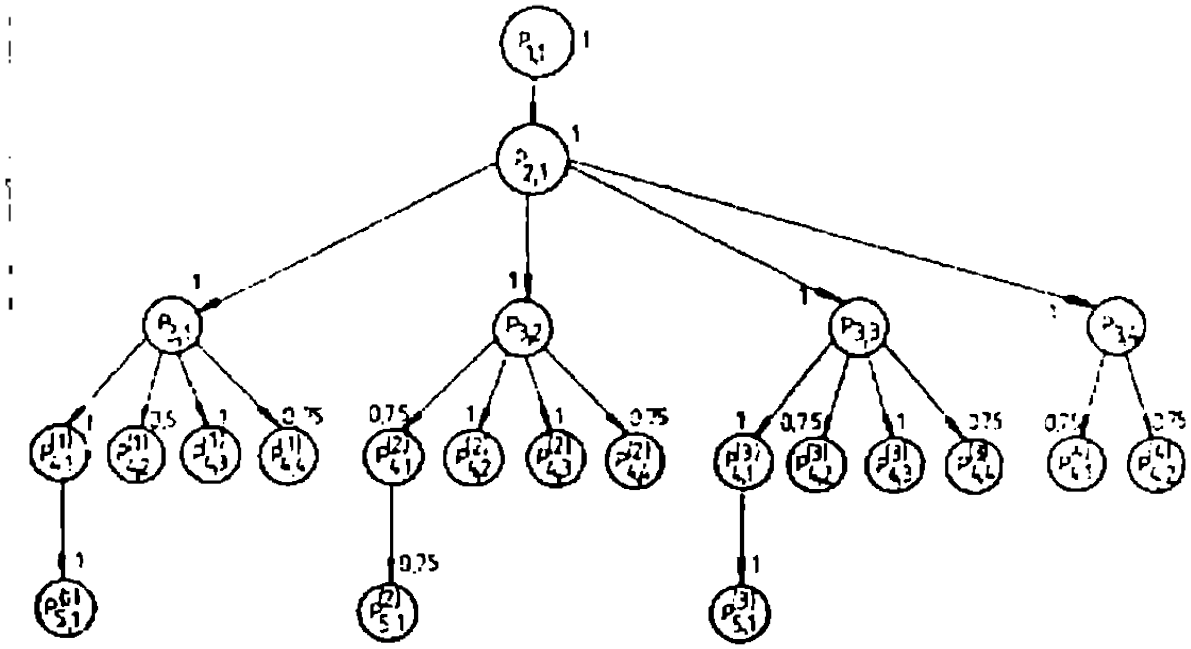


Fig. 4.5. Graful pentru calculul r_c

În prima etapă, grupele de taskuri $P_{4,1}^{(1)} - P_{5,1}^{(1)}$, $P_{4,1}^{(2)} - P_{5,1}^{(2)}$, $P_{4,1}^{(3)} - P_{5,1}^{(3)}$

dobândesc grade de reutilizare de valori 1, 0,75 respectiv 1. În următoarea etapă, se calculează gradele de reutilizare echivalente pentru grupele nodurilor subordonate (3,1), (3,3) şi (3,4).

Prezentăm in extenso doar calculele pentru subgraful de la nodul (3,1) :

$r^{3,1}=1 \Rightarrow \mu_R{}^{3,1}=1$; $r^{4,1}=1 \Rightarrow \mu_R{}^{4,1}=1$; $r^{4,2}=0.5 \Rightarrow \mu_{Me}{}^{4,2}=1$;

$r^{4,3}=1 \Rightarrow \mu_R{}^{4,3}=1$; $r^{4,4}=0.75 \Rightarrow \mu_{Me}{}^{4,4}=0.25$, $\mu_{Ma}{}^{4,4}=0.75$.

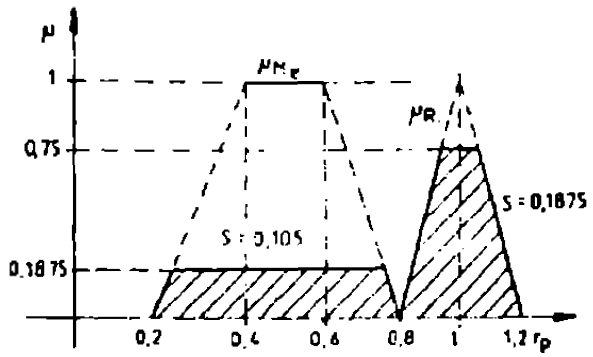


Fig. 4.6. Calculul lui $r_e^{3,1}$ prin metoda centrului de greutate

În fig. 4.5, gradele de reutilizare de modul (gradele de reutilizare primare) sunt trecute alături de noduri. Se activează deci doar regulile de inferenţă care conţin valori **Me**, **Ma**, **R**, dar sunt adevărate doar cele rezultate din liniile 6 şi 10 din Tabelul 4.1. Pentru convenţia $\wedge \rightarrow$ **PROD** şi $\vee \rightarrow$ **DIF.MARG.** obţinem valorile :

$$\mu_{e,R}{}^{3,1} = 0.75 \quad \text{şi}$$

$$\mu_{e,Me}{}^{3,1} = 0.1875$$

(cu indicele "e" pentru "echivalent").

Folosind inferenţa **MAX - MAX** şi metoda centrului de greutate, $r_e^{3,1}$ se obţine calculând centrul de greutate al porţiunii haşurate din Fig. 4.6 :

$r_e^{3,1} = 0.82$.

Valorile $r_e^{3,2} = r_e^{3,3} = 0.95$ şi $r_e^{3,4} = 0.72$ sunt calculate similar.

În etapa următoare, se calculează gradul de reutilizare pentru situaţia din Fig. 4.7. Calculul este similar şi obţinem $r_e^{2,1} = 0.86$.

Mai rămâne de calculat doar gradul de reutilizare final, din $r^{1,1}$ şi $r_e^{2,1}$. Valoarea obţinută este $r_r = 0.83$

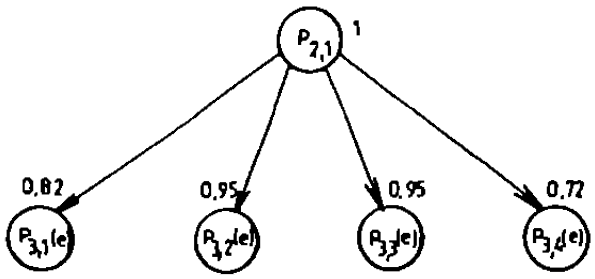


Fig. 4.7. Etapă intermediară de calcul

Rezultatul obţinut are valoare practică prin compararea cu alte valori, obţinute prin aplicarea aceluiaşi algoritm, altor clase de aplicaţii. Dacă valoarea respectivă este mai mică decât alte grade de reutilizare de clasă, vom trage firesc concluzia că acea clasă pentru care a fost calculată valoarea, nu este aşa de "*bună*" din punctul de vedere

al reutilizării, ca şi clasele de comparaţie. Dimpotrivă, dacă valoarea este mai mare, înseamnă că reflectă proprietăţi mai bune din punctul de vedere al reutilizării, decât clasele de comparaţie. Aceasta înseamnă că de fapt efortul de a obţine una sau alta dintre variante (prin proiectare), ţinând cont de limitele predefinite, acceptate ale clasei, este mai mare sau mai mic decât efortul corespunzător, în cadrul claselor de referinţă. Validarea metodei rezultă din folosirea sa în practica proiectării în cadrul claselor de aplicaţii. În capitolul 8 este prezentat un mediu **CASE** care include instrumente de evaluare pentru gradele de reutilizare propuse în acest capitol.

Observaţie importantă *:*

Măsurile software definte mai sus sunt utile pentru programe care pot fi reprezentate prin diagrame de flux de date, precum ***programele multitasking pentru conducere de proces.*** *Acestei categorii de programe îi sunt de cele mai multe ori asociate restricţii de timp de răspuns, adică sunt totodată* ***programe în timp real.*** *Măsurile software propuse* ***nu*** *includ sub nici o formă referiri la aceste* ***restricţii de timp de răspuns*** *(nici nu îşi propun aceasta), de aceea în cele ce urmează (inclusiv în Capitolul 5) ne referim exclusiv la categoria programelor multitasking pentru conducere de proces.*

Exemplul prezentat subliniază aplicabilitatea metodei propuse, pentru sisteme multitasking de conducere de proces.

4.4. Aspecte fuzzy ale apartenenţei programelor la clase de aplicaţii

Întrucât clasele de aplicaţii (în sensul din [Mit87]) nu pot fi riguros definite (v. § 1.4), aplicarea logicii fuzzy pentru stabilirea unor criterii de apartenenţă a unui program la o clasă, pare a fi o abordare "naturală". De aceea, în cuprinsul acestui paragraf, vom încerca să stabilim astfel de criterii, de asemenea, vom stabili în ce măsură criteriile fuzzy stabilite sunt consistente.

Programele (variantele) care compun o clasă de aplicaţii, sunt caracterizate de anumite *specificaţii funcţionale*. O serie de funcţii care caracterizează programele, trebuie să fie în egală măsură atribute esenţiale ale tuturor variantelor unei clase de aplicaţii. Alte funcţii pot lipsi la diverse variante ale clasei, iar o a treia categorie de funcţii suportă o rafinare pe subniveluri, care îngăduie atribuirea de valori fuzzy de apartenenţă la calitatea (funcţia) selectată, pentru variante date. Să notăm cu **V** mulţimea **variantelor** care alcătuiesc o **clasă de aplicaţii** şi cu **v** un element al acestei mulţimi (o **variantă**). În continuare, folosind unele formulări din [Lip94], considerăm definiţiile :

(4.3) Definiţie : *Numim* ***submulţime fuzzy de proprietăţi***, $\tilde{P} \subset V$, *submulţimea de elemente din V (deci,* ***variante****) caracterizate prin proprietatea că variantele din această submulţime respectă o anumită specificaţie funcţională dată*[2] .

Clasa tuturor submulţimilor fuzzy din clasa **V** se notează cu $\tilde{\Im}(\mathbf{V})$:

(4.4) Definiţie : *Se numeşte* ***mulţime fuzzy a lui V****, o parte* $\tilde{\Im} \subset V \times [0,1]$ *astfel încât :*

a) $pr_V(\tilde{\Im}) = V$

b) $(v, w_1) \in \tilde{\Im}$ *şi* $(v, w_2) \in \tilde{\Im} \Rightarrow w_1 = w_2$ [3]

Apartenenţa unui program la o submulţime fuzzy de proprietăţi trebuie să fie cuantificată prin intermediul unei funcţii de apartenenţă. Problema cuantificării fuzzy propriu-zise apare la acele proprietăţi la care se pot stabili subniveluri şi se pot acorda punctaje subiective referitoare la întrunirea condiţiilor de apartenenţă. Stabilirea acestora este strâns corelată cu rafinarea specificaţiilor funcţionale : fiecărei specificaţii îi corespunde o descriere formală Dacă această descriere putem să o grupăm pe acţiuni atomice, atunci prezenţa fiecărei acţiuni în rândul specificaţiei corespunzătoare a unei variante o putem puncta conform unui punctaj arbitrar (conform ponderii acordate fiecărei acţiuni atomice) şi însumând toate aceste punctaje, suma lor trebuie să fie unitară.

Să luăm un exemplu simplu, nuanţat pe situaţia analizată în § 4.3. Dacă izolăm din aplicaţia prezentată, doar taskul de tratare a avariilor de proces, putem spre exemplu stabili următoarele subniveluri pentru funcţia de tratare a avariilor de proces (în dreptul fiecărui subnivel am trecut în paranteză şi punctajul atribuit) :

- verificarea corectitudinii intrării (0,05) ;
- localizarea avariei în lista de avarii (0,1) ;
- construirea unui mesaj de avertizare si expedierea acestui mesaj spre driverul de consolă (0,15) ,
- construirea unui mesaj de avertizare cu destinaţia imprimantă şi expedierea acestui mesaj spre driverul de imprimantă (0,1) ;
- activarea taskului de actualizare pe grafică (0,1) ;

---

[2] Notaţiile sunt cele din [Lup94].

[3] Vom înţelege această definiţie astfel :

a) toate programele din V au proprietăţi (corespunzătoare unor specificaţii funcţionale date) care încadrează programele respective la una din submulţimile fuzzy de proprietăţi (*pr* este notaţia pentru "proiecţie") ,

b) programele caracterizate prin aceleaşi specificaţi funcţionale sunt identice ; aceasta nu este totdeauna adevărat în practică, dar este o ipoteză simplificatoare acceptabilă.

- actualizarea listei de avarii (0,2) ;
- selectarea situațiilor de excepții și eventuala activare a taskului respectiv (0,15) ;
- înscrierea unei cereri de luare la cunoștință în coadă și activarea taskului de luare la cunoștință (0,15).

În această ipoteză, pentru prima din cele 4 situații luate în considerare în § 4.3, punctajul obținut este evident 1, fiind vorba de programul generic. Analiza celei de-a două situații duce la obținerea unui punctaj de 0,9 (deoarece lipsește imprimanta), la a treia situație se obține 0,75 deoarece lipsesc grafica și tratarea excepțiilor, iar ultima situație generează un punctaj de 0,85 (lipsește tratarea excepțiilor). În mod similar se analizează toate funcțiile și se atribuie punctaje, construind astfel toate submulțimile fuzzy de proprietăți.

Rămâne să stabilim acele specificații funcționale care vor defini clasa tuturor submulțimilor fuzzy de proprietăți, $\tilde{\Im}(\mathbf{V})$. Pentru aceasta, luăm în considerare specificațiile funcționale ale **programului generic**. Fiecare specificație funcțională va **induce** o submulțime fuzzy de proprietăți. Astfel, problema aceasta este rezolvată.

Proiectarea pentru reutilizare este un proces **dinamic**. De aceea, în abordarea fuzzy, trebuie ținut cont de aceasta. Proiectarea pentru reutilizare transformă un program al unei clase, prin reutilizarea totală a unor componente, prin reutilizarea și modificarea altora, respectiv prin adăugarea unor componente noi, într-un alt program care să corespundă specificațiilor funcționale. Deci, caracterizarea unui program prin prisma unei anumite calități funcționale, suferă în cursul proiectării, o modificare în timp. Fie $\{\mathbf{V(t)}\}_{t \in T}$ o familie de variante (o clasă de aplicații - în sensul din [Mit87]) și $\{\tilde{P}(\mathbf{t})\}_{t \in T}$ submulțimi fuzzy $\tilde{P}(\mathbf{t}) \subseteq \mathbf{V(t)}$. Numim **submulțime fuzzy de proprietăți, dependentă de timp** familia $\{\tilde{P}(\mathbf{t})\}_{t \in T}$ unde funcțiile de apartenență respective sunt :

$$\tilde{P}(\mathbf{t})(.),\mathbf{t}' : \mathbf{V(t)} \rightarrow [0,1],\ t \in T.$$

Sensul funcției de apartenență $\tilde{P}(\mathbf{t})(.),\mathbf{t}'$, cu $t \leq t'$ este următorul : valorile de apartenență ale programelor la anumite categorii de proprietăți ale clasei de aplicații $\{\mathbf{V(t)}\}_{t \in T}$ suferă modificări la momentul t', în raport cu criteriile impuse (anterior) la momentul t. Altfel spus, prin proiectare, anumite funcții ale programului își modifică mai mult sau mai puțin, caracteristicile, ceea ce determină modificarea valorilor determinate de funcțiile de apartenență de mai sus.

Pentru a reuși definirea unei relații de încadrare a unor programe într-o clasă de aplicații, trebuie găsită o cuantificare potrivită a diferențierilor programelor între ele, în cadrul sau în afara unei clase. O astfel de măsură posibil de utilizat este :

(4.5) Definiţie *se numeşte* ***distanţa Hamming*** *definită pe clasa tuturor submulţimilor fuzzy de proprietăţi ale lui V, aplicaţia*

$$d : \mathfrak{I}(V) \times \mathfrak{I}(V) \rightarrow R_+$$

de forma dată de relaţia

$$d(P,Q) = \sum_{i=1}^{n} \mu_P(v_i) - \mu_Q(v_i)$$

unde $P, Q \in \mathfrak{I}(V)$ *sunt submulţimi fuzzy de proprietăţi ale clasei* ***V****, iar* ***n*** *este numărul de elemente ale clasei,* ***card(V) = n.***

În definiţia de mai sus, $\mu_P(.)$, respectiv, $\mu_Q(.)$ reprezintă valorile fuzzy date de apartenenţa unui program anume la o submulţime fuzzy de proprietăţi $\tilde{P}$, respectiv $\tilde{Q}$.

Distanţa Hamming oferă informaţii referitoare la "*distanţa*" dintre variantele unei clase, caracterizate din punctul de vedere al satisfacerii de către programele clasei, a două dintre specificaţiile funcţionale ale programului generic. Această măsură este deci un indicator global pentru proprietăţile programelor unei clase, dar este relativă doar la două proprietăţi selectate, din totalul de proprietăţi ale programului generic.

Pentru ca distanţa Hamming să nu depindă de numărul de elemente ale clasei (respectiv, de numărul de variante), preferăm folosirea variantei normalizate, respectiv :

(4.6) Definiţie ***Distanţa Hamming relativă*** *pe clasa tuturor submulţimilor fuzzy de proprietăţi ale lui V este dată prin relaţia*

$$s(P,Q) = \frac{d(P,Q)}{n} = \frac{1}{n} \sum_{i=1}^{n} \mu_P(v_i) - \mu_Q(v_i)$$

Se demonstrează [Lip94] că $0 \leq d(\tilde{P}, \tilde{Q}) \leq 1$.

Cu cât distanţa Hamming este mai mică, cu atât elementele clasei **V** "seamănă" mai mult între ele, deci clasa este caracterizată de **o coeziune** mai puternică. Putem, în acest sens, enunţa un *criteriu de coeziune* al unei clase :

(4.7) Criteriu : *O clasă dată V este* ***ε - compactă*** *sau este caracterizată de* ***coeziune tare de valoare ε*** *dacă toate distanţele Hamming relative definite pe clasa tuturor submulţimilor fuzzy de proprietăţi ale lui V sunt mai mici decât valoarea dată ε :*

$$0 \le \delta(\tilde{P}, \tilde{Q}) \le \varepsilon \quad (\forall)\tilde{P}, \tilde{Q} \subset \mathbf{V}$$

La o raportare strict subiectivă, se poate spune că o clasă de aplicații care nu este caracterizată de coeziune tare de valoare ε prestabilită, de fapt nu este (în sensul subiectiv - relativ la valoarea prestabilită ε) o clasă de aplicații ! În sensul acestei raportări, putem afirma despre Criteriul (4.7) că este o condiție necesară (dar nu și suficientă) pentru a putea afirma despre o mulțime de programe că este o clasă de aplicații. Acest criteriu este relativ, subiectiv, întrucât este legat de o valoare arbitrară (ε). Deoarece pentru verificarea condițiilor criteriului trebuie sa avem la dispoziție toate variantele, criteriul se dovedește a fi de genul **a posteriori**. Prin convenție, atribuim în continuare denumirea de **clasă de aplicații** doar unei clase **ε - compacte**.

*Extrapolarea și adaptarea unor noțiuni din **matematica mulțimilor fuzzy**, la specificul proiectării sofware pentru reutilizare, constituie, în concluzie, preocuparea paragrafului acesta. Prezentarea unor definiții și noțiuni adaptate domeniului abordat, precum submulțime fuzzy de proprietăți, mulțime fuzzy, modul cum specificațiile funcționale cărora li se acordă punctaje induc submulțimi fuzzy de proprietăți, dinamica submulțimilor fuzzy de proprietăți, folosirea distanței Hamming ca și criteriu de coeziune a unei clase de aplicații, constituie tot atâtea contribuții ale acestui paragraf, la edificarea unei teorii coerente privind **abordarea fuzzy a proiectării software pentru reutilizare**.*

4.5. Matrice fuzzy în proiectarea pentru reutilizare în cadrul claselor de aplicații

Fie o clasă de aplicații **V** având **n** programe $v_i \in V$ (variante). Dacă luăm în considerare gradele de reutilizare de program, relative la grupe de câte două programe, cu valorile respective putem alcătui o matrice care să aibă ca elemente chiar aceste grade de reutilizare :

$$R = \| r_{i,j} \|$$

cu i,j = 1,...,n și

$r_{i,j}$ - gradul de reutilizare de program, al programului j în raport cu programul i (*grad de reutilizare de la un program la altul*)

Această matrice este **matrice fuzzy** [Lip94] întrucât $0 \le r_{i,j} \le 1$.

Elementele matricei sunt valori pe care le-am numit **grade de reutilizare de la un**

program la altul. Definim şi calculăm aceste valori la fel ca *gradul de reutilizare de program*, dar relativ la un program oarecare şi nu la *programul generic*. Astfel, r_{ij} înseamnă gradul de reutilizare al programului j, calculat relativ la programul **i**, conform aceluiaşi algoritm ca şi r_{pj}. Altfel spus, algoritmul de calcul prezentat în § 4.2 se aplică prin compararea diagramei de flux de date a programului **j**, relativ la diagrama programului i.

Matricea fuzzy **R** este o reprezentare a unei **relaţii fuzzy** $\tilde{\Re}$ ($\tilde{\Re} \subset \mathbf{V} \times \mathbf{V}$). Numim această relaţie fuzzy, **relaţie de reutilizare**.

(4.8) Proprietate : ***Domeniul, codomeniul şi înălţimea***[4] *relaţiei fuzzy* $\tilde{\Re}$ *sunt egale cu 1.*

Demonstraţia este imediată şi decurge din constatarea că gradul de reutilizare al unui program în raport cu sine însuşi este 1. De aceea, elementele diagonalei matricei fuzzy sunt unitare.

Conform teoremei de descompunere [Lip94], o relaţie fuzzy $\tilde{\Re}$ se poate descompune pe **niveluri**. Această teoremă nu are o aplicabilitate imediată în teza de faţă, dar demersuri teoretice viitoare pot duce la posibile consecinţe interesante. De aceea vom prezenta pe scurt teorema de descompunere, precum şi o succintă interpretare practică.

(4.9) Definiţie : *Fie* $\alpha \in [0,1]$ *şi* $\tilde{\Re}$ *o* ***relaţie de reutilizare*** *în* $V \times V$. *Atunci mulţimea uzuală*

$$\Re_\alpha = \{ (v_i, v_j) \in V \times V \mid \tilde{\Re}(v_i, v_j) \geq \alpha \}$$

se numeşte ***nivelul de ordin*** α ***al relaţiei*** $\tilde{\Re}$.

(4.10) Teorema de descompunere : *Fie* $\tilde{\Re}$ *o relaţie de reutilizare în* $V \times V$ *şi* $\{ \Re_\alpha \}_\alpha$ *mulţimea nivelurile sale (indexată după nivelul de cuantificare arbitrar* α*). Atunci avem :*

$$\tilde{\Re} = \bigcup_{\alpha \in [0,1]} \alpha \Re_\alpha$$

Ce înseamnă în practica proiectării această teoremă ? După această teoremă de descompunere, o clasă de aplicaţii se poate restructura într-o reuniune de clase având variante cu grade de reutilizare reciproce, respectiv mai mari decât valorile prestabilite corespunzătoare nivelelor de cuantificare existente în clasa iniţială. O posibilă concluzie imediată, este aceea că în cadrul claselor în care s-a descompus clasa dată, suma tuturor distanţelor Hamming dintre variante este mai mare decât în clasa dată (demonstraţia este imediată, întrucât clasele rezultate sunt compuse din variante având grade de reutilizare reciproce, mai multe nule decât în clasa

[4]) în sensul definiţiilor din [Lip94]

dată, deci unii din termenii sumelor din definiția (4.5) sunt mai mari). Altfel spus, clasele rezultate sunt mai puțin compacte decât clasa dată. De aceea, din punct de vedere practic, mai utilă ar fi operația inversă, de **compunere**, care ar determina compactarea unor clase. Demersul teoretic în această direcție rămâne să fie făcut, prin continuarea cercetărilor inițiate prin lucrarea de față.

Să încercăm să investigăm acum dacă *relația de reutilizare* propusă induce un anumit gen de relații de ordine. Pentru aceasta, să vedem întâi ce condiții trebuie să satisfacă o relație fuzzy pentru a fi o relație de **ordine fuzzy** [Lip94]:

(4.11) Definiție : *O relație fuzzy* $\tilde{\Re}$ *în* $V \times V$ *este* ***relație de ordine fuzzy*** *dacă este* ***reflexivă, antisimetrică și tranzitivă***.

Dacă o relație fuzzy este doar reflexivă și tranzitivă, este relație de *preordine*. Despre **relația de reutilizare** definită mai sus nu se poate afirma că este *relație de preordine fuzzy*. Vom efectua câteva investigații asupra acestei afirmații, demonstrând existența câtorva

(4.12) Proprietăți ***ale relației fuzzy*** $\tilde{\Re}$ ***în*** $V \times V$:

a) *Relația este* ***reflexivă***, întrucât gradul de reutilizare de la un program la el însuși, $\tilde{\Re}(v,v) = 1, (\forall)\ v \in V$. De altfel, aceasta este valabil și pentru oricare altă definire a gradului de reutilizare de program în afara celei folosite în capitolul de față.

b) *Relația nu este* ***simetrică***, deoarece gradul de reutilizare de la un program la altul realizează o corespondență numerică pentru gradul de suprapunere al grafului unui program peste al celuilalt, iar criteriile de raportare iau în considerare totdeauna o structură de comparație, aceea a programului de referință. În consecință, nu același rezultat se obține dacă se inversează rolurile a două programe selectate arbitrar, din clasa luată în considerare, adică, în general, $\tilde{\Re}(v_1,v_2) \neq \tilde{\Re}(v_2,v_1)$. Același lucru este valabil și pentru gradul de reutilizare definit de Browne [Bro90], unde această proprietate este și mai evidentă (din simplul motiv că în general două programe selectate arbitrar nu au același număr de linii-sursă, deci numitoarele celor două grade de reutilizare sunt diferite, în condițiile în care numărătoarele sunt identice.

c) Relația nu este în general nici ***antisimetrică***, deoarece nu pentru oricare două programe diferite, arbitrare, din cadrul clasei în studiu, gradele de reutilizare de la un program la altul sunt diferite : pot exista (dar nu **necesar**) cazuri de programe care să determine grade de reutilizare de la un program la altul, egale. Să arătăm că putem construi variante care să se încadreze în

acest caz. Fie un program oarecare v. Să adăugăm la un nod (reprezentând spre exemplu un task, aşa ca în cazul de studiu din § 4.3), un task subordonat. Astfel am obţinut o variantă derivată, v'. Să adăugăm acum la acelaşi nod al programului v, un alt task, diferit de cel adăugat ulterior. Dacă aplicăm acum algoritmul de calcul prezentat în § 4.2, rezultă acelaşi grad de reutilizare de la un program la altul, indiferent de programul luat ca referinţă, întrucât fiecare program diferă de celălalt doar prin taskurile suplimentare menţionate mai sus. Prin algoritmul acesta simplu de construcţie, am demonstrat deci afirmaţia de mai sus.

Despre **tranzitivitate** nu se poate afirma nimic, întrucât aceasta ar însemna să demonstrăm că $\tilde{\mathfrak{R}} \circ \tilde{\mathfrak{R}} \subseteq \tilde{\mathfrak{R}}$; după definiţia compunerii relaţiilor fuzzy din [Lip94], aceasta înseamnă că

$\tilde{\mathfrak{R}} \circ = \tilde{\mathfrak{R}}$ **sup** $_{v \in V}$ **min** ($\tilde{\mathfrak{R}}$ **(u,v),** $\tilde{\mathfrak{R}}$ **(v,w))** $\leq \tilde{\mathfrak{R}}$ **(u,w)**

Nu avem argumente pentru a demonstra această inegalitate.

Deoarece nu putem spune despre o **relaţie de reutilizare** că este de **ordine** (întrucât nu este **antisimetrică**), nu putem stabili criterii clare de ordonare ca strategie de proiectare în cadrul unei clase.

Vom defini în continuare o *relaţie fuzzy de reutilizare* care să fie *tranzitivă*. Fie mărimea **normă reciprocă de reutilizare** $r^n_{i,j}$, complementara faţă de 1 a modulului diferenţei dintre gradele de reutilizare de program a două programe ale clasei, v_i şi v_j :

$$r^n_{i,j} = 1 - | r_{p,i} - r_{p,j} |$$

Această mărime măsoară gradul în care două programe se abat în aceeaşi măsură de la programul generic, deci potenţial refolosesc aceleaşi module din programul generic (de aceea, denumirea de normă reciprocă de reutilizare). Astfel, dacă acest număr este 1, cele două programe diferă la fel faţă de programul generic (în sensul definit prin gradul de reutilizare de program - deci, programele pot să difere în acelaşi grad dar să nu fie identice). Dimpotrivă, dacă acest număr este 0, înseamnă că cele două programe nu au elemente comune (deşi această situaţie este pur ipotetică, căci în această situaţie programele de fapt nu ar aparţine aceleiaşi clase).

Fie o *relaţie fuzzy de reutilizare* definită de *matricea fuzzy*

$$R^n = \| r^n_{i,j} \|$$

cu i,j = 1,...,n şi

$r^n_{i,j}$ - norma reciprocă de reutilizare a programelor i şi j

Această matrice este de asemenea **matrice fuzzy** [Lip94] întrucât $\mathbf{0 \leq r^n_{i,j} \leq 1}$ (deoarece cei doi termeni din modulul normei reciproce de utilizare sunt, ambii, subunitari, deci modulul

diferenței lor, cu atât mai mult, iar ca rezultat, complementara față de 1, de asemenea).

Numim matricea $\mathbf{R_n}$, **matricea normă fuzzy**, iar *relația fuzzy de reutilizare* astfel indusă o notăm cu $\tilde{\Re}_n$ ($\tilde{\Re}_n \subset \mathbf{V \times V}$).

Înainte de a investiga proprietățile noii relații astfel definite, să definim o *relație de similitudine* :

(4.13) Definiție : *O* ***relație fuzzy*** $\tilde{\Re}$ *în* ***V*** *este* ***relație de similitudine fuzzy*** *dacă este* ***reflexivă, simetrică și tranzitivă***.

În aceste condiții, să enunțăm următoarea :

(4.14) Teoremă : *Relația fuzzy de reutilizare definită de* ***matricea normă fuzzy*** R_n *:*

$\tilde{\Re}_n$ ($\tilde{\Re}_n \subset V \times V$) *este o* ***relație de similitudine***.

Demonstrația decurge din Definiția (4.13) și din demonstrarea proprietăților de simetrie, reflexivitate și tranzitivitate ale normei reciproce de reutilizare.

Să demonstrăm că relația fuzzy $\tilde{\Re}_n$ este **relație de similitudine** .

a) relația este **reflexivă** deoarece $\tilde{\Re}_n(v, v) = 1$ (modulul din relația de definire a *normei reciproce de reutilizare* este în acest caz nul,) ;

b) relația este evident **simetrică** (decurgând din simetria modulului care definește *norma reciprocă de reutilizare*) :

c) relația este de asemenea **tranzitivă** deoarece

$$\tilde{\Re}_n \circ \tilde{\Re}_n = \sup_{v \in V} \min (\tilde{\Re}_n(u,v), \tilde{\Re}_n(v,w)) \leq \tilde{\Re}_n(u,w)$$

Să demonstrăm această afirmație :

Termenii relației de mai sus sunt :

$$\tilde{\Re}_n(u,v) = 1 - | r_p(u) - r_p(v) | \; ; \; \tilde{\Re}_n(v,w) = 1 - | r_p(v) - r_p(w) | \; ;$$
$$\tilde{\Re}_n(u,w) = 1 - | r_p(u) - r_p(w) |$$

° Să efectuăm o ordonare arbitrară (pe care o putem realiza întrucât pe de o parte indexarea variantelor este arbitrară, iar pe de altă parte modulul din relaţia de definire a normei permite permutări suplimentare) :

$$0 < r_p(w) < r_p(u) < 1$$

° Să presupunem în continuare că $r_p(v)$ pentru care este verificată relaţia de tranzitivitate fuzzy are o valoare mai mare decât celelalte două valori :

$$0 < r_p(w) < r_p(u) < r_p(v) < 1$$

Presupunerea nu influenţează substanţa problemei deoarece datorită existenţei modulului în termenii relaţiei de tranzitivitate, se obţine efectiv acelaşi rezultat dacă valoarea respectivă este dispusă simetric faţă de una din valorile fixate, relativ la valoarea considerată prin ipoteza de mai sus.

° Să demonstrăm prin **reducere la absurd**, că relaţia de mai sus este adevărată. Să presupunem întâi că pentru valoarea selectată prin operatorul sup (.) din stânga relaţiei de tranzitivitate, va fi valabilă relaţia contrară celei din respectiva relaţie, iar din argumentul operatorului **min** (.) va fi selectat al doilea termen :

$$1 - | r_p(v) - r_p(w) | > 1 - | r_p(u) - r_p(w) |$$

Ţinând cont de ordonarea propusă, rezultă că :

$r_p(v) - r_p(w) < r_p(u) - r_p(w)$ $<=>$ $r_p(v) < r_p(w)$ **ceea ce contrazice ipoteza**.

° Întotdeauna putem să presupunem că al doilea termen va fi cel selectat de operatorul **min** (.), deoarece putem inversa rolul celor doi termeni fixaţi, din motive de simetrie.

Deci relaţia de tranzitivitate este **demonstrată**.

În concluzie, *relaţia fuzzy de reutilizare* definită de *matricea normă fuzzy* este o relaţie de *similitudine*. Aceasta este o consecinţă "naturală", întrucât programele unei clase sunt

echivalente[5] şi din punct de vedere empiric : pot fi obţinute unul din celălalt, prin transformări (relativ) uşor accesibile unui proiectant sofware. Rezultatul teoretic obţinut nu face decât să confirme ceea ce un practician software cunoaşte din experienţă. Posibile aplicaţii practice ale clasificărilor prin relaţii de *similitudine* sunt în domeniul **regăsirii** componentelor reutilizabile grupate în baze software (v. § 1.5), deci în domeniul construirii aşa numitelor **browsere** pentru baze software. În acest sens, citãm o definiţie din lucrarea [Lip94] :

(4.15). Definiţie : *Se numeşte **clasă de similitudine (clasă de echivalenţă fuzzy)** asociată lui* $v \in V$, *notată* $\hat{v}$, *mulţimea fuzzy cu funcţia de apartenenţă* $\aleph_{\hat{v}}(w) = \tilde{\Re}_n(v, w)$, $w \in V$.

Deci o clasă de aplicaţii **V** este compusă din *clase de similitudine*. Acestea nu sunt neapărat disjuncte deoarece *matricea normă fuzzy* este simetrică. Posibilele aplicaţii practice ale împărţirii claselor de aplicaţii în subdiviziuni numite clase de echivalenţă sunt găsibile prin investigaţii teoretice în profunzimea domeniului relaţiilor fuzzy. *Corespondenţa practică* a acestei împărţiri o constituie delimitarea unor submulţimi de programe caracterizate prin aceea că sunt obţinute prin transformări direct dintr-un program **v**, dat, sau pot fi direct transformate în programul **v** prin proiectare pentru reutilizare.

*În concluzie, folosirea **matricelor fuzzy** pentru reprezentarea relaţiilor de reutilizare în cadrul unei clase de aplicaţii, constituie un instrument flexibil şi util, atât din punct de vedere teoretic, cât şi practic. Câteva consideraţii practice interesante rezultă din aplicarea teoremei de descompunere. Deoarece o relaţie fuzzy nu este în general o relaţie de ordine (sau preordine), nu putem ordona componentele unei clase, ceea ce se regăseşte şi în practică. În schimb, putem găsi relaţii fuzzy (în acest paragraf este propus un exemplu) de similitudine pe clasă de aplicaţii, cu posibile aplicaţii practice în domeniul **browserelor** pentru bazele software cu componente reutilizabile.*

4.6. Concluzii

Analiza câtorva definiri ale gradului de reutilizare a scos în evidenţă *caracterul subiectiv, empiric al abordărilor actuale* ([Mil87][Wey90][Bro90][Ram88]). Gradul de reutilizare în accepţiunea cea mai des citată rezultă din compunerea a două grade de reutilizare: de cod şi de structură [Bro90]. **Metoda propusă de autor** *combină* caracterul *subiectiv* al reprezentării gradului de reutilizare, cu *înglobarea în aceeaşi reprezentare a caracteristicilor de*

[5] Din punctul de vedere al teoriei fuzzy, *echivalenţă* şi *similitudine* sunt termeni identici [Lip94].

reutilizare de cod și de structură, dar într-un mod mai "rafinat" și mai *nuanțat* decât în anterioarele reprezentări. Prin definirea unui **grad de reutilizare de program** și a unui **grad de reutilizare de clasă**, noțiunea dobândește un plus de consistență. În plus, gradul de reutilizare de clasă are o semnificație interesantă, de "notă" acordată calității de *adaptabilitate pentru întreaga clasă* respectivă. Definirile propuse *înlătură* o bună parte din *dezavantajele menționate* la concluziile Cap. 2. Noile măsuri propuse sunt *utile* proiectantului software prin aceea că îi permit accesul la informații care cuantifică nivelul de reutilizare al programelor pe care le proiectează, relativ la alte programe, sau nivelul de reutilizare în cadrul clasei de programe, ca măsură globală. Cuantificarea are loc într-un mod care înglobează experiența mai multor proiectanți, constituind un algoritm "expert" (în sensul *sistemelor expert*).

Pe baza celor prezentate, se poate realiza un *program* cu interfață grafică adecvată, pentru calculul automat al celor două grade de reutilizare propuse. Se poate îngloba o opțiune de calcul de grad de reutilizare în medii de dezvoltare cu posibilități de reprezentare a structurii programului sub forma unui graf. Elaborarea specificațiilor unei scule **CASE** cât mai utile în dezvoltarea programelor din cadrul unei clase trebuie să țină cont de cerințele de *asigurare a unor evaluări de metrică software* pentru *aprecierea calității proiectării pentru* ***reutilizare***. Definirile prezentate la § 4.1 sunt potrivite pentru a fi implementate în astfel de medii. Un mare **avantaj** al acestora îl constituie *flexibilitatea dată de tehnicile fuzzy*, respectiv de legea de apartenență, de regulile de inferență (Tab. 4.1), de regula de agregare și cea de defuzzificare. *Experiența unor specialiști* poate fi *ușor înglobată* în sistem, prin adăugarea de linii la Tab. 4.1. De asemenea, prin consultarea multor specialiști pentru definirea regulilor enunțate, se atenuează caracterul subiectiv al regulilor, transformând de fapt regula de evaluare într-un gen de "*sistem expert*", în sensul conferirii unei "autorități" profesionale pronunțate pentru algoritmul implementat în instrumentul **CASE** amintit.

Pentru generarea regulilor fuzzy, se mai poate apela la diverse tehnici, spre exemplu, la sugestiile din [Yua95] și [Dom95], care se referă la utilizarea **algoritmilor genetici**. Atât înglobarea experienței unor specialiști în setul de reguli fuzzy, cât și aplicarea unor tehnici specifice algoritmilor genetici pentru obținerea regulilor fuzzy, constituie tot atâtea **direcții posibile de dezvoltare**.

Metoda de definire poate fi **extinsă** prin lărgirea setului de reguli de inferență, astfel încât acestea să țină cont de dezideratele de alegere ale valorilor predicatelor concluziei (din Tab. 4.1) luând în considerare și alte aspecte de refolosire decât cele prezentate în § 4.1 : structuri de date, funcții, nivel ierarhic, etc. Astfel pot rezulta indicatori cu un mai mare grad de rafinare, însă definirea regulilor aferente tehnicilor fuzzy se complică substanțial. Oricum, aplicarea acestor reguli în calculul gradului de reutilizare este aproape imposibil de realizat în afara unor scule **CASE** eficiente.

Un câmp larg de acțiune pentru investigații teoretice și aplicații practice ale rezultatelor acestor investigații, este deschis prin **considerarea reutilizării printr-o perspectivă fuzzy**.

Folosind aparatul matematic specific acestei abordări, principalele concepte ale reutilizării capătă contur şi sunt integrate într-o **teorie unitară**, cu **aplicaţii practice**. Astfel, **utilizarea specificaţiilor funcţionale** având atribuite punctaje după o metodologie prezentată în § 4.4 pentru a defini **submulţimi fuzzy de proprietăţi**, **folosirea distanţei Hamming** ca şi *criteriu de validare a coeziunii unei clase*, folosirea gradului de reutilizare definit în § 4.1 pentru generarea unor **matrice fuzzy** pe clasa de aplicaţii, care la rândul lor induc pe aceasta o **relaţie de echivalenţă fuzzy**, constituie câteva rezultate ale abordărilor § 4.4 - 4.5. *Teoremele* şi *proprietăţile* la care a fost necesară o **demonstraţie**, au fost demonstrate (teoremele sau proprietăţile (4.8), (4.12) şi (4.14)).

Demersul teoretic poate fi continuat mult în profunzime, atât din punctul de vedere al completării unei teorii cât mai închegate şi consistente, cât mai ales din cel al unor consecinţe practice, spre exemplu ale *Teoremei de descompunere* (4.10) sau ale împărţirii în *clase de similitudine*, conform Definiţiei (4.15).

CAPITOLUL 5.
O FORMALIZARE A PROIECTĂRII VARIANTELOR

Capitolul de față s-a născut cu gândul de a suplini o lipsă majoră a abordărilor teoretice ale proiectării software în general, iar în particular, ale proiectarii pentru reutilizare. Există multe modele pentru a descrie programele de-a lungul diverselor etape ale ciclului lor de viață. Acestea sunt în general grupate în abordări unitare, în cadrul cărora sunt folosite consecvent seturi bine definite de convenții și concepte de bază, convenții grafice de reprezentare și pachete de instrumente software (a se vedea în acest sens spre exemplu [Nie95] unde sunt prezentate multe asemenea metode). În literatură nu au fost însă prezentate modele formale cuprinzătoare și riguroase pentru procesul de proiectare propriu-zisă, care să genereze și o teorie corespunzătoare, cu consecințe atât în plan teoretic cât și practic, cu atât mai puțin un formalism complet pentru acest domeniu al realității, care ar trebui să poată fi modelat ca oricare alt fragment de lume reală. Abordările existente (incomplete) au fost sintetizate în Cap. 1, § 1.10 (dar se pot studia în acest sens și sugestiile din articolul [Mat84] respectiv lucrarea [She95]).

Acest capitol prezintă un ***formalism original*** *pentru abordarea sistemică a procesului de proiectare software pentru reutilizare. În cele ce urmează, ne referim strict la proiectarea care realizează trecerea, în cadrul unei clase de aplicații, de la o variantă la alta, operație pe care o numim* ***proiectare în clasă****. Totodată, în cadrul capitolului se prezintă un formalism pentru a descrie un* ***sistem de proiectare în clasă (SPC)****, câteva propuneri de măsuri software ce decurg din acest formalism și o propunere de structurare a clasei de aplicații.*

Noțiunile de *clasă* și *variantă* sunt considerate la nivelul acestui capitol ca noțiuni primare (fiind prea vag definite pentru a putea fi încadrate formal). Pentru aceste noțiuni, în [Mit87] au fost date definiții (sau mai exact descrieri intuitive) în următoarea accepțiune : **clasa** este o mulțime de programe numite **variante**, caracterizate prin aceea că orice variantă poate fi obținută prin modificări (a căror limită este stabilită subiectiv) ale unei variante speciale aparținând aceleiași clase, numite **program generic**.

5.1. Formalismul SPC

Fie C^{α} o clasă de aplicaţii şi v_i^{α}, $i \in I^{\alpha}$ mulţimea elementelor sale, adică [Sto95-S][Sto95-A]:

$$C^{\alpha} = \bigcup_{i \in I^{\alpha}} v_i^{\alpha} \tag{5.1}$$

α este indexul specificator al clasei (în principiu, numele clasei), iar I^{α} este mulţimea indicilor elementelor (*variantelor*) $\mathbf{v_i}^{\alpha}$ ale clasei numită şi *extensia clasei*. Pentru simplitatea scrierii notăm $C^{\alpha} = (\alpha)$.

(5.1) Observaţii :

a) *Trebuie să subliniem că în general extensia clasei I^{α} are un caracter dinamic, fiind legată de evoluţia temporală a clasei. Noi membri ai clasei (noi variante) apar în măsura în care necesitatea (piaţa) dictează apariţia acestora,astfel că de cele mai multe ori clasele se prezintă sub forma unui set de variante cu extindere în general necunoscută, dar care trebuie limitată impunând criterii identice cu cele care definesc clasa, spre exemplu criterii legate de volumul şi natura informaţiei din proces tratate.* Numim aceste clase, cu o dinamică conjuncturală a extensiei, **clase deschise**.

b) *De o mare importanţă pentru evidenţierea caracterului dinamic al proiectării, este ordonarea enumerării variantelor adică stabilirea corespondenţei dintre variantele clasei şi elementele mulţimii I^{α}. Propunem două principii de ordonare :*

- *<u>-ordonarea după principiul efortului minim de transformare pentru trecerea de la o variantă la alta</u> (v.cap.3).*

  *În acest caz, ordonarea se poate realiza numai **a posteriori**, deci după ce există toate variantele : plecând de la programul generic $\mathbf{v_0}^{\alpha}$, următorul program $\mathbf{v_1}^{\alpha}$ este cel obţinut prin transformări minime din $\mathbf{v_0}^{\alpha}$, cel ce urmează, $\mathbf{v_2}^{\alpha}$, este cel obţinut prin transformări minime din $\mathbf{v_1}^{\alpha}$,ş.a.m.d. Evident, în acest caz, ordonarea nu va respecta înşiruirea "naturală" a variantelor şi la apariţia oricărei noi variante este posibilă şi o modificare, prin operarea de trasformări de stare care schimbă ordinea variantelor. Aceasta se întâmplă deoarece ordinea necesară a variantelor nu este şi neapărat ordinea de efort minim.*

  <u>-ordonarea după principiul înşiruirii "naturale" cronologice a proiectării în cadrul</u>

clasei, corespunzător ordinii în care piața solicită variantele.

În cadrul unei clase, o intervenție a proiectantului, $\mathbf{u}_i^{\alpha}$, reprezintă o acțiune a acestuia care, exercitată asupra unei variante $v_l^{\alpha} \in C^{\alpha}$ conduce la o altă variantă $v_m^{\alpha} \in C^{\alpha}$. În general notația u_i^{α} semnifică o *intervenție* care este de fapt o reuniune de acțiuni elementare ale proiectantului :

$$\mathbf{u}_i^{\alpha} \quad \bigcup_{j \in I_i^{\alpha} \subset I^{\alpha}} u_{i,j}^{\alpha,e} \tag{5.2}$$

În această relație, indicele e desemnează intervenții elementare iar submulțimea de indici I_i^{α} este reunirea indicilor acelor intervenții elementare din I^{α} care conduc la trecerea de la o variantă la alta, în sensul de mai sus.

Convenim să notăm cu U^{α} mulțimea intervențiilor proiectantului asupra programului generic v_0^{α} sau asupra variantelor v_i^{α}, $i \in I^{\alpha}$, făcute în vederea obținerii altor variante v_j^{α}, $j \in I^{\alpha}$:

$$\bigcup^{\alpha} = \bigcup_{i \in I^{\alpha}} \mathbf{u}_i^{\alpha} \tag{5.3}$$

Trebuie să subliniem că procesul proiectării software se desfășoară în timp. Afirmația oarecum trivială dobândește în contextul capitolului de față o importanță specială, întrucât dacă dorim să creăm un model dinamic pentru procesul proiectării în sine, va trebui, sub o formă sau alta, să introducem în reprezentările folosite axa timpului.

(5.2) Observație : *În mod natural trecerea de la o variantă v_i^{α}, la alta v_m^{α} (cuantificabilă prin productivitatea proiectării) durează un interval de timp fizic. Din punct de vedere formal, pentru a avea o reprezentare dinamică, este însă suficient să atribuim momente exacte de timp pentru o astfel de trecere, respectiv pentru efectuarea unui "pas" de proiectare de la varianta v_1^{α} - v_1^{α} la varianta v_{1+1}^{α} - v_m^{α}. Două dintre posibilitățile pentru a realiza corespondența dorită pe axa timpului sunt următoarele :*

*-folosirea unui "timp abstract", asociat cu trecerea de la varianta **I** la varianta **i+1**, în accepțiunea de rang al unui șir, în consecință este vorba de întrebuințarea unei discretizări abstracte, care nu asociază fiecărui pas de proiectare o cuantă de timp clar și univoc definită, conform unei eșantionări bine precizate a timpului, ci doar a unui rang marcat pe o axă de timp abstract ; această abordare este generală și permite folosirea oricărui criteriu de ordonare de la **Observația (5.1,b)** ;*

- folosirea unui "timp concret", rezultat din asocierea univocă a unor intervale reale de timp pentru fiecare intervenție a proiectantului, corespunzând câte unei treceri

de la o variantă la alta ; din motive de simplitate, în acest caz se consideră o axă a timpului, discretizată, pe care au fost omise eventualele evenimente considerate nerelevante pentru studiul nostru, evenimente caracterizate prin pauze în activitatea de proiectare (datorate lipsei de cerere sau unor motive obiective) ; evident, în acest caz trebuie să ne raportăm strict la a doua metodă de ordonare de la ***Observaţia (5.1, b)*** *; dezavantajul acestei metode este că nu permite decât un singur mod de parcurgere a clasei, anume acela care corespunde "istoriei" practice, reale, a variantelor clasei.*

În cele ce urmează, vom folosi prima modalitate de introducere a timpului în modelul propus şi anume utilizarea *timpului abstract*. Evident, durata reală a unei intervenţii depinde de "efortul" de proiectare asociat ei. De aceea, pasul de discretizare pentru un model cu timp fizic, real, nu este constant, deci modelele rezultate nu pot lua în considerare acele tehnici de calcul şi proprietăţi din teoria sistemelor, care presupun, ca ipoteză, un timp concret parcurs cu un pas de discretizare constant.

Cu observaţiile de mai sus, procesul proiectării "în sine" poate fi descris de operatorul **d(.)** (*design*) :

$$d^{\alpha} : \mathbf{C}^{\alpha} \times \mathbf{U}^{\alpha} \times \mathbf{T}^{\alpha} \rightarrow \mathbf{C}^{\alpha} \tag{5.4}$$

Aici, $\mathbf{T}^{\alpha}$ reprezintă mulţimea "timp discretizat", considerată conform celor specificate mai sus. Operatorul de proiectare corespunde efectiv unei familii de aplicaţii, fiecare aplicaţie fiind asociată strict unei tranziţii univoc definite, de la o variantă, notată convenţional , v_k^{α} la o variantă v_{k+1}^{α}

Asimilând mulţimea variantelor cu un "spaţiu al stărilor" şi privind proiectarea ca atare ca un proces dinamic, formal, obţinerea unei noi variante printr-o intervenţie (aplicaţie) demarată la momentul $t_k \in T^{\alpha}$ se poate exprima prin :

$$\mathbf{v}_{k+1}^{\alpha} = \mathbf{d}^{\alpha}(\mathbf{v}_k^{\alpha}, \mathbf{u}_k^{\alpha}, \mathbf{t}_k) \tag{5.5}$$

Fie $p^{\alpha}(v_k^{\alpha}, t_k)$ operatorul *performanţe* ale **SPC** :

$$p^{\alpha} : \mathbf{C}^{\alpha} \times \mathrm{T}^{\alpha} \rightarrow \mathbf{P}^{\alpha} \tag{5.6}$$

unde $\mathbf{P}^{\alpha}$ reprezintă spaţiul performanţelor adică al deziderateloe pe care trebuie să le întrunească programul.

În aceste condiţii, procesul proiectării într-o clasă (α) poate fi reprezentat ca în Fig. 5.1. În figură, operatorul de *întârziere* prezintă un aspect formal, de reţinere a unei variante, pentru a

servi la proiectarea următoarei variante. Prin introducerea acestui operator, întreaga schemă devine o reprezentare sistemică *clasică*, cu **intrări** (acțiunile care inițiază proiectarea u_k^{α}), **ieșiri** (programul dorit v_{k-1}^{α}, în sine, respectiv performanțele programului dorit p_{k-1}^{α}), **prelucrări de proiectare** ($d^{\alpha}(.)$ proiectarea propriu-zisă), **prelucrări de evaluare** (funcția de ieșire - operatorul *performanțe*), bucla de **reacție** care conține operatorul de *reținere* a unei variante în vederea modificării ei pentru varianta următoare și o intrare de inițializare care introduce **condițiile inițiale** (*programul generic*).

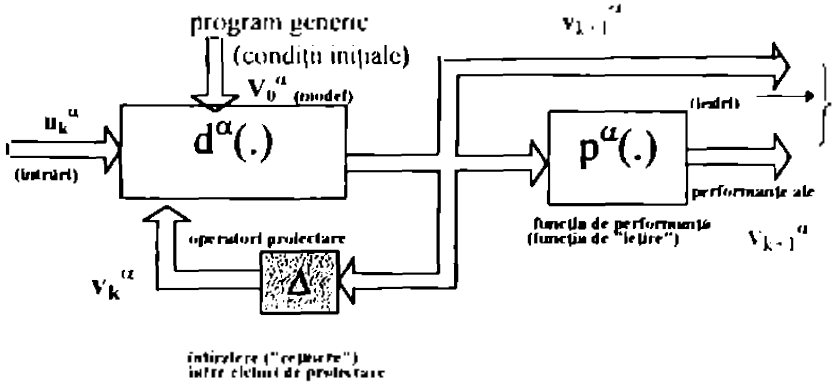


Fig. 5.1. Procesul propriu-zis al proiectării în clasă

Din punct de vedere practic, modelul redat de această schemă este incomplet. El nu surprinde faptul că intervențiile operatorului sunt determinate însești dezideratele de atins, respectiv de calitățile cerute programului de realizat, în ultimă instanță de tema de proiectare și implicit de specificațiile programului (atât specificații funcționale cât și restricții). De aceea, un "model real" trebuie să includă și transformările asociate spațiului *calităților* programelor clasei.

În acest context, trebuie să precizăm în primul rând, o imagine a acestui spațiu, notat cu $\mathcal{F}^{\alpha}$. Fie n numărul de trăsături definitorii ale programelor clasei, notate cu $C^{\alpha,1}$, $C^{\alpha,2}$, ... Potrivit naturii problemei, prin ipoteză, fiecare trăsătură definitorie $C^{\alpha,i}$ poate lua un număr de valori de cuantificare într-o mulțime de cuantificare $I_i^{c,\alpha}$ (ca în metoda de punctare propusă la Definiția 4.4). Evident, mulțimea $I_i^{c,\alpha}$ este mărginită, ceea ce definește în fond chiar limitele clasei (α) în discuție. Astfel, spațiul $\mathcal{F}^{\alpha}$ al *calităților* (sau *funcțiilor*) programelor clasei este un spațiu n-dimensional mărginit, care poate fi reprezentat ca în Fig. 5.2 (pentru o schemă simplificată având n=3). Pe figură au fost marcate limitele pe fiecare axă (*calitate*) în parte. Aceste limite pot delimita și "găuri" în spațiul *calităților*, întrucât există situații în care

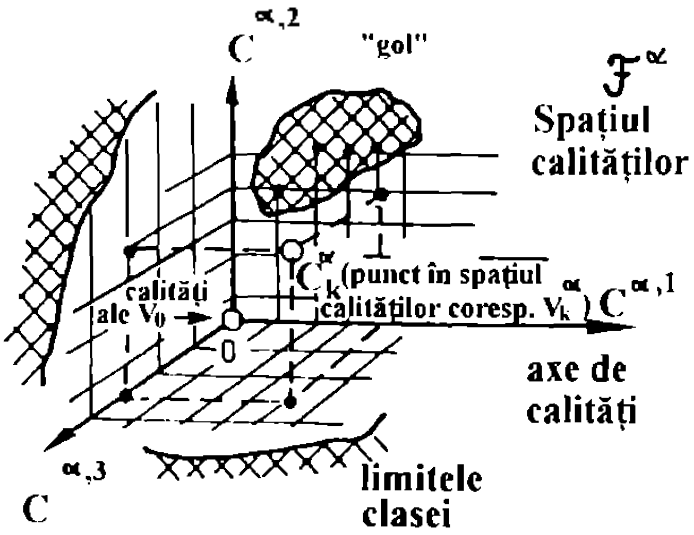


Fig. 5.2. Spațiul *calităților* programelor clasei

asocierea anumitor calități, având atribuite anumite valori, poate fi interzisă (din motive de contradicție internă în funcționarea programului). Un exemplu clasic în acest sens îl constituie impunerea unor restricții de timp de reacție (pentru sisteme în timp real) în condițiile unor specificații funcționale care să impună o anumită complexitate a structurii programului.

Transformările (intervențiile de proiectare) amintite mai sus, se efectuează asupra *calităților* variantelor, pentru a trece de la *calitățile* unei variante, la cele ale altei variante. Să notăm setul asociat *calităților* unei variante **k**, prin $\mathbf{c_k}^{\alpha} \in \mathcal{F}^{\alpha}$ și cu $\mathbf{s}^{\alpha}(\mathbf{c_k}^{\alpha}, \mathbf{t_k})$ operatorul prin care se realizează tranziția în spațiul *calităților*, de la calitățile unei variante la cele ale altei variante:

$$\mathbf{s}^{\alpha} : \mathcal{F}^{\alpha} \times \mathbf{T}^{\alpha} \rightarrow \mathcal{F}^{\alpha} \tag{5.7}$$

Între spațiul calităților și spațiul programelor există un *izomorfism conceptual*, în sensul că orice transformare în spațiul calităților se răsfrânge asupra programului față de care se exercită acțiunea de proiectare pentru reutilizare, rezultând acea variantă de program care are tocmai calitățile transformate. O justificare intuitivă, pentru această afirmație rezultă dacă luăm în considerare o schematizare a situației reale care atribuie fiecărei calități un anumit *modul de program* astfel încât cuantificările calității respective să corespundă prezenței, în cadrul modulului respectiv, a unor fragmente de cod specificate univoc. În acest caz, dacă în spațiul calităților efectuăm o deplasare în sensul trecerii de la o realizare la alta a unei anumite calități, atunci în spațiul variantelor, acesteia îi va corespunde o inserare sau eliminare de linii-sursă, asociată diferenței de cuantificare a acelei calități (între cele două variante - înainte de modificare - după modificare). În consecință, o deplasare în spațiul calităților se reflectă prin modificări *izomorfe în spațiul variantelor*.

Procesul complet, real, de proiectare în clasă este reprezentat în Fig. 5.3 (pentru o clasă (α)). În figură, intrarea *"reală"* a **sistemului de proiectare** este setul de *cerințe* și *restricții* (*tema de proiectare*) care se impune programului care se proiectează ca și în Fig. 1.1, dar prin intermediul *"blocului expert"*. Schema include și două bucle de reacție care asigură respectarea temei de proiectare pentru proramul proiectat, prin compararea *performanțelor* și *cerințelor* de intrare cu rezultatele proiectării (ca în [Sav96], v. Fig. 1.19). Blocul numit *"sistem expert"*, are rolul de a transforma prescripțiile de intrare ajustate cu reacțiile sistemului, în mutații în *spațiul calităților*, iar blocul *translator*, are rolul de a transforma (traduce, translata) cerințele de *calitate* ale noului program care trebuie obținut, în intervenții asupra programelor, respectiv în *intrări* în **SPC**. Blocul translator are atribuții care apropie modelul de metoda **DRACO** (§ 1.7.1,a), prin ideea aceasta a unor reprezentări interne ale proiectării, respectiv, de realizare de *transformări* dintr-o reprezentare folosită, în alta.

Izomorfismul conceptual de care aminteam mai înainte, se transformă în *izomorfism real*, dacă procesul **proiectării în clasă** se stabilizează. Stabilizarea are loc prin iterarea modificărilor și ajustărilor necesare pentru a atinge dezideratele de performanță și cerințele

funcționale impuse prin tema de proiectare (a se vedea în acest sens și schema din Fig. 1.19).

Atribuțiile *blocului expert* sunt preluate de cele mai multe ori de factorul uman (proiectant sau echipă de proiectanți). Spre deosebire de intervenția asupra proiectului, modelat teoretic de operatorul d(.), *blocul expert*, al cărui rol este preluat de factorul uman, este greu de modelat în detaliu, întrucât atribuțiile sale sunt chiar și intuitiv dificil formalizabile. Din același motiv, intrările desemnate ca și *cerințe*, respectiv, *performanțe*, sunt de asemenea greu formalizabile. În general, procesul mental al proiectării în sine scapă modelării operaționale. Astfel, experiența subiectivă de proiectare extrapolează cel mult reguli euristice, utile în practica proiectării, dar departe de dezideratele necesare modelării și implicit automatizării proiectării. Unele observații pertinente în acest sens sunt prezentate în [Sav96] Pentru rezolvarea problemei automatizării proiectării, există încercări de folosire a unor limbaje speciale, numite *limbaje de specificații* [Vas85][Che84-L][Big84]. În acest caz, proiectarea se transformă într-o compilare a specificațiilor. Pe bună dreptate s-a afirmat că o descriere formală corectă, completă și univocă a cerințelor ar rezolva aproape integral proiectarea automată a

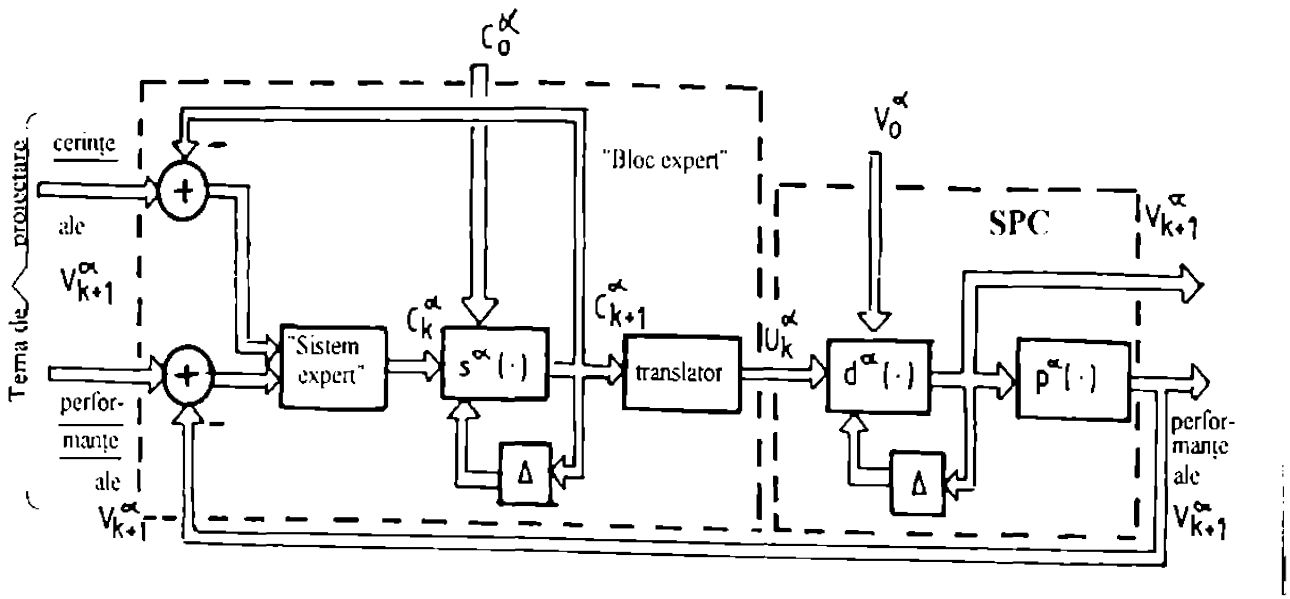


Fig. 5.3. Procesul complet al proiectării în clasă

aplicațiilor.

Dacă activitatea desemnată mai sus ca fiind subiectivă este preluată de un program, **SPC** devine un **generator de aplicații (GA)**. Acesta, datorită dificultăților amintite mai sus, precum și din motive legate de complexitatea modelului $(\mathbf{d(.),p^{\alpha}(.)})$, este în general destinat generării de variante într-o clasă *restrânsă* de aplicații.

Date fiind cele de mai sus, putem să încercăm să definim **sistemul de proiectare în clasă** ca *sistem dinamic*. Vom folosi în acest scop definiția unui sistem dinamic, din [Kal75]. În acea definiție, **valorile de intrare**, respectiv **valorile de ieșire** sunt definite relativ la funcții de

intrare admise ω^{α}, respectiv funcții de ieșire admise γ^{α}, aparținând unor *clase de funcții de intrare admise* Ω^{α}, respectiv *clase de funcții de ieșire admise* Γ^{α} :

$$\Omega^{\alpha} = \{ \omega^{\alpha} : T^{\alpha} \rightarrow U^{\alpha} \}$$
$$\Gamma^{\alpha} = \{ \gamma^{\alpha} : T^{\alpha} \rightarrow P^{\alpha} \}$$

Conform acestor notații, dăm următoarea :

(5.3) Definiție : *Octuplul (indexat)* $(T^{\alpha}, C^{\alpha}, \Omega^{\alpha}, U^{\alpha}, P^{\alpha}, \Gamma^{\alpha}, d^{\alpha}(.), p^{\alpha}(.))$ *se numește* ***sistem de proiectare în clasa*** (α) ***(SPC-*** α***).***

Indexul α este corespunzător celui prezentat la începutul paragrafului (denumirea clasei). **SPC** este deci un sistem dinamic discret, cu o mulțime discretă de *timp abstract* (pe cicluri de proiectare), având intrări (intervențiile operatorului), ieșiri (programe și performanțe ale acestora) și un operator specific de transformare (*proiectare*), precum și unul de ieșire (*performanțe*).

Acest sistem are câteva proprietăți evidente, pe care le enumerăm mai jos :

(5.4) Proprietăți:

1. - este ***sistem dinamic*** *(întrucât suferă transformări în timp);*

2. - este ***sistem discret*** *(atât în domeniul timp, deoarece transformările se derulează în etape-cuante de timp, asimilate ciclului de viață al programelor, cât și în domeniul stărilor, având stări discontinui - variantele de program);*

3. - este sistem cu ***număr finit de stări*** *(limitarea se poate realiza după o convenție oarecare a proiectantului, convenție care intervine și la definirea clasei (α) ; a se vedea în acest sens Observația (5.1));*

Principalul rezultat al acestui paragraf este definirea **SPC** *ca fiind un sistem dinamic care modelează prin operatorii* proiectare și performanțe *procesul de proiectare pentru reutilizare.*

5.2. Caracterizarea SPC

În cele ce urmează, vom prelua unele definiţii din [Kal75], cu adaptarea la specificul lucrării. Formulările se vor referi în special la definiţii care permit caracterizarea **SPC** în sensuri practice.

(5.5) Definiţie : *Faza (t_k, v_k) este* ***accesibilă*** *dacă şi numai dacă există $s_k \le t_k$ şi o intrare u(.) care transferă (s_k, v_0) în (t_k, v_k).*

(5.6) Definiţie : *Faza (t_k, v_k) este* ***controlabilă*** *dacă şi numai dacă există $t \ge t_k$ şi o intrare u(.) care transferă (t_k, v_k) în (t, v_0).*

(5.7) Definiţie : *Dacă orice (t_k, v_k) cu k fixat şi $v_k \in C^{\alpha}$ este accesibilă, atunci vorbim de* ***accesibilitate completă la momentul*** *t_k.*

Dacă în plus pentru orice k sunt întrunite condiţiile de accesibilitate completă, spunem că **SPC** este **complet accesibil**.

În cazul nostru, *accesibilităţii* trebuie să-i conferim o semnificaţie mai importantă decât *controlabilităţii*. Într-adevăr, scopul proiectării nu este acela de a reveni la varianta generică ci de a produce, plecând de aici, noi variante.

(5.8) Observaţie: *Evident că în cazuri reale, în situaţia existenţei unor clase deschise (în care membrii clasei sunt generaţi doar în măsura cererii pieţei), este dificilă caracterizarea globală a clasei din punctul de vedere al accesibilităţii, şi aceasta din mai multe motive :*

> ***5.8.1.*** *De cele mai multe ori C^{α} nu este riguros stabilit (a se vedea Observaţia (5.1)) şi datorită dificultăţilor de modelare nu putem afirma apriori (chiar când dispunem de un model pentru* ***d(.)****) că o variantă poate fi obţinută cu certitudine, atunci când prin cerinţe poate fi încadrată la clasa* ***(a)****.*
> ***5.8.2.*** *Lipsind un formalism riguros pentru caracterizarea* ***SPC*** *operatorul* ***d(.)*** *nu este suficient de riguros definit.*

Situaţia opusă celei din Obs. (5.8) este aceea în care clasa de aplicaţii este riguros definită, mulţimea variantelor este finită şi cunoscută, iar trecerea de la o variantă la alta este de

asemenea clar definită. În acest caz, enunţăm următoarea :

(5.9) Proprietate: *Dacă modelul **SPC** este exact şi C^{α} este riguros definit şi mărginit, **SPC** este un **automat finit**.*

Numim clasele pentru care **SPC** este *automat finit*, ***clase închise***.

Proprietatea enunţată este o urmare directă a definiţiei unui automat finit [Mun77]. Întrucât un **SPC** este un sistem dinamic, având definite mulţimea stărilor (variantele), mulţimea semnalelor de intrare (intervenţiile operatorului), mulţimea semnalelor de ieşire (variante şi performanţe), respectiv două funcţii (funcţia de proiectare, pe care o putem numi şi *funcţie de tranziţie*, respectiv funcţia de ieşire - operatorul *performanţe* - pe care o putem numi şi *funcţie de transcodaj*), rezultă că un **SPC** este un *automat Mealy* [Mun77]. Deoarece ipotezele de lucru ale Proprietăţii (5.9) limitează mulţimile caracteristice ale **SPC**, rezultă cu necesitate că acel **SPC** care respectă ipotezele amintite, este un *automat finit*. Problema proiectării în acest caz se rezolvă foarte simplu şi corespunde situaţiei din practică în care pentru un set finit de aplicaţii ale clasei (clasa fiind redusă la acest set) sunt definite riguros variabilele, iar din când în când, în funcţie de cerere, se generează una sau alta din variantele deja cunoscute (sau având, în cazul mai general, algoritmi cunoscuţi de obţinere). O tehnică extrem de simplă în acest caz este folosirea parametrizărilor în sursa programului generic (a se vedea în acest sens şi [Gog84]). În legătură cu această proprietate, enunţăm următoarea:

(5.10) Lemă: *În condiţiile de la Proprietatea (5.9) **SPC** prezintă accesibilitate completă.*

Demonstraţia este imediată: întrucât **SPC** se reduce la un **automat finit**, orice variantă poate fi obţinută într-un număr determinat de paşi (deci într-un orizont de timp finit), trecând printr-un număr (mărginit) de tranziţii ale automatului finit. Fiind sistem discret, de fapt orice stare poate fi atinsă cel mult

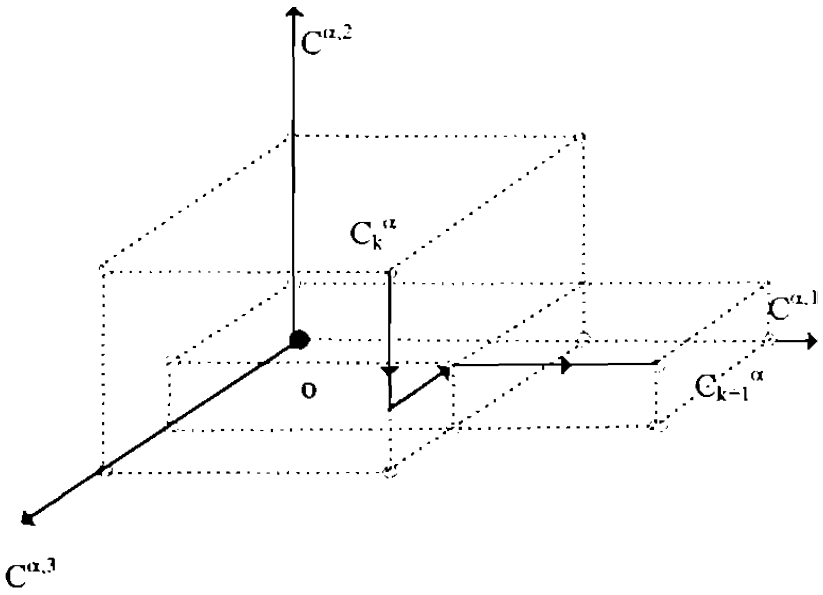


Fig. 5.4. Transformări ortogonale în procesul complet al proiectării în clasă

într-un număr de paşi egal cu dimensiunea spaţiului calităţilor, deci cu numărul de cerinţe ale sistemului. Bineînţeles, aceasta se realizează atunci când putem elabora orice comandă, deci operatorul poate interveni oricât de "activ" în procesul de proiectare. În realitate, dacă proiectarea nu este realizată de un **generator de aplicaţii**, operatorul uman (proiectantul) are o capacitate de intervenţie limitată de abilităţile sale de a procesa doar *secvenţial*. Aceasta se traduce prin aceea că de fapt transformările efectuate asupra unei variante se reflectă prin transformări în spaţiul *calităţilor*, de-a lungul axelor de coordonate, deci o deplasare de la o variantă la alta va avea loc prin adiţionarea unor deplasări ortogonale, ca în Fig. 5.4.

Enunţăm în continuare următoarea

(5.11) Teoremă: *Un SPC nu este în general complet accesibil.*

Intuitiv, enunţul poate fi pus în legătură cu Observaţia 5.8. Pentru demonstraţie, e suficient să observăm că în condiţiile existenţei unor "combinaţii interzise" în spaţiul calităţilor (v. Fig. 5.2), prin izomorfismul conceptual amintit în § 5.1, există programe (variante) spre care nu sunt definite tranziţii. În aceste condiţii, automatul Mealy constituit de **SPC** respectiv, este un **automat incomplet** [Mun77]. Asta înseamnă că nu toate stările pot fi atinse, deci nu toate stările sunt **accesibile**.

În legătură cu proprietatea de **observabilitate**, aşa cum este cunoscută în teoria sistemelor, nu putem face aprecieri riguroase întrucât în lipsa unei definiţii clare a operatorului **d(.)** nu ni se furnizează nici un fel de informaţii despre *stări*, altele decât cele definite de noi, din C^{α}. De altfel, orice punct din spaţiul *calităţilor*, implicit orice punct din spaţiul variantelor este **observabil** în sensul că prin coordonatele fixate pe axele calităţilor, starea este univoc definită şi observabilă. Aceasta este o *observabilitate implicită*, decurgând din modul de definire a stării în spaţiul calităţilor. De altfel, în acest spaţiu, este banal să arătăm că şi criteriile clasice, sistemice, de observabilitate, sunt satisfăcute. Într-adevăr, fie $c_k^{\alpha} \in \mathcal{C}^{\alpha}$ vectorul calităţilor unui program v_k^{α}:

$$c_k^{\alpha} = (c_k^1, c_k^2, ..., c_k^n)^T.$$

Ca *model intrare-stare-ieşire discret* al părţii sistemului de proiectare care realizează trecerea de la o *calitate* la alta, se poate considera ansamblul de ecuaţii :

$$\begin{bmatrix} c_{k+1}^1 \\ c_{k+1}^2 \\ . \\ . \\ . \\ c_{k+1}^n \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & & 0 \\ 0 & 1 & 0 & & 0 \\ \cdots & & & & . \\ \cdots & & & & . \\ \cdots & & & & . \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\mathbf{A}} \cdot \begin{bmatrix} c_k^1 \\ c_k^2 \\ . \\ . \\ . \\ c_k^n \end{bmatrix} + \underbrace{\begin{bmatrix} 1 & 0 & 0 & & 0 \\ 0 & 1 & 0 & & 0 \\ \cdots & & & & . \\ \cdots & & & & . \\ \cdots & & & & . \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\mathbf{B}} \begin{bmatrix} s_k^1 \\ s_k^2 \\ . \\ . \\ . \\ s_k^n \end{bmatrix}$$

$$\begin{bmatrix} c_k^1 \\ c_k^2 \\ . \\ . \\ . \\ c_k^n \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & & 0 \\ 0 & 1 & 0 & & 0 \\ \cdots & & & & . \\ \cdots & & & & . \\ \cdots & & & & . \\ 0 & 0 & 0 & \cdots & 1 \end{bmatrix}}_{\mathbf{C}} \begin{bmatrix} c_k^1 \\ c_k^2 \\ . \\ . \\ . \\ c_k^n \end{bmatrix}$$

Trecerea de la o *calitate* la alta necesită aplicarea unor intrări de forma :

$\mathbf{s_k}^{\alpha} = (s_k^1, s_k^2, ..., s_k^n)^T = (n_1, n_2, ...n_n)^T$, cu $n_i \in \mathbf{Z}$

Matricele **B** şi **C** fiind unitare, evident, matricele de *observabilitate* şi *controlabilitate* ale acestui sistem au rangul egal cu n.

Variantele în sine deşi teoretic sunt **observabile**, în realitate, datorită *izomorfismului* din spaţiul *calităţilor* în spaţiul *variantelor*, sunt disponibile doar după încheierea procesului de proiectare pentru fiecare variantă în parte. În acest sens, este dificil spre exemplu să imaginăm construirea unui *observator de stare*, în înţelesul din teoria sistemelor. În schimb, în condiţiile Proprietăţii (5.9) (respectiv, **SPC** este un automat finit), variantele sunt toate observabile, deoarece oricând putem obţine varianta dorită, prin faptul că un model de proiectare este disponibil şi deci varianta dorită este obţinută simplu, prin procesul de proiectare în sine.

Concluzia acestor consideraţii de mai sus este că, deşi **SPC** este **observabil** în sensul cunoscut din teoria sistemelor, această proprietate nu are în contextul prezentării un sens direct utilizabil *practic*. În sensul limbajului natural şi din punctul de vedere strict al *utilizatorului*, **propunem** însă o clasificare, utilizabilă practic, referitoare la conţinutul unor categorii de **SPC** şi la rezultatul obţinut în urma reutilizării :

(5.12) Clasificare:

*1 - **SPC** în care proiectarea este realizată direct de utilizator, prin folosirea unor tehnici de proiectare în clasă "clasice" (adică, proiectare obişnuită, dar folosind module de program concepute special pentru reutilizare) : în acest caz, evident proiectantul ştie ce este în sursa programului generic şi produsul procesului de proiectare este de asemenea complet disponibil sub formă de cod-sursă, de aceea aici **SPC** este **complet observabil**.*

*2 - **SPC** sub formă de **GA** care produc integral cod-sursă: în acest caz, **SPC** este de asemenea **complet observabil**, deoarece produsul obținut este disponibil integral sub formă de cod sursă, deci poate fi inspectat complet ;*

*3 - **SPC** sub formă de **GA** care produc cod-sursă, dar codul obținut are înglobate și apeluri de funcții de bibliotecă transparente pentru utilizator: în acest caz, **SPC** este **parțial observabil** (caz prezentat în [Nin94] - programul LabWINDOWS).*

*4 - **SPC** sub forma unor **GA** care produc direct cod executabil: aici **SPC** este **neobservabil** (cazuri prezentate în [KMe93] și [NIC92] - programele ViewDAC, respectiv, LabVIEW).*

Aici fiind vorba doar de aparența exterioară a unui **SPC**, nu se poate face o legătură între accepțiunile din clasificarea de mai sus și proprietățile operatorului **d(.)**

Pentru a explora unele proprietăți interesante ale **SPC**, dăm în continuare un set de definiții [Kal75] adaptate specificului **SPC**. Aceste definiții folosesc noțiunea de *fază* : cuplul variabilă de stare - moment de timp asociat. Pentru simplificarea notației, a fost omisă indexarea funcție de clasa de aplicații (α).

(5.13) Definiție: *Două faze (t_k,v_{k1}) și (t_k,v_{k2}) ale unui **SPC** aparțin aceleiași **clase de observare** (nu se pot deosebi în viitor) dacă și numai dacă:*

$$p(d(v_{k1},u(.),t_k),t) \quad p(d(v_{k2},u(.),t_k),t)$$

pentru toți $t \geq t_k$ și toți $u(.)$.

Definiția complementară este:

(5.14) Definiție: *Două faze (t_k,v_{k1}) și (t_k,v_{k2}) ale unui **SPC** aparțin aceleiași **clase de construcție** (nu se pot deosebi în trecut) dacă și numai dacă*

$$p(d(v_{k1},u(.),t_k),s) \quad p(d(v_{k2},u(.),t_k),s)$$

pentru toți $s \leq t_k$ și toți $u(.)$.

Definițiile (5.13) și (5.14) sunt de factură generală. Aplicate condițiilor **SPC**, duc la rezultate specifice. Să comentăm câteva dintre acestea. Înainte de toate, să observăm că în cazul **SPC**, dacă avem două variante având același set de proprietăți, putem afirma că de fapt cele două variante sunt identice. Demonstrăm afirmați prin reducere la absurd : să presupunem că cele două variante sunt diferite dar au același set de proprietăți (deci în ultimă instanță satisfac același set de cerințe). Înseamnă că proiectantul a generat două programe diferite având aceeași destinație, respectând același set de cerințe. Din considerente practice, această situație e

absurdă : spre exemplu, în cadrul unei organizaţii software, paralelismele în proiectare sunt aproape întotdeauna evitate şi se duce o politică organizaţională care să descurajeze puternic asemenea tendinţe, întrucât duc la risipă de resurse. Pe de altă parte, am presupus că există un *izomorfism conceptual* între spaţiul variantelor şi spaţiul programelor. Rezultă, cu atât mai mult, că două variante cu aceleaşi proprietăţi sunt identice. De aceea, în aproape toate situaţiile practice, Definiţia (5.13) este trivială pentru un **SPC** şi corespunde situaţiei în care o variantă poate fi obţinută prin tranziţie de la două variante distincte, în care caz, evident, putem afirma că acele două variante *nu pot fi deosebite în viitor*, întrucât au fost transformate de fapt în una şi aceeaşi variantă. A doua definiţie exprimă o situaţie contrară : prin transformări, plecând de la aceeaşi variantă, se vor obţine cele două variante care *nu se pot deosebi în trecut*. Cele două situaţii sunt prezentate în Fig. 5.5, a) şi b).

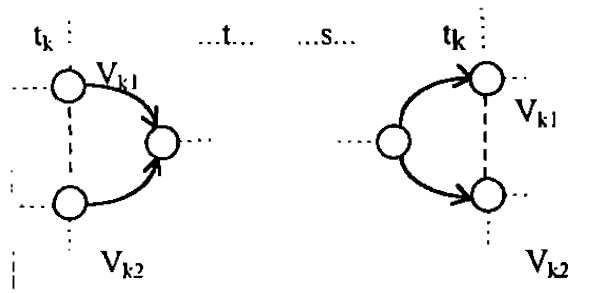


Fig.5.5,a)
Faze care nu pot fi deosebite în viitor

Fig.5.5,b)
Faze care nu pot fi deosebite în trecut

(5.15) Definiţie: *Faza (t_k,v_k) a **SPC** este **neobservabilă** dacă şi numai dacă aparţine clasei de observare a lui (t_k,v_0).*

(5.16) Definiţie: *Faza (t_k,v_k) a **SPC** este **necontrolabilă** dacă şi numai dacă aparţine clasei de construcţie a lui (t_k,v_0).*

Să comentăm definiţiile (5.15) şi (5.16). Într-adevăr, dacă o fază oarecare şi o fază asociată cu programul generic aparţin aceleiaşi clase de observare, înseamnă că ele vor duce, prin transfomări, la aceeaşi variantă, ceea ce în condiţiile **SPC** înseamnă că de fapt varianta fazei respective coincide chiar cu programul generic (căci nu putem avea în cadrul clasei programe din care să se obţină variante identice cu cele care pot fi obţinute direct din programul generic, altfel spus, nu există în cadrul clasei programe care să joace rolul programului generic). Aceasta ne duce înapoi la afirmaţia că de fapt **SPC** este *observabil*. Astfel, definiţia (5.15) devine şi ea trivială pentru un **SPC**, întrucât programul fazei care aparţine aceleiaşi clase de observare cu faza asociată programului generic, este de fapt chiar programul generic. A doua definiţie sugerează situaţia în care două programe, dintre care unul este programul generic, nu pot fi deosebite în trecut. Dar nu putem avea programe anterioare programului generic, de aceea şi această definiţie este trivială, întrucât faza necontrolabilă devine şi aici însuşi faza asociată programului generic.

În sensul definiţiilor de mai sus, formulăm următoarea

(5.17) Lemă: *v_0 este neobservabilă şi necontrolabilă.*

Demonstrația este imediată, prin cumularea comentariilor de mai sus, referitoare la Definițiile (5.15) și (5.16). Într-adevăr, *programul generic* trebuie neapărat să fie proiectat convențional. Aceasta nu face decât să confirme aserțiunile din [Tra88], după care, înainte de a dezvolta software *reutilizabil*, trebuie să-l folosești de cel puțin 3 ori (deci de fapt primele câteva programe ale unei clase, nu numai *programul generic*, sunt obținute prin tehnici convenționale de programare).

Dacă operatorul p(.) cu definiția din relația (5.6) îl restricționăm ca operator de **performanțe generale p"** atunci se poate formula următoarea

(5.18) Teoremă: *În raport cu operatorul* p^{α}, *clasa (a) din* $SPC\ (T^{\alpha}, C^{\alpha}, U^{\alpha}, P^{\alpha}, d^{\alpha}(.), p^{\alpha}(.))$ *este clasă de observare și clasă de construcție.*

Aceasta înseamnă că pentru toate variantele unei clase există o caracterizare comună care face ca variantele să nu poată să fie deosebite pe baza trăsăturilor generale (care, de fapt, dau specificul clasei respective), nici *în viitor* și nici *în trecut*. Aceasta înseamnă că indiferent unde suntem plasați în cadrul clasei de aplicații, oricare variantă poate fi obținută plecând de la variante diferite și de asemenea plecând de la o anumită variantă, se pot obține variante diferite. De fapt, prin această afirmație, justificăm *echivalența* programelor (variantelor) în cadrul unei clase (care nu este însă același lucru cu *echivalența* stărilor unui automat [Mun77]), adică proprietatea ca orice variantă să poată fi obținută plecând de la oricare program al clasei, considerat program generic. Totuși, această echivalență, astfel formulată, are un rol pur formal, întrucât este mult mai practic să se plece în proiectare întotdeauna de la programul generic, considerat programul cel mai "universal", cel mai "complet" al clasei. Efortul de proiectare al obținerii unei variante este astfel pe ansamblu mult mai mic decât dacă s-ar pleca în proiectare de la o variantă oarecare.

Afirmația că există trăsături generale care caracterizează membrii unei clase pare trivială, căci nimic nu pare mai mai firesc decât să susținem că entități din aceeași categorie (clasă) au o serie de trăsături comune care de fapt dau aspectul de coeziune între membrii categoriei respective. Aceasta se dovedește o capcană a accepțiunilor curente, induse de folosirea limbajului natural. În realitate această afirmație se dovedește o reflectare a unor caracteristici structurale. Delimitarea unei categorii restrânse de proprietăți ca și "*performanțe generale*" are un caracter profund subiectiv, dacă ne raportăm la accepțiunea curentă. Într-un alt paragraf, vom prezenta și o încercare de încadrare formală a acestei categorii.

Delimitarea operatorilor p^{α} și p^{α} prezintă aceleași dificultăți ca și celelalte consecințe rezultate din Observația (5.1).

*Acest paragraf este important pentru formalismul **SPC** întrucât subliniază importanța conceptului de **accesibilitate** pentru problema tratată. De asemenea, se demonstrează că sistemul izomorf **SPC** din spațiul calităților este **observabil** (ceea ce este un rezultat de așteptat) iar **SPC** pot fi clasificate dintr-un punct de vedere exterior, practic, conex cu observabilitatea, ca în clasificarea (5.12). Se mai demonstrează în acest paragraf că programul generic al **SPC** nu poate fi obținut decât prin tehnici clasice, ceea ce confirmă de asemenea ceea ce se știe din practică.*

5.3. Măsuri în SPC

Analog *gradului de controlabilitate* conturat în Observația (5.11) din lucrarea de referință [Kal75], unde acesta este definit relativ la "energia semnalului de reglare", propunem ca măsură a *gradului de accesibilitate* pentru o fază, *efortul de proiectare*. Conceptul de *efort de proiectare* este introdus și tratat în [Ram88], fără a se face vreo conexiune cu conceptul de *accesibilitate* (vezi § 2.2).

Propunem în cele ce urmează, o definiție proprie a gradului de accesibilitate care să țină cont de gradul de reutilizare de clasă sau - de program precum și de volumul de program de prelucrat. În aceste condiții, dăm următoarele definiții (în parte formulate în lucrările [Sto94-S],[Sto95-A] și [Sto95-S]):

(5.19) Definiție : *Se numesc **efort relativ de acces** (la o variantă), notat cu* e^a_p *, respectiv **efort absolut de acces** (la o variantă), notat cu* w^a_p *, mărimile :*

$$e^a_p = 1 - r_p \qquad (5.8,a)$$

$$w^a_p = n_{v0} * e^a_p = n_{v0} * (1 - r_p) \qquad (5.8,b)$$

unde: r_p - gradul de reutilizare de program

n_{v0} - numărul total de linii-sursă al programului generic

Intuitiv, *efortul relativ de acces*, ca o mărime complementară gradului de reutilizare de program, furnizează o informație relativă, *adimensională* - pe o scară subunitară - referitoare la modificările suferite de programul generic pentru a ajunge la o variantă anumită (varianta pentru care se calculează gradul de reutilizare de program). *Efortul absolut de acces* furnizează o imagine "*dimensională*" asupra numărului de linii-sursă modificate pentru a ajunge de la programul generic la varianta pentru care se calculează respectivul efort (*estimează* acest număr, de fapt dă un număr proporțional cu valoarea amintită mai sus). Acest coeficient numeric poate

fi numit din acest motiv şi *volum informativ de muncă* depus pentru a obţine varianta pentru care este calculat.

(5.20) Definiţie : *Se numeşte* ***grad de accesibilitate de program,*** *notat cu* r^a_p, *mărimea*

$$r^a_p = \frac{w^a_p}{n_{v0}} = \frac{n^2_{v0}(1 - r_p)^2}{n_{v0}} = n_{v0}(1 - r_p)^2 \qquad (5.9)$$

Gradul de accesibilitate propus are (prin analogie) aspect de "*energie a semnalului de reglare*" (care este o mărime fizică având dimensiuni de energie şi reprezintă energia transferată unui sistem în buclă închisă prin semnalul de reglare) [Kal75], întrucât pentru procesul de proiectare software pentru reutilizare, "*semnalul de reglare*" este volumul de muncă prestat în vederea obţinerii unei variante noi, respectiv numărul de linii-sursă modificate pentru a obţine acea variantă. Considerând volumul de muncă, măsurat în număr de linii-sursă, la pătrat, relaţia prezentată dobândeşte aspect de "*energie*", deci analogia dorită este realizată.

Gradul de accesibilitate de program măsoară de fapt "cât de uşor" se poate obţine un program din cadrul unei clase (deci o variantă) plecând de la programul generic, adică oferă o estimare (după o regulă fuzzy, conform definiţiei *gradului de reutilizare de program*) a numărului de linii-sursă care trebuie modificate pentru a obţine varianta.

Definiţia (5.20) nu este legată strict de definirea fuzzy a gradului de reutilizare de program. În fond, aceeaşi definiţie poate lua în considerare accepţiunea din [Bro90], cea mai uzuală, pentru gradul de reutilizare de program (însă, cu dezavantajele de rigoare, prezentate la Cap. 2).

Definim în continuare un efort specific pentru caracterizarea globală a unei clase :

(5.21) Definiţie: *Se numesc* ***efort mediu relativ de acces,*** *notat cu* e^a_c , *respectiv* ***efort mediu absolut de acces,*** *notat cu* w^a_c , *mărimile :*

$$e^a_c = 1 - r_c \qquad (5.10,a)$$

$$w^a_c = n_{v0} * e^a_c = n_{v0} * (1 - r_c) \qquad (5.10,b)$$

unde: r_c - gradul de reutilizare de clasă

Semnificaţiile eforturilor medii de acces sunt similare cu cele ale eforturilor de acces, dar se referă la caracterizarea globală a unei clase, furnizând cuantificări care dau o imagine de ansamblu a volumului de muncă "*mediu*" care trebuie prestat pentru a ajunge la o variantă oarecare a clasei.

(5.22) Definiţie : *Se numeşte* ***grad de accesibilitate de clasă,*** *notat cu* r^a_c, *mărimea* :

$$r_c^a = \frac{w_c^{a^2}}{n_{v0}} = \frac{n_{v0}^2(1-r_c)^2}{n_{v0}} = n_{v0}(1-r_c)^2 \qquad (5.11)$$

unde : r_c - gradul de reutilizare de clasă

Astfel, se obţine un indice cantitativ care permite estimarea **volumului de activitate** pentru realizarea proiectării în cadrul clasei (respectiv, obţinerea variantelor plecând de la programul generic). Indicele acesta diferă însă de *gradul de accesibilitate* din definiţia (5.20), deoarece nu se referă la o anumită variantă din cadrul clasei ci la întregul efort de obţinere a variantelor, în sensul unei caracterizări globale, la nivelul clasei. Acest grad de accesibilitate poate de fapt să măsoare calitatea medie de *reutilizabilitate* pentru clasa de programe pentru care este calculat.

Indicii obţinuţi permit deci aprecierea volumului de activitate. Această apreciere este utilă la nivelul unor colective mari de programatori, mai ales în ceea ce priveşte aprecierea resurselor umane, a necesarului şi a evoluţiei acestora, respectiv, a costurilor de proiectare.

Exemplu numeric :

Pentru sistemul prezentat în [Tiv87-N], referitor la un sistem de telemecanică, s-a obţinut (§ 4.2) pentru gradul de reutilizare de clasă valoarea $r_c = 0.83$.

Presupunând că programul generic are 7000 linii-sursă, prin folosirea relaţiei de definire (5.11) se obţine un *grad de accesibilitate de clasă* :

$$r^a_c = n_{v0}(1 - r_c)^2 = 7000 * (1 - 0.83)^2 = 202$$

Cu cât numărul obţinut este mai mare, cu atât efortul de proiectare - în **general**, referitor la clasa respectivă - este mai mare (adica, efortul de a trece de la o variantă la alta, luând în considerare toate schemele posibile de reproiectare, în limitele pentru care a fost definită clasa). Numărul obţinut reflectă, în acest mod, cât de "bună" este clasa din punctul de vedere al reutilizării, cât de accesibil - **în medie** - este un program aparţinând acestei clase. Dacă pentru o altă clasă se obţine un număr mai mare, putem afirma că această clasă nu este aşa de "bună" din punctul de vedere al reutilizării, ca şi prima clasă, deci, aici, efortul de a obţine o variantă plecând de la programul generic, este, în medie, mai mare decât pentru prima clasă. Adică, programele celei de-a doua clase nu sunt aşa de **accesibile** precum cele din prima clasă.

Într-un fel, numărul obținut mai sus, este direct dependent de numărul de linii-sursă care trebuie modificate pentru a obține o anumită nouă versiune, în așa fel încât acest număr va reflecta proprietățile unui program mediu din punctul de vedere considerat. Altfel spus, dacă luăm în considerare toate variantele din clasă, respectiv toate schemele de modificare pentru a le obține din programul generic, numărul de linii-sursă care trebuie modificate în total, trebuie să fie egal cu cel proporțional în modul arătat mai sus, înmulțit cu numărul de variante. Intuitiv, clasa din exemplul nostru este caracterizată de proprietatea că membrii acestei clase diferă, în medie și în linii mari, de programul generic prin 202 linii-sursă. În realitate, numărul rezultat din calculul de mai sus este proporțional cu numărul de linii-sursă de modificat, dar legătura este dificil de exprimat analitic, în condițiile folosirii exprimării fuzzy pentru indicatorii primari.

Pentru ca reprezentarea să fie mai clară, mai realistă, propunem o exprimare proprotională, prin introducerea unui **factor de calibrare** care să aducă numărul rezultat din aplicarea relației-definiție (5.11) la dimensiunile practice și intuitive ale numărului de linii-sursă de modificat:

$$r_c^{a(cal)} = f^{cal} * r_c^a \qquad (5.11,a)$$

unde : $r_c^{a(cal)}$ - **grad calibrat de accesibilitate de clasă**

f^{cal} - factor de calibrare de acces de clasă

Astfel, există o legătură mai bună între definirea abstractă și modelul de reprezentare acceptabil intuitiv. Pentru determinarea factorului de calibrare propunem o tehnică instrumentală pe care o vom detalia în paragraful 5.7.

Se poate intui în continuare o *viteză* de trecere de la o variantă la alta, care poate fi exprimată în primul rând printr-o viteză "instantanee", cu referire la o singură fază a procesului de reproiectare (adică, un singur ciclu de proiectare, care înseamnă trecerea de la v_k la v_{k+1}). În relația de definiție a acestei viteze, apare o "diferență" între două variante succesive, notată "Θ".

(5.23) Definiție : *Se numește* ***viteza de fază a proiectării,*** *notată cu* $s_k^{f,a}$ *, mărimea :*

$$s_k^{f,a} = \frac{v_k^a \Theta v_{k-1}^a}{t_{k,k-1}} \qquad (5.12)$$

unde : $t_{k,k-1}$ - timpul fizic (concret) de realizare a proiectării (a trecerii de la v_{k-1} la v_k)

(5.24) Observație: *Dificultatea definirii unei viteze de (re)proiectare constă în definirea unei metrici potrivite pentru calculul "diferenței" dintre două variante.*

Într-adevăr, măsurile propuse de noi combină definiţii de măsuri software cu tehnici şi abordări specifice teoriei sistemelor. În lucrarea de faţă nu susţinem că măsurile software propuse pot "măsura" toate aspectele reutilizării software, a proiectării pentru reutilizare. Mai exact, nu susţinem că dacă din măsurile software propuse rezultă că programul X este mai "bun" ca programul Y, atunci rezultă cu certitudine că programul X este de fapt mai rapid, sau are o structură mai "bună", sau este în alt fel mai bun decât programul Y. Afirmăm doar că dacă din analiza măsurilor definite mai sus rezultă că programul X este mai bun decât programul Y, aceasta înseamnă că, *folosind o reprezentare cantitativă care ţine cont de aspectele de reutilizare software* în modul definit de noi, atunci din acest punct de vedere am indicat rezultatele unei evaluări cantitative a calităţii de reutilizare, respectiv, că programul X este mai bun decât programul Y din *punctul de vedere al reutilizării software*. Însuşindu-ne punctul de vedere al lui Rammamurthy şi Melton [Ram88], potrivit cărora metrica este un concept matematic bine definit, iar o *măsură software* nu este chiar o metrică definită în sens matematic, dorim să utilizăm cu precauţie termenul *metrică* în software. O metrică este o funcţie care evaluează două argumente şi a cărei valoare returnată reprezintă o "diferenţă" între cele două argumente. Însă în cele mai multe abordări [Fen91], funcţia corespunzătoare are un singur argument şi deci valoarea returnată reprezintă o măsură referitoare la caracteristici ale doar unui argument. Inadvertenţa este eliminată în bună măsură prin propunerile de mai jos, ale lucrării de faţă. Începem, însă, prin a aduce observaţiilor de mai sus o serie de amendamente:

i) Reprezentarea noastră pentru măsurile software primare (gradele de reutilizare) este fuzzy, deci ea este chiar potrivită pentru genul de definiţii afectate de subiectivism, ca şi cele de mai sus.

ii) Pentru calculul vitezei de proiectare de faţă, este necesară efectuarea diferenţei între două variante. Mittermeir şi Oppitz introduc în [Mit87] funcţii homomorfice pentru procesul proiectării şi putem uşor defini o diferenţă între două variante folosind o metrică adecvată; bineînţeles, problema nu se rezolvă uşor şi imediat.

iii) Viteza este o caracteristică naturală a proceselor dinamice, deci şi pentru cel al proiectării şi reproiectării software (viteză de proiectare).

iv) Faptul că utilizarea noţiunii de metrică software comportă anumite precauţii, impune definirea riguroasă a unor indicatori cantitativi riguroşi luând în considerare acele precauţii.

În concluzie, credem că este mai bine să găsim reprezentări cantitative, chiar dacă acestea sunt afectate de subiectivism. Acest din urmă dezavantaj poate fi înlăturat dacă definim regulile de inferenţă luând în considerare experienţa mai multor specialişti.

De aceea în cele ce urmează, încercăm să definim clar şi metrica software utilizată pentru diferenţa dintre două variante.

O manieră de rezolvare este adoptarea uneia din următoarele două *cvasi-metrici* pentru definirea operatorului "θ" din relaţia de definire (5.12) :

- definirea "diferenţei" dintre două variante ca modulul diferenţei dintre eforturile absolute de acces ale celor două variante :

$$d(v_k, v_{k-1}) = v_k \, \theta \, v_{k-1} = | w^a_{p,k} - w^a_{p,k-1} | = n_{v0} * | r_{p,k} - r_{p,k-1} | \qquad (5.13)$$

- definirea "diferenţei" dintre două variante ca fiind volumul de muncă de trecere directă de la o variantă la alta.

În al doilea caz, similar cu definirea efortului absolut de acces, înmulţim numărul de linii-sursă, dar nu al programului generic ci al programului transformat (respectiv, v_{k-1}), cu efortul relativ de acces de la un program la altul. În consecinţă, este necesară folosirea *gradului de reutilizare de la un program la altul*, $r_p^{k,k-1}$ definit la începutul § 4.5. Drept urmare, a doua *cvasi-metrică* propusă va fi definită astfel :

$$d(v_k, v_{k-1}) \quad v_k \, \theta \, v_{k-1} = n_{v0} * (1 - r_p^{k,k-1}) \qquad (5.14)$$

Să demonstrăm în continuare că relaţia (5.13) defineşte o **semimetrică** [Gas81] pe clasa de aplicaţii (α) (În demonstraţie am folosit variante indexate arbitrar, cu i,j, respectiv k, ceea ce conferă demonstraţiei o valabilitate mai generală) :

1) În definiţia din relaţia (5.13), modulul nu poate fi decât pozitiv. În plus, dacă se calculează "diferenţa" dintre o variantă şi ea însăşi, modulul devine nul, adică :

$d(v_i, v_j) \geq 0$ şi $d(v_i, v_i) = 0$

Deci relaţia propusă este *semipozitiv definită*.

2) Datorită existenţei modulului în relaţia de definire, $d(v_i, v_j) = d(v_j, v_i)$, deci relaţia este *simetrică*.

3) *Inegalitatea triunghiului* $d(v_i, v_j) \leq d(v_i, v_k) + d(v_k, v_j)$ este respectată, întrucât :

$d(v_i, v_j) = n_{v0} * | r_{p,i} - r_{p,j} |$

$d(v_i, v_k) + d(v_k, v_j) = n_{v0} * (| r_{p,i} - r_{p,k} | + | r_{p,k} - r_{p,j} |)$

respectiv

$| r_{p,i} - r_{p,j} | \leq | r_{p,i} - r_{p,k} | + | r_{p,k} - r_{p,j} |$

(inegalitatea triunghiului este adevărată pentru o sumă de module).

Deci am demonstrat de fapt următoarea :

(5.25) Teoremă : *O clasă de aplicații este spațiu semimetric în raport cu funcția definită prin relația (5.13).*

Funcția (relația) (5.13) *nu este o metrică* deoarece $d(v_i, v_j) = 0$ nu implică neapărat $v_i = v_j$. Într-adevăr, putem să ne imaginăm existența a două variante cu același grad de reutilizare de program, dar să fie diferite între ele : acestea sunt două programe care diferă de programul generic în aceeași măsură, dar în sensuri diferite, adică prin module de program și structuri diferite.

Relația (5.14) nu este nici măcar *semimetrică*, întrucât *gradul de reutilizare de la un program la altul* nu este simetric (a se vedea în acest sens sugestiile de la pct b) al demonstrației Proprietății (4.12,b)) :

$$d(v_i, v_j) = n_{v0} * (1 - r_p^{i,j}) \neq n_{v0} * (1 - r_p^{j,i}) = d(v_j, v_i)$$

În continuare, definim o viteză "globală" pentru reproiectarea software, care caracterizează întreaga clasă, din punctul de vedere al ușurinței reproiectării și al vitezei "medii" de obținere a unei variante :

(5.26) Definiție : *Se numește* ***viteza de grup a proiectării*** *mărimea*

$$s^{g,\alpha} = \frac{w_c^{a,\alpha}}{t_{car}} = n_{v0} * \frac{1 - r_c^{\alpha}}{t_{car}} \qquad (5.15)$$

În relația de mai sus t_{car} este *timpul caracteristic*. El caracterizează global, din punctul de vedere al timpului de proiectare, clasa de aplicații (α). Pentru definirea unor valori pentru acest *timp caracteristic*, putem adopta următoarele strategii :

a) desemnarea ca t_{car} a *timpului minim* de proiectare în clasă ; astfel, obținem o *viteză maximă de grup a proiectării* ;

b) desemnarea ca t_{car} a *timpului maxim* de proiectare în clasă ; astfel, obținem *viteza minimă de grup a proiectării* ;

c) desemnarea ca t_{car} a *timpului mediu* de proiectare în clasă ; această variantă de definire *este cea mai apropiată* de înțelesul unanim recunoscut al unei mărimi de caracterizare globală.

Pentru calcularea timpului mediu, propunem în continuare următorul algoritm :

i) Se desemnează din cadrul clasei un grup de câteva programe, considerate *relevante* (asupra acestui aspect vom mai reveni în cuprinsul acestui capitol) ;
ii) Se stabilesc timpii de obținere (de proiectare) ai tuturor acestor *programe relevante* ale clasei ;
iii) Se calculează timpul mediu t_{car} ca medie ponderată a timpilor de proiectare, pentru toate aceste programe, factorii de ponderare fiind rapoartele dintre efortul relativ de acces și efortul mediu relativ de acces, pentru fiecare program în parte, rezultând (prin folosirea relațiilor de definire (5.8,a) și (5.10,a)) :

$$t_{car} = \frac{1}{n_s} \sum_{i=1}^{n_s} \frac{e^a_{p,i}}{e^a_c} * t_i = \frac{1}{n_s} \sum_{i=1}^{n_s} \frac{1 - r_{p,i}}{1 - r_c} * t_i \qquad (5.16)$$

În relația de mai sus, am marcat cu indexul i, mărimile care caracterizează un anumit program din cadrul programelor desemnate *relevante*. t_i fiind timpul de proiectare al variantei v_i din cadrul celor n_s programe *relevante*.

Folosind relațiile de definire (5.15) și (5.16), se obține o formă completă de exprimare a vitezei de grup a proiectării, pe care o numim în continuare **viteza medie de grup a proiectării**, notată $s^{g,\alpha}_{med}$:

$$s^{g,\alpha}_{med} = n_{v(\alpha)} * n_s * \frac{1 - r_c}{\sum_{i=1}^{n_s} \frac{1 - r_{p,i}}{1 - r_c} * t_i} \qquad (5.17)$$

Pentru a nu complica exprimarea în membrul drept al ultimelor două relații de mai sus, am omis indicii de apartenență la clasa (α).

Viteza medie de grup a proiectării furnizează o informație utilă în aprecierea globală a posibilităților de reproiectare în cadrul clasei : practic, dă o imagine asupra facilității de a obține un anumit program, *în medie*, în cadrul clasei, măsurînd *permisivitatea* clasei relativă la obținerea membrilor ei. Mărimea respectivă, măsurată în număr de linii-sursă în unitatea de timp (s. ex. zile sau săptămâni), este potrivită pentru a fi utilizată în cadrul unor strategii coerente de *evaluare* în cadrul *organizațiilor software*. Mărimea furnizează informație atât

asupra calităților clasei (prin mărimea numărătorului) cât și asupra calificării proiectanților (prin mărimea numitorului).

Acest paragraf propune o serie de măsuri software originale : efort de acces, efort mediu de acces (relativ, absolut), grad de accesibilitate de program, grad de accesibilitate de clasă, viteza de fază a proiectării, viteza de grup a proiectării. Unele din aceste măsuri sunt referitoare la un program oarecare al clasei, altele caracterizează ***global*** *proprietățile clasei (acest din urmă principiu de caracterizare este, după cunoștința autorului, în totalitate original). Se propune și o* ***sentimetrică*** *care permite în continuare definirea vitezei de proiectare. Aceste măsuri software sunt atractive mai ales pentru implementarea unor strategii organizaționale.*

5.4. Problema fundamentală a PC

Prin analogie cu **problema realizării** [Kal75], adică a construirii (obținerii) modelului matematic al unui sistem dinamic pe baza unei *colecții abstracte de date experimentale*, formulăm :

(5.27) Problema fundamentală a PC este *problema construirii unui* ***SPC*** *în sensul Definiției (5.3) pe baza existenței unui program generic, a elementelor definitorii ale clasei și a factorului subiectiv (experiența proiectantului).*

În cazul nostru, *colecția abstractă de date* înseamnă existența unor perechi de cerințe-versiuni (intrări-ieșiri), versiunile fiind obținute prin tehnici convenționale de proiectare. Din punct de vedere practic, această problemă are o rezolvare relativ dificilă. Aceasta se datorează conturării vagi a limitelor unei clase (a se vedea în acest sens Observația (5.1,a)). De aceea, este discutabilă desemnarea *aprioricâ* a celor câteva programe disponibile (*colecția de date*) ca și programe *relevante* în cadrul clasei, în vederea construirii **SPC** pe baza acestora, prin rezolvarea **problemei fundamentale**. *A priori*, aceste programe disponibile, obținute convențional, pot fi considerate reprezentative *(relevante)* doar cu un anumit grad de aproximație : acela al creditării ipotezelor de apartenență la o anumită clasă de aplicații. Această dificultate există întrucât criteriile de desemnare a programelor *relevante* sunt, în formulările prezentate în continuare, toate, criterii **a posteriori**, deci care nu sunt instrumentale decât în măsura în care programele clasei sunt disponibile. Dar în acest caz rezolvarea **problemei fundamentale** se justifică doar pentru eventuale viitoare proiectări de variante, deci dacă există suficientă cerere estimată, în viitor.

Din comentariile de mai sus se desprinde şi necesitatea nuanţării rezolvării **problemei fundamentale**. Dificultăţile subliniate mai sus se referă mai ales la situaţia *claselor deschise* (v. Observaţia (5.1),a). Rezolvarea aceleiaşi probleme este mult mai simplă în cazul în care **SPC** este un *automat finit* (Proprietatea (5.9)), din motivul că în acest caz toate stările (variantele) sunt bine cunoscute, tranziţiile sunt riguros definite, deci se poate realiza (relativ) uşor un **GA** care să implementeze un **SPC** de această manieră. În cazul acestor *clase închise*, **problema fundamentală** se consideră rezolvată prin simpla definire riguroasă a *automatului finit* care este de fapt un **SPC**.

*Rezolvarea **problemei fundamentale** înseamnă de fapt construirea unui **GA** pentru clasa de aplicaţii pentru care se tratează acea problem, adică rezolvarea problemei automatizării proiectării pentru clasa de aplicaţii în cauză. Tocmai de aceea, considerăm că analogia cu **problema realizării** din teoria sistemelor este bună şi deci şi în domeniul reutilizării software putem considera că problema enunţată este chiar **problema fundamentală a PC**..*

5.5. Un criteriu de apartenenţă a unui program la o clasă

Potrivit celor precizate în introducerea capitolului de faţă, noţiunea de clasă se consideră o noţiune primară. Aceasta nu exclude necesitatea definirii unor criterii clare pentru delimitarea ei. În spiritul observaţiei (5.1), stabilirea marginilor, limitelor unei clase este relativ dificilă. În contrast, stabilirea unor criterii de apartenenţă a unui program dat, la o clasă de aplicaţii este mult mai simplă. Evident, primul criteriu de apartenenţă a unui program la o clasă constă în întrunirea unui set de cerinţe, de *calităţi* de către program. Această modalitate de definire a apartenenţei este un criteriu *calitativ*, afectat de subiectivism. Dacă nu putem face uz de acest criteriu, ori dorim să stabilim criterii cantitative univoce, este necesară o a doua manieră de abordare pentru stabilirea apartenenţei.

În cele ce urmează, se propune un criteriu "*a posteriori*" de apartenenţă [Sto95-O].

Considerăm o clasă de programe existentă, (α). Dorim să stabilim dacă un anumit program v neincadrat (deocamdată) în clasă poate fi inclus între variantele clasei. Propunem următorul

(5.28) Criteriu: *Fie o clasă de aplicaţii C^{α} având gradul de reutilizare de clasă r_c^{α} şi fie $\tilde{C}^{\alpha}$ $C^{\alpha} \cup v$ clasa de aplicaţii extinsă, obţinută prin adăugarea programului v la clasa C^{α}.*

Considerăm, în plus, că programul adăugat v este program relevant (în sensul folosit la calculul gradului de reutilizare de clasă) al clasei extinse şi că gradul de reutilizare de clasă al acesteia este $\tilde{r}_c^{\alpha}$.

Dacă diferența dintre cele două grade de reutilizare, r_c^{α} și $\tilde{r}_c^{\alpha}$ este mai mică în modul decât o limită prestabilită ε, considerăm că programul v aparține clasei C^{α}, cu precizia ε.

În mod concentrat criteriul se exprimă prin implicația

$$|r_c^{\alpha} - \tilde{r}_c^{\alpha}| \leq \varepsilon \quad \rightarrow v \in C^{\alpha}$$

Dacă, în plus, toate programele care au condus la gradul r_c^{α} satisfac această condiție, atunci este valabilă echivalența

$$v \in C^{\alpha} \quad <-> \quad |r_c^{\alpha} - \tilde{r}_c^{\alpha}| \leq \varepsilon$$

Valoarea ε o numim **deviație de clasă**, iar modulul diferenței de mai sus, $|r_c^{\alpha} - \tilde{r}_c^{\alpha}|$ - **normă de apartenență de clasă.**

(5.29) Observații :

5.29.1. Ipoteza că programul v este un program relevant pentru clasa extinsă $\tilde{C}^{\alpha}$ este instrumentală, fiind absolut necesară pentru a se putea calcula gradul de reutilizare $\tilde{r}_c^{\alpha}$.

5.29.2. Limita ε este de factură pur subiectivă (ținând de fapt de caracterul "fuzzy" al majorității noțiunilor din sfera de analiză a lucrării de față și este adoptată de proiectant, spre exemplu, în funcție de perspectiva dezvoltării clasei (acest criteriu având evident un caracter pronunțat subiectiv).

5.29.3. Programul v poate sau nu să fie considerat în continuare relevant în clasa C^{α}, dacă a fost stabilită apartenența programului la clasa respectivă ; criteriile de stabilire a acestui specific al programului v nu au legătură cu criteriul (5.28) ci doar cu cele prezentate în cap. 4, la calculul gradului de reutilizare de clasă (§ 4.2).

Acest criteriu de apartenență a unei variante la o clasă, deși este un criteriu a posteriori, *este util pentru dezvoltarea în continuare a unei teorii consistente pentru proprietățile programelor în cadrul claselor de aplicații.*

5.6. Consistența claselor de aplicații

Caracterul de noțiune *primară*, al noțiunii de clasă, trebuie perceput în sensul că această noțiune este definită pur intuitiv, ca și o mulțime de programe caracterizate prin proprietatea că fiecare program poate fi obținut prin modificări ale unui program numit generic (definiția [Mit87]). Ambiguitatea poate fi parțial înlăturată introducând criterii de definire *a posteriori* a **consistenței** unei clase, care să postuleze condiții care, odată verificate, ne asigură că programele respective *merită* să aparțină clasei :

(5.30) Criteriu : *O clasă este* ***consistentă*** *relativ la deviația de consistență* ε_c *dacă oricare program aparținând clasei satisface criteriul (5.28)* :

$$\mathbf{C^{\alpha}\text{ - consistentă}} \Leftrightarrow \mathbf{C^{\alpha}} = \{ v_i \in \mathbf{C^{\alpha}}, i \in I^{\alpha} \mid r_c^{\alpha} - \tilde{r}_c^{\alpha} \leq \varepsilon_c \}$$

Prin calcul se poate determina deci foarte simplu, odată definite toate elementele mulțimii de programe ale unei clase, în ce măsură putem acorda mulțimii respective înțelesul *autentic* de clasă.

Să presupunem că o clasă este **consistentă** relativ la *deviația de consistență* ε_1. Dacă dorim să verificăm consistența clasei relativ la o *deviație de consistență* $\varepsilon_2 < \varepsilon_1$, putem să găsim un număr de programe ale clasei, pentru care criteriul de consistență este satisfăcut. Numim această submulțime, partea **tare consistentă** a clasei, relativ la ***deviația de consistență*** $\in_2$.

Pentru a defini un criteriu de omogenitate a clasei, să definim **vecinii** unei variante. Pentru aceasta, să reluăm prin reprezentarea din Fig. 5.6 imaginea conturată în Fig. 5.2, reprezentând spațiul *calităților*. Fie P_v punctul corespunător unei variante fixate, **v**, și mulțimea *calităților* definitorii ale variantei **v**, reprezentată de coordonatele punctului P_v. Evident, în rețeaua determinată de coordonatele de pe axele *calităților*, acest punct are *vecini* (notați în Fig. 5.6 cu $P_v^{(i)}$). Însă în rețeaua amintită, nu toate punctele sunt "ocupate" de variante existente. Mai mult decât atât, în spațiul *calităților* pot să existe "goluri" (v. comentariile referitoare la Fig. 5.2) corespunzând unor combinații interzise de *calități*. Să selectăm din mulțimea punctelor vecine, acele puncte care corespund unor variante din clasa analizată (aceste puncte sunt marcate prin îngroșare pe Fig. 5.6). Variantele care corespund acestor puncte, le numim **vecinii** variantei **v**. În continuare, indexăm notațiile inclusiv cu un index de program arbitrar, **k**. Notăm în continuare prin $\varepsilon_{o,k}$ maximul dintre modulele diferențelor dintre gradul de reutilizare de program al variantei v_k (notat cu $r_{p,k}$) și gradele de reutilizare de program ale tuturor vecinilor variantei (vecini notați cu $r_{p,k}^{(v),i}$, unde $i \in I^{\alpha}_{(v),k}$ - mulțimea indicilor vecinilor, $I^{\alpha}_{(v),k} \subset I^{\alpha}$) :

$$\varepsilon_{0,k} = \max_{i} \left| r_{p,k} - r_{p,k}^{(v)j} \right| \qquad , \text{cu } i \in I^{\alpha}_{(v),k}$$

Numim această valoare, **normă de omogenitate locală** a variantei v_k.

Să enunțăm în aceste condiții, un :

(5.31) Criteriu : *O clasă este **omogenă** relativ la **ecartul de omogenitate** ε_0 dacă oricare program v_k al clasei ($k \in I^{\alpha}$) are norma de omogenitate mai mică sau egală cu ε_0 :*

$$C^{\alpha} \text{ - omogenă } <=> C^{\alpha} = \{ v_i \in C^{\alpha}, i \in I^{\alpha} \mid \varepsilon_{0,k} \leq \varepsilon_0 \}$$

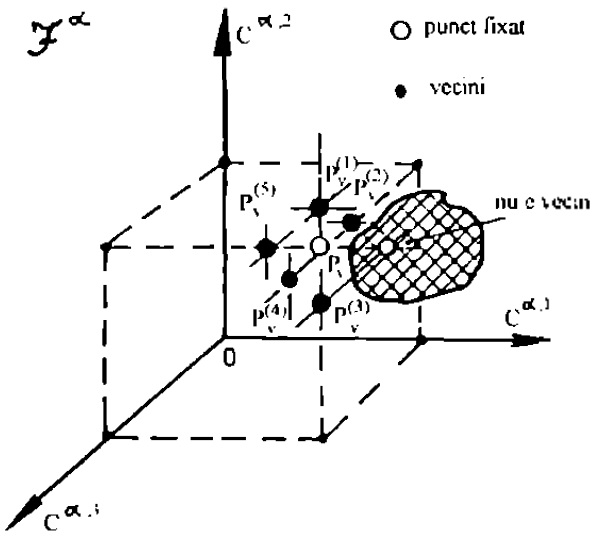


Fig. 5.6. Vecinii unei variante

Alegerea valorii ε_0, numită de noi **ecart de omogenitate**, are un caracter subiectiv exprimând opțiunea celui care întreprinde o analiză a proprietăților clasei (α), în ceea ce privește stabilirea limitelor de variație ale *normei de omogenitate locală*. Cu cât această valoare este mai mică, cu atât clasa respectivă este mai **omogenă** în mod *absolut*. Caracteristica de omogenitate reflectă de fapt proprietatea unei clase de a avea caracteristici uniforme sau lent variabile din aproape în aproape

Să definim situația în care o clasă este consistentă și omogenă în același timp

5.32) Definiție : *O variantă v_k ($k \in I^{\alpha}$) aparține **de drept** unei clase (α) relativ la **deviația de consistență** ε_c, având **ecartul** de omogenitate ε_0, dacă :*

*- clasa (α) este consistentă relativ la **deviația de consistență** ε_c :*

*- **norma de omogenitate locală** a variantei v_k este mai mică sau egală cu un **ecart de omogenitate** prestabilit :*

$$\varepsilon_{0,k} \leq \varepsilon_0$$

Să definim o proprietate care este mai restrictivă decât cele de mai sus :

(5.33) Definiţie : *O clasă este* ***super-consistentă*** *dacă toate variantele sale îi aparţin* ***de drept*** *relativ la* ***deviaţia de consistenţă*** ε_c *şi* ***având ecartul de omogenitate*** ε_o.

O clasă **super-consistentă** este în esenţă o clasă de aplicaţii având *calităţi* suficient de fin cuantificate pe nivele şi variantele clasei acoperă cât mai multe astfel de nivele, astfel încât aceste variante să aibă din aproape în aproape, proprietăţi foarte asemănătoare. O astfel de clasă este compusă din programe între care tranziţiile produse prin proiectare pentru reutilizare pot fi uşor realizate.

În acest paragraf am definit condiţiile pe care trebuie să le îndeplinească o clasă pentru a fi consistentă. Noţiunii i-a fost conferit un înţeles original, adaptat specificului domeniului abordat. De asemenea, am definit noţiunea de omogenitate a unei clase, ca fiind o proprietate care caracterizează uniformitatea de proprietăţi a clasei. Totodată am definit o serie de parametri (indicatori) necesari pentru aprecierea condiţiilor de consistenţă şi omogenitate : deviaţie de consistenţă, ecart de omogenitate, normă de omogenitate locală.

5.7. Nucleul unei clase

S-a văzut în § 5.5 că verificarea apartenenţei unui program la o clasă presupune desemnarea unor programe ca fiind *relevante*. Atributul introduce un criteriu subiectiv, suplimentar, relativ la relaţiile între membrii clasei şi la proprietăţile acestora. Aceste aspecte subiective pot fi relativizate ca în paragrafele precedente, unde ne raportăm la deviaţia de clasă. Propunem în continuare un criteriu pentru verificarea apartenenţei unui program la submulţimea programelor "relevante" din cadrul clasei bazat pe introducerea unui nou parametru ε_m , numit **eroare de merit :**

(5.34) Criteriu : *Un program este* ε_m *-* ***relevant*** *dacă acesta satisface criteriul (5.28) pentru norma de apartenenţă de clasă* ε *şi în plus modulul din criteriul (5.28) este mai mic sau egal cu* $\varepsilon_m + \varepsilon$.

Numim valoarea ε_m, **eroare de merit**. Dăm erorii de merit următoarea semnificaţie: în cadrul unei clase, caracterizate de deviaţia de consistenţă ε_c , putem desemna un set de programe cu atribute speciale, în sensul că sunt cele mai apropiate de programul "caracteristic" al clasei, adică programul "mediu", "tipic". Programele relevante sunt cele care, selectate din

programele clasei, dictează proprietăţile esenţiale ale întregii clase. Sunt deci cele mai *reprezentative* programe din clasă, dar relativ la o valoare numerică subiectivă, *eroarea de merit* $\varepsilon_m < \varepsilon$. Pe de altă parte, aceste programe, dictând proprietăţile esenţiale ale întregii clase, trebuie să aibă o distribuţie suficient de largă (în limitele *erorii de merit*) pentru a putea cuprinde aceste proprietăţi (calităţi). Din acest motiv, programele acestea vor trebui să fie suficient de *deosebite* între ele, două câte două.

(5.35) Definiţie : *Numim* ***nucleu*** *al unei clase* C^{α} , *relativ la eroarea de merit* ε_m, *mulţimea de programe* ε_m **- relevante** *ale clasei, notată cu* $\mathbf{Ker}^{\varepsilon m}(C^{\alpha})$, *cu proprietatea* :

$$\mathbf{Ker}^{\varepsilon m}(C^{\alpha}) = \{v_i \mid v_i \in C^{\alpha},\ v_i, v_j \text{ sunt } \varepsilon_m\textbf{-relevante};\ |r_{p,i} - r_{p,j}| \geq \delta(\varepsilon_m),\ i,j \in I^{\alpha}\}$$

Se observă că programele din *nucleu* aparţin părţii de *consistenţă tare* a clasei, relativ la *deviaţia de consistenţă* ε_m.

Prin analogie cu unele probleme specifice din teoria sistemelor şi din algebra liniară, considerăm că *nucleul* trebuie să conţină mai multe programe decât numărul de *calităţi* din spaţiul calităţilor (adică decât *dimensiunea* spaţiului calităţilor).

Problema formulată, de definire a unei mulţimi de *programe relevante*, este esenţială pentru multe din definiţiile, proprietăţile şi algoritmii propuşi în Capitolele 4 şi 5 ale lucrării de faţă (calculul gradului de reutilizare de clasă, problema fundamentală a **SPC**, viteza de grup a proiectării, criteriul de apartenenţă al unui program la o clasă, criteriul de consistenţă, etc.). Întrucât definiţia propusă pentru nucleul unei clase, poate fi aplicată doar ca şi *criteriu de verificare*, nu şi pentru găsirea acelor programe care fac parte din nucleu, propunem următorul algoritm iterativ, pentru găsirea elementelor nucleului :

i) se desemnează o mulţime de programe ca şi programe ε_m - *relevante* (pe baza experienţei proiectantului) ;

ii) pe baza acestei prime selecţii a programelor de nucleu se calculează toate mărimile necesare pentru verificare (r_C, *norme de apartenenţă*, etc.) ;

iii) se verifică dacă programele fac parte din nucleu ; dacă nu toate programele fac parte din nucleu, se elimină cele care nu respectă condiţiile din definiţie, se introduc în setul studiat alte programe şi algoritmul se reia.

Revenim asupra relaţiei (5.11,a). Am specificat în § 5.4 că este necesară o tehnică instrumentală pentru efectuarea calibrării relaţiei amintite. Propunem o tehnică care se bazează pe trecerea în revistă şi evaluarea tuturor programelor nucleului, iar factorul de calibrare obţinut va fi considerat, prin extrapolare, valabil pentru întreaga clasă.

(5.36) ***Tehnică de calibrare*** *pentru calculul valorii factorului de calibrare* f^{cal} *din relația (5.11,a), pentru clasa* ***omogenă*** C^a *:*

a) Se calculează gradul de reutilizare de clasă r_c *pentru clasa restricționată la nucleul clasei* C^a, $Ker(C^a)$.

b) Se parcurg toate variantele membre ale nucleului și se înregistrează numărul de linii-sursă prin care fiecare variantă diferă de programul generic v_0.

c) Se calculează media $r_c^{a.(med)}$, *a numărului de linii-sursă ale fiecărui program al nucleului.*

d) Se calculează gradul de accesibilitate de clasă r_c^a.

e) Se determină factorul de calibrare prin relația (rezultată din rel 5.11,a) :

$$f^{cal} = \frac{r_c^{a.(med)}}{r_c^a} \qquad (5.18)$$

Tehnica de calibrare propusă este valabilă doar pentru clase *super-consistente*, întrucât doar acestea au proprietăți ale versiunilor din nucleu, foarte apropiate de ale celorlalte membre ale clasei, iar algoritmul propus realizează calcule doar asupra nucleului și extrapolează rezultatele la nivelul întregii clase.

Acest paragraf încheie prezentarea unui formalism cu caracter original, destinat descrierii proiectării pentru reutilizare în cadrul unei clase de aplicații prin definirea ***nucleului SPC*** *ca fiind o submulțime de programe având trăsături definitorii pentru clasa respectivă. Analogia cu teoria sistemelor și cu noțiuni similare din algebra liniară, este evidentă. Această analogie sugerează și numărul minim de programe ale nucleului ca fiind numărul de calități din spațiul calităților (dimensiunea spațiului calităților). Pe baza celor prezentate, este descrisă și o procedură de calibrare pentru gradul calibrat de accesibilitate de clasă.*

5.8. Concluzii asupra utilizării formalismului SPC

După cunoștința autorului, în literatură nu există o tratare *sistematică* a unei abordări formale a procesului de proiectare pentru reutilizare în cadrul unei clase de aplicații. Pentru a suplini acest gol, capitolul de față propune un **formalism** deschis pentru *modelarea,*

CAPITOLUL 6.

TEHNICI ŞI STRUCTURI DE PROGRAM PENTRU CREŞTEREA GRADULUI DE REUTILIZARE AL APLICAŢIILOR MICI DE CONDUCERE DE PROCES (TEHNICI DE SPEŢA A DOUA)

Câteva din situaţiile în care se scriu aplicaţii mici de conducere de proces în limbaj de asamblare sunt cele caracterizate prin .

- *resurse* hardware/software disponibile *reduse*,
- *lipsa unor compilatoare* care să genereze cod executabil în limbajul-maşină al calculatorului-ţintă,
- *cerinţe deosebite* ca şi performanţe.

Pe de altă parte, unul din obiectivele importante ale ingineriei programării vizează adaptabilitatea programelor, prin creşterea gradului de reutilizare al programelor. Pentru genul de aplicaţii de mai sus, **reutilizarea** se referă atât la posibilităţile imediate de refolosire pentru aplicaţii din aceeaşi clasă, cât şi la portabilitatea aplicaţiei. Înglobăm aici şi aspectele de *portabilitate* (care de obicei sunt tratate distinct) întrucât acest gen de aplicaţii rulează pe o mare varietate de structuri hardware şi din acest motiv *reutilizarea* presupune şi preocuparea pentru portarea micilor aplicaţii.

Pentru limbajele de nivel înalt au fost dezvoltate diverse modalităţi de creştere a gradului de reutilizare, inclusiv medii de programare sofisticate, cu posibilităţi de descriere grafică a aplicaţiei [Fre87][Bro90][[NIC92]. Programarea orientată pe obiecte (**POO**) este o altă modalitate de deplasare a efortului de proiectare de la specialist la utilizator, reprezentând nivelul cel mai înalt al reutilizării. Desigur, există o serie de tehnici simple şi pentru îmbunătăţirea reutilizării programelor scrise în limbaj de asamblare . folosirea structurilor de date şi a tabelelor pentru descrierea interfeţelor şi formatelor, parametrizarea, folosirea unor structuri de program similare monitoarelor, folosirea bibliotecilor de funcţii, etc. Limbajele de asamblare însă nu prezintă facilităţi pentru implementarea acestor tehnici, spre exemplu nu există posibilitatea folosirii explicite a structurilor de date. Aceasta nu înseamnă că principiile de proiectare pentru refolosire nu pot fi de loc implementate în structura programelor scrise în astfel de limbaje. Capitolul de faţă îşi propune să prezinte tocmai astfel de exemple, referitoare la modul cum se pot folosi - implicit - "*structuri de date*" în aplicaţii mici de conducere de proces, cum se poate astfel realiza decuplarea programelor de partea specifică aplicaţiei, parţial sau chiar (cvasi-) total. Acestea se numesc în accepţiunea clasificării (5.37) de la sfârşitul Cap.

5, **tehnici de speța a doua**. Astfel, prezentul capitol reprezintă o continuare directă, aplicativă, a capitolului precedent, în sensul că în cadrul său prezentăm câteva exemplificări pentru aceste tehnici de speța a doua, aplicate asupra unor situații de conducere de proces cu sisteme mici. Prin tehnicile și structurile de program, prezentate, proiectantul are la îndemână un set de instrumente simple prin care poate obține aplicații cu un grad ridicat de reutilizare software. Tehnicile prezentate sunt rodul experienței autorului și exemplele prezentate sunt implementate pe sisteme care funcționează în exploatare curentă.

*Astfel, vom prezenta exemple începând cu cele mai simple, care se referă doar la aspecte locale de refolosire, trecând spre altele, mai complexe, care presupun modificarea unor întregi aplicații doar prin schimbarea unor structuri de date, fără a modifica partea de cod executabil, sau efectuând doar modificări minime. De asemenea, vom prezenta și posibilități de extindere a unor module **BIOS**, astfel ca acestea să asigure majoritatea funcțiilor reclamate de microsistemele de măsură sau conducere de proces, simple. Exemplele prezentate se vor referi toate, la aplicații scrise în limbaj de asamblare.*

6.1. Structuri de program pentru reutilizare locală

*În acest paragraf prezentăm o posibilitate de a realiza o refolosire parțială de aplicații, prin câteva structuri de date specifice pentru refolosire, a căror instanțiere poate să adapteze părți de aplicație pentru o mare varietate de cerințe. Prezentarea demonstrează că autorul a căutat să extrapoleze o serie de concepte specifice **POO**, sub o formă utilizabilă în aplicații scrise în limbaj de asamblare.*

Un prim exemplu [Tiv87-B] se referă la

a) **generarea de mesaje pentru operator** în aplicații de telemecanică (dar nu numai), prin compunerea mesajului din mai multe texte selectate dintr-o bază de mesaje, conform valorilor dintr-o listă de parametri și construirea mesajului într-un buffer.

Descrierea structurii și conținutului mesajului este realizată exhaustiv de structura de date din Fig. 6.1.

Explorarea structurii este asigurată de o rutină reentrantă (pentru a putea fi folosită simultan de toate taskurile generatoare de mesaje) cu următorii parametri de intrare :

- numărul bibliotecii folosite (practic, specifică fiecărui task);
- pointer către lista de parametri de selecție (adică, valorile de intrare);
- pointer către bufferul de formare a mesajului.

Astfel, diferite taskuri pot să folosească un concept unitar : aceeași structură de date, același mod de generare de mesaje, aceeași subrutină pentru generarea mesajelor destinate driverelor de consolă și imprimantă. Ansamblul acestora este de fapt aspectul esențial al aplicabilității tehnicii respective, pentru creșterea gradului de reutilizare.

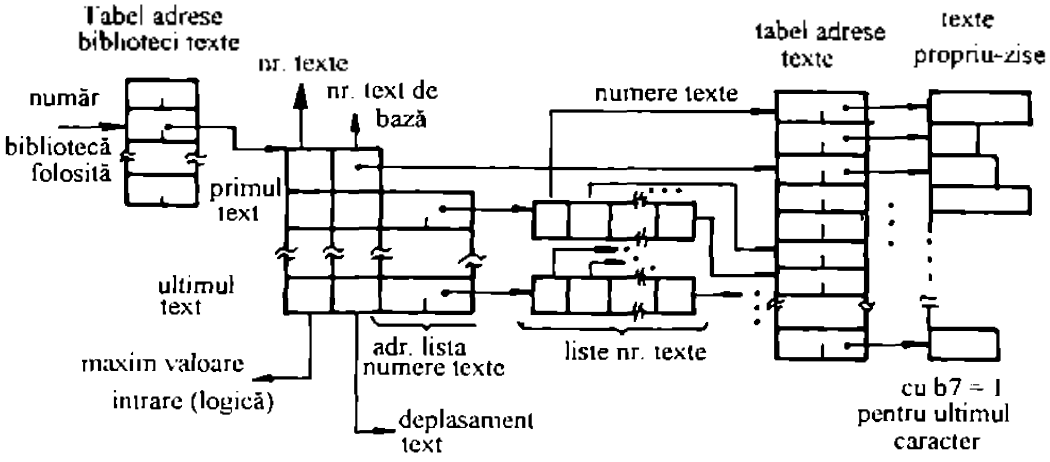


Fig. 6.1. Structura de date pentru o bază de mesaje

Structura din Fig. 6.1. este de fapt o colecție de *tabele de corespondență* pentru transformarea numerelor *logice*[1] din listele de parametri de intrare în șiruri de caractere (texte) selectate dintr-o structură unică de texte "elementare" și dispunerea controlată a acestora (prin deplasamentele precizate în structură). În plus față de facilitatea generării propriu-zise a mesajului dorit, structura și subrutina de generare permit și rejectarea parametrilor cu valori nepermise, prin faptul că structura conține și informația privind valoarea maximă admisibilă a fiecărui parametru (valoare care de fapt permite și determinarea lungimii listelor de corespondențe pentru fiecare parametru de intrare).

Aspectul de reutilizare este reprezentat de posibilitatea adaptării rapide la cerințele unei noi aplicații modificând doar structurile de date și rezultând texte și mesaje noi, conform cerințelor noului utilizator.

În perioada de proiectare a aplicației din articolul mai sus-citat, pentru pachetul de facilități tratat a fost proiectat un program pentru inițializarea structurilor de date, scris în limbaj de asamblare pentru **Z80** și care pe baza unui dialog cu proiectantul, construiește un fișier conținând codul-sursă pentru baza de mesaje. Cele 3 elemente : **structura** propusă, **subrutina**

---

[1] Aici, numărul logic are înțelesul de atribuire *convențională* a unui număr de ordine pentru o anumită valoare lorită a valorii de ieșire ; acest număr logic va fi utilizat ca valoare de intrare pentru a obține prin transformare valoarea dorită (în cazul nostru, un anumit șir de caractere).

care permite **utilizarea structurii** propuse (adică a bazei de mesaje) şi **programul de iniţializare** a structurii constituie într-un fel o încercare de "**încapsulare**" a datelor şi procedurilor de acces (deşi ultimul nu este propriu-zis un mecanism de iniţializare ca în **POO**) similar celui folosit în **POO**.

Subrutina şi un exemplu de iniţializare pentru structurile de date sunt prezentate în Anexa 3.1.

Al doilea exemplu [Tiv87-B] se referă la

b) modul de **generare a schemelor sinoptice** pentru aplicaţii mici de telemecanică utilizând microprocesor **Z80** (dar nu neapărat) caracterizat prin faptul că schemele generate sunt descrise prin coordonatele absolute ale punctelor de frângere şi eventual prin coordonatele colţurilor sud-vest ale unor scheme recursiv adăugate la schema de bază şi prin faptul că la acestea se adaugă mesaje fixe, mesaje cu stări (*afişat neafişat pâlpâitor*) respectiv entităţi numite *întrerupătoare* în poziţiile *închis deschis*.

Aceste facilităţi nu reprezintă prea mult, mai ales pentru un programator obişnuit cu facilităţile oferite de limbajele de nivel înalt. Dar pentru o aplicaţie mică, scrisă în limbaj de asamblare, simplitatea şi spaţiul mic de memorie, ocupat, reprezintă argumente suficiente. Structura utilizată este prezentată în Fig. 6.2.

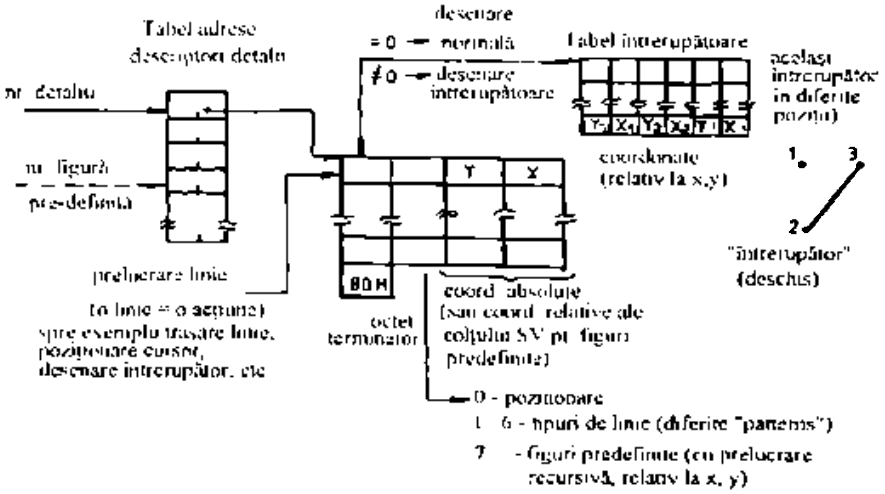


Fig. 6.2. Structura de date pentru generarea schemelor sinoptice

Se observă că din motive de simplitate *întrerupătoarele* sunt tratate distinct pentru diverse poziţii în care sunt incidente pe schema sinoptică (deci nu au fost prevăzute facilităţi grafice de rotire). Structurile de date pentru localizarea şi conţinutul textelor care se afişează pe schema sinoptică, sunt asemănătoare.

Subrutinele care utilizează aceste structuri sunt grupate într-o bibliotecă având câteva intrări accesibile utilizatorului. Legăturile dintre componentele bibliotecii şi punctele de acces sunt reprezentate în Fig. 6.3. Intrările accesibile permit trasarea unei scheme sinoptice (sau un

"*detaliu*"), tratarea tuturor *întrerupătoarelor* sau basculareа unui anumit *întrerupător*. Trasarea unei scheme (a unui "*detaliu*") se face apelând subrutina de trasare cu parametru de apel numărul schemei, toate schemele dintr-o aplicație fiind grupate în aceeași structură și fiind regăsite prin adresa dintr-un tabel de adrese. Caracterele generate sunt bufferizate (inclusiv caracterele de poziționare, atribute, etc.). Bufferul "grafic" este privit ca o resursă partajată, de aceea este protejată prin semafor, prin apelul subrutinelor **PSEND1** și **VSEND1** (care asigură, în afara inițializării, respectiv, eliberării pointerului și bufferului folosit, și apelul unor funcții de tip **P** și **V**, respectiv trimiterea unui pointer precum și a lungimii bufferului către driverul grafic, așteptând apoi confirmarea preluării mesajului).

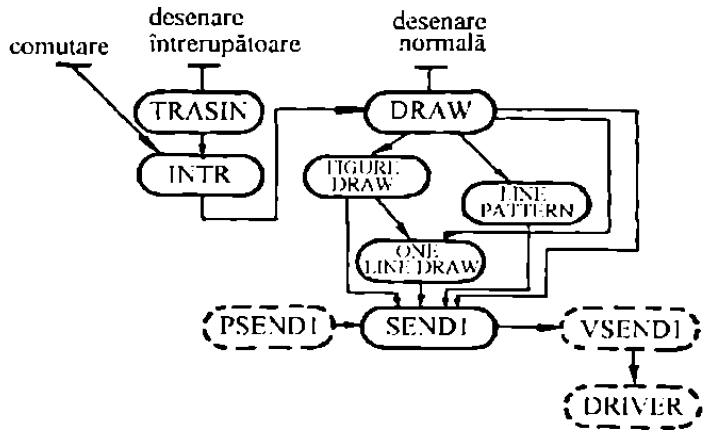


Fig. 6.3. Subrutinele de generare a schemelor sinoptice

Subrutinele respective însoțite de un exemplu de inițializare a structurilor grafice, sunt prezentate în Anexa 3.2 (în cele de mai sus, nu au fost prezentate subrutinele pentru texte).

Structura propusă, împreună cu subrutinele prezentate, de asemenea "încapsulează", ca în exemplul precedent, tot procesul generării de scheme sinoptice.

În sinteză, putem afirma că, deși limbajele de asamblare nu oferă posibilități de **POO**, într-o oarecare măsură, printr-o anumită disciplină de proiectare, se poate implementa, într-un mod elementar, ideea "*încapsulării*" *datelor și programelor într-un modul independent*, linkeditabil cu aplicația, cu intrări limitate ca număr. Subrutinele pot fi prevăzute cu mecanisme pentru restricții de acces (ca în primul exemplu). Nu se dispune însă, astfel (nici nu e necesar pentru dezvoltare de aplicații mici), de mecanisme de inițializare, decât prin preprocesare, ceea ce nu constituie mecanism de inițializare în sensul acceptat de **POO**.

În concluzie, acest paragraf prezintă câteva exemple simple privind o primă posibilitate de creștere importantă a posibilităților de reutilizare : utilizarea unor structuri de date care să ducă la o mai mare flexibilitate a unor fragmente ale aplicațiilor, astfel încât codul executabil

*corespunzător să nu depindă de condiţiile concrete ale aplicaţiei. În acest caz, structurile de date respective, asociate cu funcţiile care utilizează aceste structuri, constituie o imitare a unor principii de încapsulare ale **POO**. Acest principiu constituie un exemplu elementar relativ la modul de implementare a unor structuri de program, concepute conform clasificării (5.3°) - ca şi tehnici de speţa a doua.*

6.2. Intefaţă flexibilă cu operatorul

Un grad mai ridicat de abstractizare permite, în aplicaţia prezentată în continuare, descrierea prin structuri de date nu numai a formei de prezentare a interfeţei cu operatorul ci şi a secvenţei de funcţionare şi comunicaţie cu aplicaţia [Dia88]. Suportul hardware al aplicaţiei pentru care este concepută structura de program prezentată, este un panou de operare inteligent prevăzut cu afişaj alfanumeric şi tastatură şi comunicaţie serială cu calculatorul de proces care este gazda aplicaţiei propriu-zise de proces (un echipament de supraveghere a proceselor de aşchiere).

Interfaţa cu operatorul se referă la următoarele facilităţi :

- afişarea unor mesaje informative (spre exemplu, regimul de lucru) ;
- afişarea unor valori numerice ale mărimilor din proces ;
- afişare de mesaje *help* .
- introducerea unor valori numerice de către operator ;
- scanarea unor tabele ;
- selectarea unor opţiuni, etc.

Aceste facilităţi sunt asemănătoare celor asigurate de un "*spread-sheet*" clasic, doar că aici se lucrează în limbaj de asamblare cu puţină memorie disponibilă şi, în plus faţă de aceasta, prin proiectare s-a urmărit în mod special posibilitatea **refolosirii**. De aceea toate facilităţile prezentate, precum şi secvenţa de funcţionare, sunt codificate printr-o structură de date adecvată.

Funcţional, panoul poate fi privit ca un **automat secvenţial** (v. Proprietatea (5.9) din Capitolul 5). Trecerea de la un *lay-out* al afişajului la altul depinde de opţiunile operatorului şi de mesajele venite de la calculatorul de proces. De aceea, în funcţie de aceste reacţii ale mediului, se poate realiza "*legătura*" (**"link"**) cu un *lay-out* sau altul, legăturile fiind tabelate.

*Numim în cele ce urmează, **context de funcţionare**, sau pe scurt, **context**,* lay-out-*ul (aspectul afişajului, formatele de afişare, variabilele asociate valorilor numerice afişate,*

mesajele afișate, starea acestora, etc.) având asociată o anumită diagramă de tranziție *spre alte* contexte *(implicit cu alt* lay-out*), prin anumite perechi* "condiții de tranziție - lay-out următor"*, condiții dictate de* reacțiile operatorului *și de* mesajele *venite de la partenerul de dialog (calculatorul de proces).*

Unele mesaje de la calculatorul de proces sunt "normale" într-un context dat, provocând basculareа contextului, conform convențiilor tabelate. Altele sunt tratate ca și **derute**. Pentru a mări gradul de generalitate, structura de date înglobează și adresa unei eventuale secvențe utilizator (**user sequence**), apelată la un context dat.

Structura de date care asigură codificarea expusă mai sus, este prezentată în Fig. 6.4.

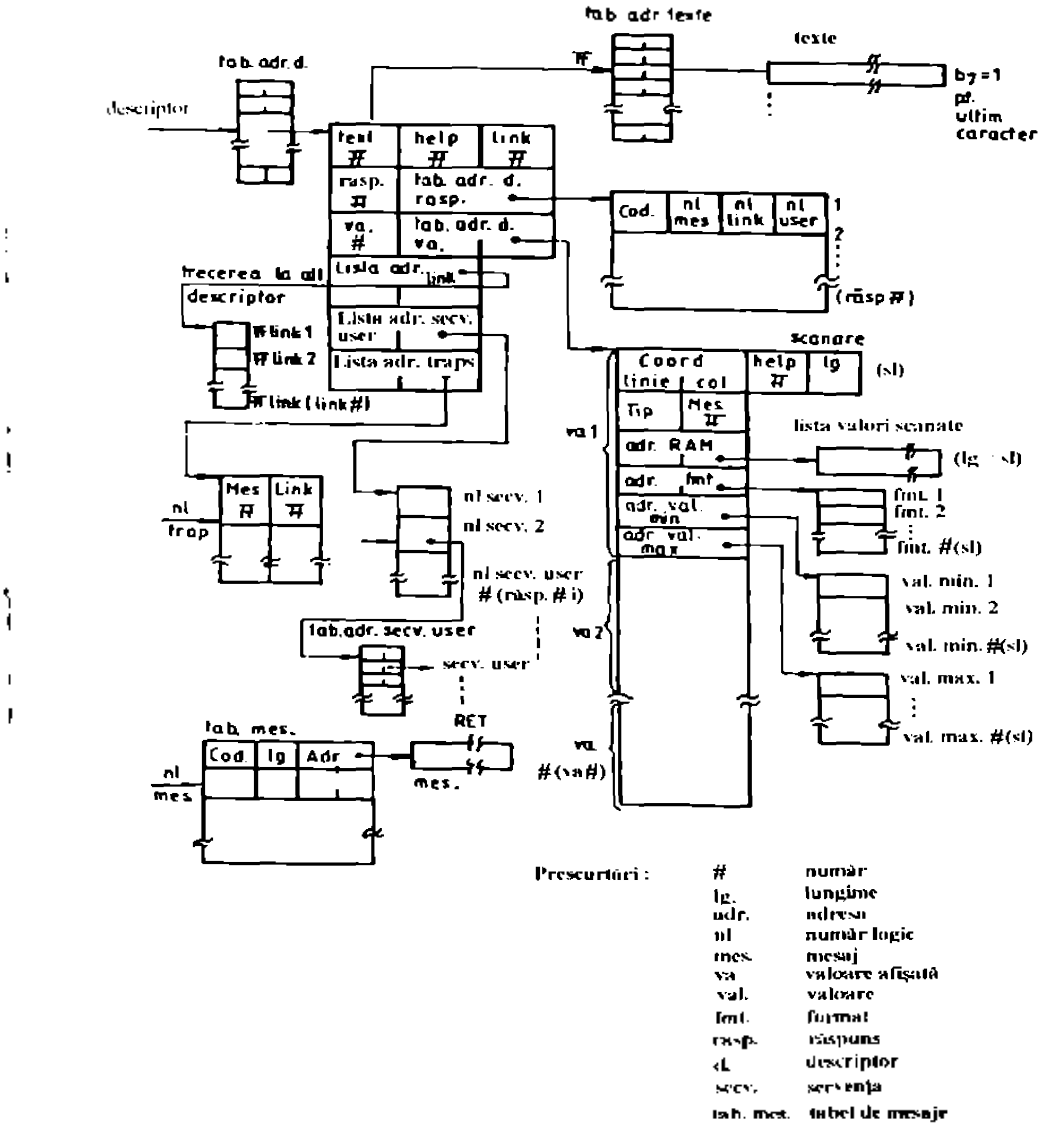


Fig. 6.4. Structura de date pentru interfața flexibilă

Programul care explorează structura de date inițializată pentru o anumită aplicație, este într-un fel un mic sistem de operare, care "*rulează*" programul panoului, adică diagrama de funcționare implementată prin structura de date prezentată. Bineînțeles, necesitatea simplității își spune cuvântul : în structura de date, formatele sunt codificate cât mai simplu și se referă doar la numere întregi sau în virgulă fixă cu un număr precizat de digiți, mesajele care vin de la partenerul de dialog au un antet de un octet care conține **codul** mesajului, etc. Însă, pentru a mări flexibilitatea, se folosesc convenții interne (**numere logice**) pentru a localiza mesajele, legăturile, secvențele utilizator, derutele. Pentru a transforma un număr logic în adresă fizică, se folosesc **tabele de adrese**. Practic, structura de date prezentată asigură o **flexibilitate** absolută pentru reconfigurarea aplicației, bineînțeles *în limitele clasei sale*. Asta înseamnă că prin modificarea structurii de date se obține o aplicație cu totul nouă, care poate diferi chiar foarte mult de cea precedentă. Diferența este formală și nu structurală, anume, doar din punctul de vedere al utilizatorului, deoarece, de fapt, este vorba în toate cazurile, așa cum s-a arătat, de un *spread-sheet*, având implementat în plus un mecanism de *automat secvențial* cu *condițiile de tranziție conectate cu reacțiile operatorului și cu mesajele recepționate pe unitatea serială*).

Modificarea structurilor de date permite definirea unor noi aplicații de tip panou de operare inteligent, extrem de variate ca aspect exterior. Dacă totuși există *cerințe neprevăzute*, nesuportate de informația conținută în structurile de date prezentate mai sus, rezolvarea acestor cerințe este lăsată în seama secvențelor utilizator.

Pentru a mări utilitatea celor prezentate și în condițiile în care ar exista suficientă cerere pentru aplicații din această clasă, se poate proiecta un **preprocesor** care să genereze prin dialog cu utilizatorul, structura conform cerințelor. Astfel, în foarte scurt timp, se pot realiza panouri de operare extrem de diferite.

*Prin asocierea dintre structura de date prezentată, programul care realizează accesul la această structură (exploatarea structurii) și preprocesor de inițializare a structurii, realizăm din nou implementarea conceptelor **POO** (v. Cap. 2) într-o formă simplificată, dar mai sofisticată decât în paragraful precedent, având un grad mai înalt de abstractizare.*

O primă implementare a principiului de proiectare și a structurilor de date prezentate a fost realizată în cadrul a două echipamente de supraveghere a proceselor de așchiere, proiectate și realizate la **IPA Timișoara** [Sil88]. Un alt exemplu este prezentat în Anexa 3.3. Aici este prezentată instanțierea structurilor de date pentru panoul de operare al unui *echipament de comandă numerică pentru tăiere cu flacără* **NUMEROM 633** [Pop89]. Panoul de operare este un microsistem cu microprocesor **Z80**, având comunicație serială **RS-232** cu microcalculatorul central. Structura hardware este deci din categoria celor pentru care a fost concepută structura de date prezentată. De aceea s-a folosit principiul de proiectare prezentat, structura de date descrisă și programul care utilizează această structură de date. Proiectarea noului program a fost

mult scurtată folosind cele prezentate mai sus.

Modalitatea de reutilizare descrisă în acest paragraf reprezintă un nivel de reutilizare mai ridicat decât tehnicile de programare prezentate în primul paragraf, întrucât prin structuri de date se implementează adaptarea programului la condiţiile concrete nu numai pe o porţiune restrânsă a aplicaţiei, ci pentru întregul context de funcţionare. Modalitatea de reutilizare prezentată, este adecvată pentru implementarea unor automate secvenţiale, precum panoul de operare prezentat.

6.3. Decuplarea totală a programelor de condiţiile specifice de funcţionare, în cadrul unei clase

În lucrarea [Tiv87-N] este prezentată o *clasă de aplicaţii* de telemecanică (în sensul din [Mit87]). Pentru a creşte **viteza de grup a proiectării** (definiţia (5.26)) - adică viteza medie de trecere, în cadrul clasei, de la un membru la altul - prin proiectare se poate realiza un program care să funcţioneze pentru oricare membru. Particularizarea (**adaptarea**) pentru *structura hardware* şi pentru *volumul şi structura informaţiei din proces* şi *forma interfeţei cu operatorul* se realizează tot prin structuri de date, ca şi în paragrafele precedente.

Pentru a realiza cele propuse, s-au folosit câteva tehnici simple, dar eficiente [Tiv87-B], dintre care amintim :

- folosirea **structurilor de date** prezentate la paragraful 6.1 pentru generarea mesajelor şi a schemelor sinoptice (şi a altor structuri asemănătoare) ;
- **adresarea indirectă** pentru zonele de date de mărime dependentă de volumul informaţiei din proces ; concentrarea acestor adrese în tabele de adrese plasate la adresă cunoscută ;
- **utilizarea codurilor intermediare** ("*numere logice*") pentru diverse situaţii (spre exemplu pentru generarea mesajelor către driverul de consolă) ;
- folosirea unui modul **BIOS** pentru adaptarea la nivelul fizic, acesta poate fi generalizat inclusiv pentru porturi de **I/E** şi pentru I/E analogice ;
- **organizarea schimbului de informaţii** staţii - dispecer, de o manieră cât mai generală (bineînţeles, în limitele clasei de aplicaţii respective) ;
- **generarea rapoartelor** pe baza unor structuri specifice de descriere a formatelor şi a folosirii unor **tehnici de compresie** (prin folosirea recursivă a unor facilităţi de salt în procesul de expandare, pe o lungime precizată, şi a parantezelor pentru acţiuni repetitive, cu factor de multiplicare în faţa parantezei - metodă asemănătoare reprezentării folosite în

formatul *.PCX de memorare a imaginilor [Mat'96]).

- **folosirea indicatorilor de lungime** pentru listele cu mărime dependentă de proces şi comasarea acestora în *zone* de adresă cunoscută .

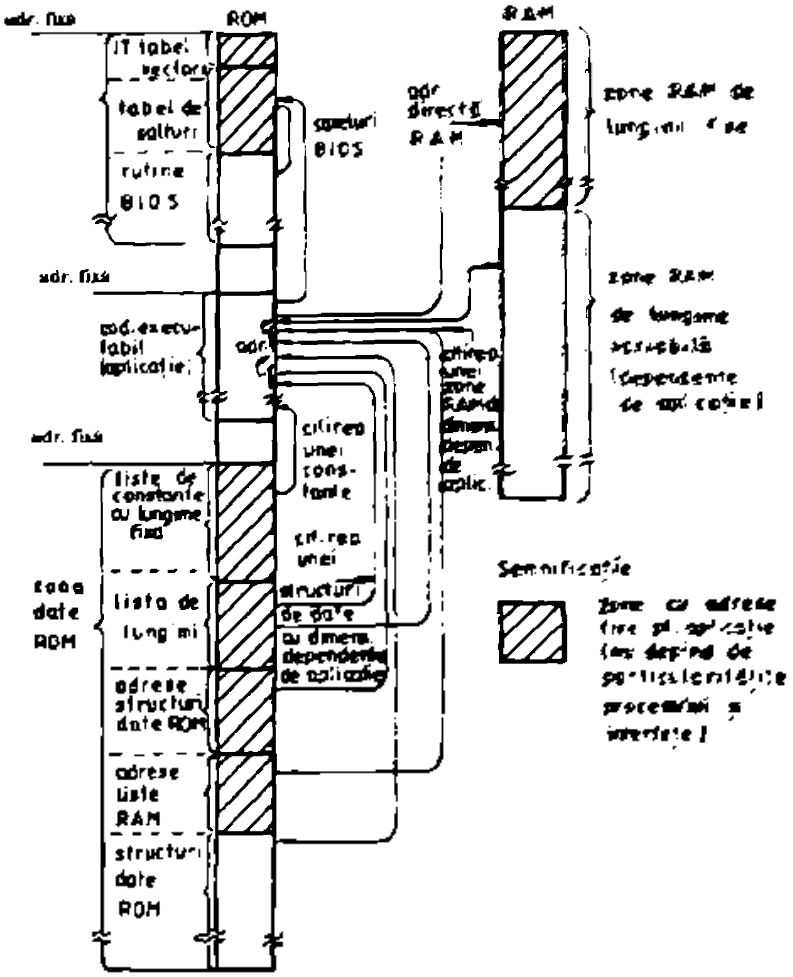


Fig. 6.5. Decuplarea părţii de cod executabil de partea de date depinzând de aplicaţie, în cadrul unei clase

Scopul principal al acestor tehnici este ca *partea fixă, de cod executabil*, să nu conţină decât *referiri la adrese absolute*, astfel nemaifiind necesară reasamblarea întregii aplicaţii. Ca rezultat al acestor precauţiuni, **realizarea unei noi aplicaţii** din clasa respectivă înseamnă *scrierea unui nou modul BIOS* (conform convenţiilor stabilite la proiectarea programului de aplicaţie) şi *modificarea structurilor de date* (descrierile de mesaje, scheme sinoptice, interfeţe cu operatorul, rapoarte, descriptori de structuri de informaţii din proces, etc.). Astfel, la memoriile fixe conţinând programul executabil nemodificat se mai adaugă câteva capsule de memorie conţinând noul BIOS şi noile structuri de date şi aplicaţia nouă este disponibilă, deoarece programele au fost deja testate. Efortul de implementare este astfel redus la minimum. Structura generală a programelor va arăta deci ca în fig. 6.5.

Deci, prin combinarea diverselor tehnici de programare prezentate, se pot realiza aplicații de conducere de proces aparținând unei ***clase de aplicații****, astfel încât codul executabil să nu depindă de aplicație decât în mică măsură (numai funcțiile* ***BIOS****). Adaptarea aplicației va fi realizată de structuri de date, prin diverse tehnici simple, propuse mai sus. Modalitatea aceasta de rezolvare a problemei reutilizării în cadrul unei clase de aplicații are toate atributele unui* ***SGBS****, așa cum a fost prezentat în Cap. 1, respectiv, constituie o abordare evoluată de tehnică de speța a doua (Clasificarea (5.3°)), reprezentând un* ***SPC (Definiția (5.3))****. Acesta conține în sine în stare latentă posibilități de generare de aplicații, întrucât toate informațiile structurale necesare reproiectării aplicațiilor în cadrul clasei se rezumă la structurile și tehnicile enumerate mai sus, iar nu la structuri de programe (acestea din urmă, cu excepția modulului* ***BIOS****, sunt fixe pentru toate programele clasei). Utilizatorul consumă un efort minim de proiectare, care constă în adaptarea (instanțierea) structurilor de date conform datelor (specificațiilor) proprii ale procesului pentru care dezvoltă aplicația. În plus, deoarece programele diverselor variante sunt cvasi-identice (mai puțin modulele* ***BIOS****) iar conform modalității de calcul descrise în § 4.2 doar structurile de program intervin în calculul acestui grad de reutilizare,* ***gradul de reutilizare de clasă*** *în acest caz este foarte ridicat.*

6.4. Un modul BIOS extins pentru sisteme de instrumentație cu microcontrollere din familia 8051

În paragraful precedent am arătat că folosirea unui modul **BIOS** crește portabilitatea aplicației, de altfel fiind o metodă unanim agreată de toți producătorii de echipamente de calcul. Nu același lucru se poate spune despre micile sisteme de achiziții folosite curent fie în instrumentație, fie în aplicații locale sau distribuite de supraveghere și conducere de proces. Aici nu există un standard din simplul motiv al extrem de marii varietăți de posibile structuri hardware și de cerințe ale diferitelor aplicații. Se pot însă deduce necesități comune pentru o mare varietate de situații de genul amintit mai sus. Pentru aceasta, întâi vom defini unele **cerințe pentru membrii clasei** la care ne referim :

- **structura** sistemului este cea din Figura 6.6 ;
- microsistemul are **intrări** din proces (de tipul : intrări analogice, intrări numerice, intrări de contorizare, **întrerupere externă**) ;
- sistemul are **ieșiri** de comandă a procesului (de tip analogic, numeric, PWM) ;

- sistemul este prevăzut cu o **legătură serială**, pentru comunicarea cu un eventual sistem ierarhic superior ;

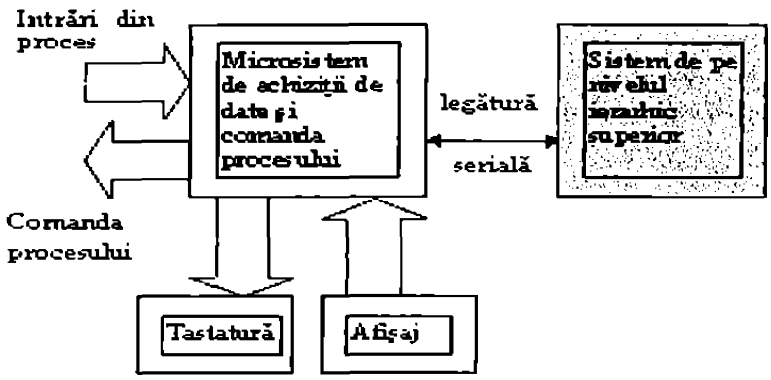


Fig. 6.6. Structura sistemului pentru clasa de aplicații de instrumentație

- funcționarea sistemului decurge după un **algoritm de monitorizare a comenzilor** venite prin portul serial, comenzi care se referă la :
  - **programarea parametrilor** de prelevare a mărimilor de intrare ;
  - **comanda unor acțiuni** de prelevare a mărimilor de intrare ;
  - **cereri de raportare** a rezultatelor citirilor din proces;
  - **controlul** schimbului de informații ;
- sistemul este prevăzut cu facilități pentru punere în funcțiune, depanare și testare, hard și soft, de tip **monitor serial** ;
- cerințele pentru sistem presupun existența unor **servicii** realizate prin software, care nu sunt puse la dispoziție direct de hardware-ul disponibil :
  - măsurarea unor **intervale de timp**;
  - **lansarea întârziată** a unor secvențe de program definite de utilizator;
  - **lansarea periodică** a unor secvențe definite de utilizator;
  - instalarea unor **rutine utilizator** pe întreruperile sistemului;
  - instalarea unei **aplicații utilizator** care este lansată automat la punerea sub tensiune;
- sistemul este prevăzut cu o **periferie minimală** alcătuită dintr-o tastatură și un afișaj.

Pentru a defini un modul **BIOS**, trebuie să adăugăm o trăsătură în plus, anume tipul de microcontroller folosit. Fără aceasta, putem defini funcțiile **BIOS**, dar nu putem să le realizăm, să le proiectăm, întrucât lipsesc elementele de nivel fizic care de fapt trebuie puse în legătură cu utilizatorul, prin intermediul funcțiilor **BIOS**.

Pentru a realiza proiectarea modulului **BIOS** pentru sistemul ale cărui cerințe au fost enumerate mai sus, întâi vom realiza o clasificare a funcțiilor **BIOS** pe grupe de servicii, apoi vom rafina pe fiecare nivel cerințele, le vom dezvolta, apoi vom particulariza toate acestea pentru microcontrollerul și limbajul de asamblare folosit. Ultima etapă este adaptarea la

structura concretă a microsistemului, prin modificarea echivalărilor adreselor specifice sistemului concret, respectiv prin mici schimbări necesare în program, în funcție de anumite particularități ale hardware-lui.

Deci, în primul rând, în cele ce urmează, propunem următoarea delimitare pentru **grupele de funcții BIOS** pentru clasa de mai sus :

a) funcții generale :

- inițializări : funcții de inițializare la cald, la rece, etc. ;
- folosirea legăturii seriale prin care sistemul comunică la nivelul ierarhic superior;
- funcții de monitor serial care să fie activate în lipsa existenței unei aplicații utilizator ;

b) funcții de intrări/ieșiri, specifice :

- deservirea tastaturii ;
- deservirea afișajului ;

c) funcții de instalare :

- instalări de rutine utilizator pe întreruperile sistemului, inclusiv a unei aplicații a utilizatorului care să fie lansată la punerea sub tensiune ;
- instalarea unui tabel de corespondență între comenzile transmise prin legătura serială și apeluri de funcții **BIOS** sau rutine utilizator ;

d) funcții de măsurare :

- servicii legate de măsurarea timpului ;
- deservirea intrărilor numerice ;
- deservirea directă a intrărilor analogice ;
- asigurarea unor servicii de achiziții de semnale analogice ;
- realizarea unor prelucrări primare ;
- deservirea intrărilor de contorizare ;
- măsurarea poziției ;
- măsurarea turației ;

e) funcții de ieșire :

- ieșiri analogice ;
- ieșiri numerice ;
- ieșiri **PWM** ;

f) funcții speciale :

- diverse raportări ;
- comunicare prin magistrală serială.

Funcțiile de mai sus se pot divide în **funcții standard**, folosite și la alte categorii de microsisteme, și în **funcții specifice**, prezente doar în sistemele care prezintă schimb de informații cu un proces efectiv (acele funcții care fac apel la acest schimb de informații). Pe de

altă parte, putem deosebi:

- **funcţii puternic cuplate** cu structura fizică specifică (spre exemplu funcţiile de lucru cu tastatura şi afişajul, unde putem avea o mare varietate de soluţii hardware) ;
- **funcţii cuplate normal** (majoritatea funcţiilor **BIOS**, care depind de facilităţi standard, acestea determinând din acest punct de vedere o variantă redusă în structura sistemelor, cu atât mai mult cu cât acestea folosesc microcontrollere, care înglobează o bună parte a facilităţilor hardware necesare acelor facilităţi standard) ;
- **funcţii slab cuplate** (spre exemplu funcţiile de prelucrare primară, raportările etc.).

Aşa cum se poate concluziona din enumerarea de mai sus, în viziunea noastră un modul **BIOS** pentru un asemenea sistem trebuie să aibă un **set extins de funcţii**, cuprinzând inclusiv funcţii specifice de măsurare, care în mod normal nu sunt cuprinse în modulele **BIOS** standard. Numim acest gen, **modul BIOS extins**.

Etapa următoare constă în **rafinarea cerinţelor** în cadrul fiecărei grupe de mai sus, după cum urmează :

1. **Iniţializările** : urmărim aici definirea acelor funcţii care au rol de iniţializare, respectiv pornirea sistemului la rece (*cold boot*), pornirea la cald (*warm boot*), salt în monitorul serial etc. Aceste funcţii au menirea :
   - programării unor circuite,
   - iniţializării unor zone RAM folosite de funcţiile **BIOS** şi de monitorul serial,
   - iniţializarea stivei etc.

2. **Deservirea legăturii seriale** : aici desemnăm funcţii de emisie octet, recepţie octet, recepţie doar dacă este octet în bufferul de recepţie, recepţie informaţie organizată în formatul **HEX-INTEL** (*download*), emisia conţinutului unei zone de memorie în formatul **HEX-INTEL** (*upload*), implementarea unor protocoale de comunicaţie simple etc.

3. **Monitorul serial** : oferă facilitatea punerii la dispoziţia utilizatorului a unei interfeţe uşor de folosit pentru accesul acestuia la resursele microsistemului, la un nivel elementar, prin intermediul legăturii seriale. Operatorul are la dispoziţie un set minimal de comenzi referitoare la vizualizarea conţinutului memoriei, schimbarea conţinutului, afişarea conţinutului unor regiştrii interni, lansarea în execuţie a unui program cu eventuala inserare a unor puncte de oprire (*breakpoints*) etc. Modulul **BIOS** va fi astfel realizat încât în lipsa unei aplicaţii a utilizatorului, instalată printr-un apel **BIOS** special realizat în acest scop, la punerea sub tensiune se va rula o secvenţă de iniţializare la rece şi apoi se va face salt în monitorul serial (deci opţiunea *default* va fi chiar acest monitor serial).

4. **Deservirea tastaturii** : funcţiile **BIOS** pentru serviciile de deservire a tastaturii sunt dependente de elemente de structură hardware corespunzând unei mai mari varietăţi decât alte servicii **BIOS**, de aceea aceste funcţii trebuie mai degrabă rescrise la trecerea la noi aplicaţii, decât altele din categorii depedente de facilităţi de bază ale hardware-lui, prezente identic la mai mulţi membri ai clasei; aici înglobăm funcţia de citire cu aşteptare (echivalent cu funcţia **getch** din limbajul **C**) şi funcţia de test şi eventuală preluare a caracterului tastat (echivalent cu **kbdhit** din limbajul **C**).

5 **Deservirea afişajului** : funcţiile care vor fi definite în această categorie sunt în aceeaşi situaţie de puternică dependenţă fată de hardware, ca şi acelea de la deservirea tastaturii. Aici definim funcţii de afişare caracter, ştergere afişaj, poziţionare cursor, setare atribute de afişare etc. Dacă sistemul are şi alte elemente de afişare/semnalizare decât afişajul propriu-zis (spre exemplu **LED**-uri), atunci vor fi definite funcţii **BIOS** pentru activarea / dezactivarea acestor elemente de afişare şi setarea/resetarea unor atribute (spre exemplu *blinking*).

6. **Instalările de rutine utilizator** : prin transmiterea către sistem a adreselor unor rutine definite de utilizator, funcţii definite special în acest scop vor realiza apel automat al acestor rutine la fiecare apel al tratării secvenţelor de întrerupere pe care utilizatorul a dorit instalarea acelor rutine definite de el : prevedem funcţii de instalare pe întreruperea de ceas de timp real, pe întreruperea generată de sfârşitul conversiei analog-numerice (dacă există aşa ceva), pe întreruperea externă şi pe întreruperea generată de unitatea serială pe condiţia de terminare a unei acţiuni (recepţie sau emisie). De asemenea, o funcţie **BIOS** va determina instalarea unei aplicaţii a utilizatorului, care să fie apelată la punerea sub tensiune, după iniţializarea la rece. Astfel, utilizatorul dobândeşte un plus de flexibilitate în ceea ce priveşte utilizarea sistemului.

7. **Instalarea tabelului de control al aplicaţiei** : orice aplicaţie simplă de instrumentaţie în structura prezentată în figura 6.6 are o schemă logică simplă, construită pe următorul algoritm :

   1- iniţializare la punerea sub tensiune;
   2- recepţia unei comenzi;
   3- selectarea acţiunii dorite, pe baza comenzii recepţionate;
   4- tratarea comenzii, pe baza acţiunii selectate;
   5- raportare asupra rezultatelor obţinute în urma execuţiei comenzii (spre exemplu, trimiterea şirului de valori citite, dacă această comandă se referă la

pornirea unei achiziții);
6- salt la 2.

Evident, este necesară realizarea unei corespondențe între acțiunea dorită și comanda recepționată prin legătura serială. O funcție **BIOS** specială poate determina instalarea unui asemenea tabel de corespondențe, în care intrările să fie comenzile iar ieșirile, fie funcții **BIOS** (în care caz sistemul master trebuie să transmită parametrii de apel, după o anumită convenție, imediat după comandă), fie rutine utilizator. Monitorul serial prezentat mai sus trebuie să conțină o comandă prin intermediul căreia utilizatorul să poată instala acest tabel, de asemenea trebuie să aibă o altă comandă prin care aceasta aplicație simplă să poată fi lansată (comandă distinctă de cea a lansării unui program oarecare, aceasta fiind folosită mai ales în etapa de testare și punere la punct).

8. **Măsurarea timpului** : în majoritatea covârșitoare a situațiilor de măsurare, sunt necesare servicii legate de măsurarea ("cronometrarea") sub o formă sau alta, a unor intervale de timp, fie legate de repere exterioare (întreruperi externe) fie cerute de necesitățile măsurării (lansare periodică sau întârziată a unor acțiuni); vom defini în acest sens funcțiile de armare a unei temporizări, cu execuția întârziată a unei rutine utilizator, lansarea periodică a unei rutine utilizator, cu perioada indicată de acesta, cronometrarea unui interval de timp între două sau peste mai multe întreruperi externe etc. Secvențele utilizator pot fi definite a fi executate prioritar (chiar în secvența de tratare a întreruperii de ceas de timp real) sau neprioritar (după returul din secvență de tratare a întreruperii). În cazul măsurării unor intervale de timp între repere externe trebuie să definim funcții distincte de start măsurare și de prelevare a valorii măsurate, cu așteptarea sfârșitului măsurării (în buclă de *polling*) sau doar cu testarea terminării și retur.

9. **Deservirea intrărilor numerice** : se atribuie numere logice pentru toate porturile de intrare și funcțiile **BIOS** de intrare vor fi apelate cu numerele acestea transmise ca parametri de intrare; putem defini, pentru anumite aplicații, funcții care citesc un port doar dacă vreunul din biții de intrare a suferit o modificare.

10. **Deservirea directă a intrărilor analogice** : aici includem acele funcții primare prin intermediul cărora utilizatorul are acces la resursa "*intrări analogice*", respectiv inițializarea intrărilor analogice (stabilirea regimului de lucru), selectarea canalului pe care se face citirea, realizarea propriu-zisă a citirii unei intrări etc.

11. **Serviciile de achiziție analogică** : prin apelul funcțiilor de la punctul anterior, vom defini funcții complexe care, pentru *cicluri complete de achiziție* inițiază, conduc până la sfârșitul

operației, semnalează sfârșitul acțiunii, fiind de fapt programe de achiziție complete. Funcțiile din acest grup vor fi proiectate pentru a realiza inițializarea parametrilor de achiziție, startul ciclului de achiziție, așteptarea terminării achiziției, testarea sfârșitului achiziției, forțarea terminării achiziției (*abort*). Achiziția poate fi condusă de *ceasul de timp real* (deci în acest caz sistemul citește un semnal cu o anumită perioadă de eșantionare - *time-driven*) sau de *întreruperi externe* (*event-driven*). Funcțiile de inițializare vor include ca și parametri de apel, inclusiv specificarea modului de lucru.

12. **Prelucrările primare** : câteodată, specificațiile sistemului de instrumentație cer efectuarea unor prelucrări simple asupra semnalelor citite prin intrări analogice, prelucrări care să fie efectuate în timp real, imediat după prelevarea mărimilor; astfel, putem aici introduce funcții care să suprapună peste funcția de achiziție o filtrare prin medie alunecătoare [Ion82], utilizatorul indicând sistemului numărul de pași ai mediei. Alte prelucrări posibile sunt mediera sincronă, calarea față de zero (dacă convertorul funcționează în cod binar deplasat), corecția de traductor după o curbă dată, filtrarea după un algoritm de filtrare dat , etc.[Ion82][Rad79]. Toate pot fi incluse în serviciile standard pe care le asigură sistemul, deci în modulul **BIOS** extins.

13. **Deservirea intrărilor de contorizare** : fie că aceste intrări sunt folosite pur și simplu pentru a contoriza evenimente din proces, fie că sunt folosite pentru măsurarea mai exactă a unor intervale de timp (mai exact decât o poate realiza ceasul de timp real al sistemului), propunem includerea în modulul **BIOS** al sistemelor din clasa analizată, a câteva funcții **BIOS** pentru exploatarea facilităților oferite de structura hardware de contorizare. Astfel, o funcție realizează startul unei contorizări și o altă funcție (pereche) oprește contorizarea și raportează asupra valorii contorizate, în cazul contorizării evenimentelor din proces. Un alt set de funcții inițiază măsurarea unui interval de timp, prin aceea că asociază un anumit contor unui eveniment extern care generează întrerupere și pornește măsurarea, sincron cu prima întrerupere externă, iar o altă funcție asociată așteaptă terminarea măsurării și returnează valoarea contorului, sau doar raportează de starea acesteia.

14. **Măsurarea poziției** : aceasta se poate realiza fie folosind intrări de întrerupere pentru impulsurile de la un traductor incremental și contorizarea acestora prin program, fie folosind două intrări de contorizare. Utilizatorul are posibilitatea, prin intermediul apelului unei funcții **BIOS**, să instaleze una sau alta din strategiile de măsurare, iar apoi, prin apelul unei alte funcții **BIOS** are posibilitatea să citească poziția.

15. **Măsurarea turației** : dacă procesul are un ansamblu în mișcare de rotație iar pe acesta este prevăzut un traductor de poziție, folosind măsurarea de poziție se poate calcula turația (prin

trei metode : măsurarea timpului cerut de o anumită deplasare unghiulară, fixată, măsurarea deplasării într-un anumit interval de timp, sau măsurarea timpului între două impulsuri de traductor - de fapt variantă a primei metode). Funcţiile **BIOS** din această grupă realizează instalarea unui anumit algoritm de măsură, returnează la cerere valoarea acestei turaţii sau doar raportează despre stadiul măsurării.

16. **Deservirea ieşirilor analogice** : dacă sistemul are ieşiri de tip analogic către proces, vom include între funcţiile **BIOS** o funcţie care să realizeze selectarea ieşirii dorite (desemnate printr-un număr logic) şi să transmită acolo o valoare precizată.

17. **Deservirea ieşirilor numerice** : se pot adopta aici două strategii de abordare, prima care consideră ieşirile în configuraţia naturală, de 8 biţi/port, a doua, care consideră fiecare bit distinct, numerotat cu un anumit număr logic. Astfel, putem defini două categorii de funcţii **BIOS**, prima care consideră porturile de ieşire numerotate logic şi le accesează ca atare, spre exemplu sistemele cu microcontrollere din familia **PCB8XC552** [Phi95]), trebuie prevăzută o funcţie **BIOS** asemănătoare cu cea care deserveşte ieşirile analogice.

18. **Deservirea ieşirilor PWM** . dacă sistemul are în componenţă una sau mai multe astfel de ieşiri (spre exemplu sistemele cu microcontrollere din familia **PCB8XC552** [Phi95]) trebuie prevăzută o funcţie **BIOS** asemănătoare cu cea care deserveşte ieşirile analogice.

19. **Raportările** : utilizatorul trebuie să poată solicita, la nivelul legăturii seriale, raportări asupra stării sistemului la un moment dat şi asupra valorilor măsurate. De aceea, definim câteva funcţii **BIOS** de raportare, pe categorii de informaţii de stare sau rezultate ale diverselor măsurări.

20. **Deservirea magistralei seriale** : unele microsisteme [Phi95] sunt prevăzute cu magistrale I^2C pentru comunicare între sisteme sau cu unele periferice (spre exemplu, memorii seriale). În acest caz, vom prevedea funcţii **BIOS** de scriere, de citire, alte funcţii speciale (spre exemplu, ştergerea unei memorii) etc.

Prin existenţa unui modul **BIOS extins**, de complexitatea şi bogăţia celui descris mai sus, o aplicaţie mai simplă poate fi proiectată şi realizată foarte repede, întrucât aproape toate funcţiile specifice de măsurare şi interfaţă cu mediul există deja, utilizatorului nemairămânându-i decât să construiască o succesiune de apeluri de funcţii **BIOS**, cu un minim de cod între apeluri, corespunzând necesităţilor de adaptare la situaţii concrete. Acesta este primul pas înspre proiectarea unui metalimbaj, care să genereze aplicaţia dorită. De altfel, folosind facilităţile de macrodefinire ale limbajelor de asamblare, este posibilă realizarea unui mod

asemănător de definire a aplicației. Cu un sistem având acest modul **BIOS**, o aplicație de instrumentație rulând pe un calculator **PC** intră foarte ușor în dialog. Etapele parcurse de acest program sunt :

- trimiterea unei comenzi care să determine returul pe unitatea serială a prompterului monitorului serial (*"recunoașterea"* existenței sistemului la capătul legăturii seriale);
- *încărcarea aplicației* prin intermediul activării comenzii de *down-load* a monitorului serial;
- *lansarea aplicației și dialogul cu sistemul*, pe baza comenzilor care să apeleze funcțiile de măsurare și cererile de raportare.

Dacă sistemul este prevăzut cu o memorie de tip **flash-EPROM** sau cu **RAM-CMOS**, aplicația rămâne încărcată și deci la următoarea pornire a sistemului nu mai este necesară încărcarea. Dacă aplicația definită de utilizator este mai simplă, practic este suficient apelul funcției **BIOS** care instalează tabelul de corespondență dintre comenzi și apeluri de funcții **BIOS**, în care caz sistemul master trimite doar comanda monitorului serial care realizează această instalare, apoi comanda de lansare a aplicației induse de tabelul definit anterior.

În Anexa 3.4 este prezentat listingul unui modul **BIOS** extins cuprinzând un subset al funcțiilor prezentate mai sus.

Avantajele folosirii unui modul **BIOS** definit ca mai sus sunt legate în primul rând de rapiditatea dezvoltării de aplicații mici din clasa de sisteme de intrumentație sau supraveghere de proces, comandate prin legătură serială. În continuare este prezentat un exemplu foarte simplu de aplicație - un sistem de achiziții de semnale analogice - dezvoltat aproape integral prin apelul unor funcții **BIOS** din cele definite mai sus. Exemplul prezentat apelează la funcțiile definite în modulul din Anexa 3.4, deci nu apelează la facilitățile ansamblului mai sus prezentat, în particular la cea cu nr. 7. Se observă că proiectarea aplicației este rapidă și ușoară. Aplicația realizează monitorizarea comenzilor de la master și răspunde la două comenzi posibile : setarea parametrilor de achiziție (perioada de eșantionare, numărul de valori citite, canalul de citire) și start achiziție. Ca urmare a realizării achiziției, spre master va fi trimis un caracter care semnifică raportul, urmat de valorile citite. Dacă se primește spre interpretare o comandă necunoscută (ceea ce poate avea ca semnificație și o eventuală ieșire din secvența normală de comenzi), sistemul trimite spre master un cod de eroare și efectuează și un restart la "cald" în vederea eliminării unor posibile disfuncționalități ale microsistemului. În continuare prezentăm listingul aplicației :

```
;
;     Exemplu de program care foloseste functii de BIOS extins
;     ________________________________________________________
;

;
;     memorie interna folosita
;
nvalh data  70h         ;   contor numar de valori
nvall data  71h
snvh  data  72h         ;   salvare numar de valori
snvl  data  73h
poinh data  74h         ;   pointer depunere rezultate citire
poinl data  75h
stack data  60h         ; stiva
;
;     biti folosiţi
;
bitind data 26h         ; octet de indicatori pe bit
startb bit  bitind.0          ;  bit start achizitie
;
;     memorie externa folosita
;
zoncit data 1000h       ; zona unde vor fi rezultatele citirii
;
;     caractere de comanda
;
param data  'P'         ;  comanda "Parametrii de achizitie"
start data  'S'         ;  comanda "Start achizitie"
ercode data 'E'         ;   raport "Eroare"
raport data 'R'         ;   raport "Valori citite"

;
;     programul principal
;
begin:
      ld    a,#stack
      call  icboot              ;  initializare
mainlp:
      call  sgetch              ; citirea comenzii
      cjne  a,#param,mainl      ; nu sint setari de param.
                                ;  aici, setari de parametri de achizitie
      call  sgetch              ; citire per. esantionare high
      mov   R3,a                ; retinere high
      call  sgetch              ; citire per. esantionare low
      mov   R4,a                ;  retinere low
      call  sgetch              ; citire nr. valori high
      mov   nvalh,a             ; si retinere
      mov   snvh,a
      call  sgetch              ; citire nr. valori low
      mov   nvall,a             ;  si retinere
      mov   snvl,a
      call  sgetch              ;  citire canal
```

```
        call    msadco              ;  selectare canal
        sjmp    mainlp              ;   reluarea monitorizarii
main1:
        cjne    a,#start,main2      ;  nu e start
                                    ; aici, start
        mov     poinh,#HIGH(zoncit)     ;  initializare
        mov     poinl,#LOW(zoncit)      ;    pointer zona citire valori

        setb    startb              ;  setare bit start
        mov     a,#HIGH(citire)     ; instalare
        mov     R0,a                ;   rutina
        mov     a,#LOW(citire)      ;      de
        mov     R1,a
        call    tiperp              ;   citire
loopg:                              ; bucla de asteptare a sfirsitului conv.
        jb      startb,loopg        ; daca bitul e 1, nu e gata achizitia
                                    ;  aici, trimiterea valorilor citite
        call    thperp              ; dezinstalare rutina lansata periodic
        mov     poinh,#HIGH(zoncit)     ;  reinitializare
        mov     poinl,#LOW(zoncit)      ;   pointer valori citite
        mov     a,#raport
        call    sputch              ;  transmitere caracter raport
lsend:                              ;  bucla de emisie
        movx    a,@DPTR             ; extragere valoare high
        inc     DPTR
        call    sputch              ;  emisie high
        movx    a,@DPTR
        inc     DPTR
        call    sputch              ;  emisie low
        clr     C
        mov     a,svnl
        subb    a,#1
        mov     svnl,a              ; decrementare
        mov     a,svnh
        subb    a,#0
        mov     svnh,a              ;  in vederea contorizarii
        jnc     lsend               ;  reluarea emisiei daca mai sunt valori
        sjmp    mainlp              ; altfel se reia bucla principala

main2:                              ;  caz de eventuala iesire din secventa
        mov     a,#ercode           ;   (sau eroare de programare la master)
        call    sputch              ; emisie cod de eroare
        call    iwboot              ; reset soft
        sjmp    mainlp              ; si reluarea buclei principale

;
;       subrutina care realizeaza citirea
;
citire:
        mov     DPH,poinh
        mov     DPL,poinl           ;  init. pointer
        call    mradco              ; citirea convertorului
        mov     a,R0                ;  si
```

```
movx   @DPTR,a          ;    depunerea
mov    a,R1
inc    DPTR
movx   @DPTR,a          ;         rezultatului
inc    DPTR             ;   cu incrementarea pointerului
mov    poinh,DPH        ;  apoi retinerea acestuia
mov    poinl,DPL
clr    C
mov    a,nvall
subb   a,#1             ;  decrementare
mov    nvall,a
mov    a,nvalh          ;   numar
subb   a,#0             ;    valori
mov    nvlah,a          ;      de citit
ret    nc               ;  inca nu e gata sirul de citiri
clr    startb           ;  gata, deci se marchează terminarea
ret                     ; si retur
```

Este uşor de observat că programul realizat este de fapt o buclă infinită de program care asigură recepţia caracterelor trimise de master şi iniţializează parametrii de achiziţie sau determină declanşarea achiziţiei. Pentru realizarea propriu-zisă a achiziţiei a fost apelată facilitatea de instalare a unei subrutine a utilizatorului, care să fie lansată periodic, chiar cu perioada solicitată de master. Utilizatorul nu mai are de scris decât acea subrutină simplă, care să realizeze citirea propriu-zisă (folosind tot funcţii **BIOS**), depunerea rezultatului şi decrementarea unui contor de număr de valori. În exemplul prezentat, am marcat prin accentuare funcţiile **BIOS** extins, folosite.

*Din exemplul prezentat se observă că prin facilităţile oferite modulul **BIOS** prezentat simplifică mult proiectarea aplicaţiilor mici şi nu solicită proiectantului cunoştinţe extinse asupra facilităţilor hardware oferite de microsistemele cu microcontrollere. Multe alte cerinţe pentru sistemele informatice de măsură pot fi uşor implementate prin programe scurte şi simple.*

*În acest paragraf a fost prezentat un concept (**BIOS extins** pentru sisteme de instrumentaţie şi interfaţă de proces) care permite implementări practice. Micul exemplu prezentat subliniază uşurinţa realizării de programe folosind funcţiile **BIOS extins**, iar modulul **BIOS** prezentat în Anexa 3.4 este o implementare (parţială) a acestui concept. Implementarea tuturor funcţiilor de modul **BIOS extins**, definite în acest paragraf, pune la dispoziţia proiectantului un instrument puternic pentru realizarea rapidă a unor aplicaţii (relativ) complexe.*

*Descrierea unui modul **BIOS** specific sistemelor mici de conducere de proces este ultima etapă a descrierii unui **SPC** pentru aplicaţii mici, în completarea celor prezentate în paragraful precedent. Împreună, cele prezentate în cele două paragrafe permit abordarea mult simplificată a unui **SPC** pentru realizarea de aplicaţii mici de supraveghere de proces sau de instrumentaţie. Dacă cele prezentate în paragraful precedent descriu principiile de realizare a*

*unor structuri de program şi de date care să simplifice mult proiectarea unor aplicaţii precum cele descrise în acel paragraf, modulul **BIOS** extins prezentat aici este aspectul complementar care trebuie avut în vedere la realizarea proiectării în clasă, întrucât eventualele diferenţe de cod executabil sunt la nivelul acestor module **BIOS**.*

6.5. Concluzii asupra utilizării tehnicilor şi structurilor de program pentru creşterea gradului de reutilizare

În acest capitol au fost prezentate mai multe exemple de folosire a unor tehnici de programare specifice limbajelor de nivel înalt, în cazul unor aplicaţii mici de conducere de proces, scrise în **limbaj de asamblare**. Acestea au vizat :

- **creşterea gradului de reutilizare**, prin utilizarea de concepte care să implementeze în mod mult simplificat unele concepte **specifice POO** (ca şi în § 6.1),
- **descrierea întregii aplicaţii prin structuri de date** (§ 6.2) care să implementeze un **automat secvenţial** (definind aici şi un concept original - **contextul**) sau **separarea totală a codului executabil de datele specifice procesului** (§ 6.3),
- definirea unui modul **BIOS** specific aplicaţiilor din sfera de interes a acestei lucrări, modul care să crească viteza de realizare de noi aplicaţii (§ 6.4).

Toate acestea duc în primul rând la creşterea însemnată a productivităţii proiectării de noi aplicaţii.

*Tehnicile şi structurile prezentate permit implementarea unor **SPC** pentru automate secvenţiale sau pentru aplicaţii mici de conducere de proces şi se constituie în aplicaţii practice directe ale formalismului (definiţii, proprietăţi, modele) prezentat în Capitolul 5. Acolo unde a fost cazul, au fost făcute sublinierile de rigoare (spre exemplu, la § 6.2). De altfel, experienţa autorului, rezultată din practica proiectării programelor de conducere de proces pentru sisteme mici, reflectată (parţial) în acest capitol, a constituit unul din punctele de plecare pentru propunerea de formalizare prezentată în Capitolul 5. Astfel, conceptele teoretice din acel capitol îşi află un pandant firesc în prezentările de aplicaţii concrete din capitolul de faţă.. Cele prezentate aici sunt (câteva) aplicaţii specifice tehnicilor şi structurilor de program pentru creşterea gradului de reutilizare, numite **tehnici de speţa a doua**, în cadrul clasificării (5.37) de la sfârşitul Capitolului 5, constituind deci o direcţie de dezvoltare practică a conceptelor acelui capitol.*

Principalele concluzii finale care se impun sunt :

- prin utilizarea acestor tehnici se urmăreşte de multe ori implementarea sub formă simplificată a unor **concepte POO** ;
- aplicarea tehnicilor de programare avansate nu trebuie făcută decât **până la punctul în care aceasta este necesară** pentru clasa de aplicaţii la care se lucrează ;
- trebuie realizat un **compromis** acceptabil între *flexibilitatea* obţinută prin tehnicile amintite şi *complexitatea* programului (să nu se uite că se lucrează în limbaj de asamblare şi deci facilităţile oferite sunt mai puţine) ;
- proiectarea pentru reutilizare folosind aceste tehnici este **rentabilă** în situaţia în care se estimează o **cerere** reală de aplicaţii din acea clasă sau dacă efortul suplimentar nu este mare.

*Cele prezentate în Cap. 6 demonstrează existenţa unor premise pentru revigorarea folosirii limbajelor de asamblare pentru aplicaţii mici de conducere de proces. De altfel, există încă părţi de aplicaţii sau de software de bază (s. ex. modulele **BIOS**, v. § 6.5), sau chiar clase de aplicaţii care nu pot fi scrise decât în limbaj de asamblare.*

CAPITOLUL 7.
GENERATOARE DE APLICAȚII PENTRU SISTEME DE INSTRUMENTAȚIE

Atunci când utilizatorul comunică sistemului doar *cerințele* sale (prin intermediul unui *metalimbaj*, al unui *limbaj de specificații*, al unui *spread-sheet* clasic sau al unei *interfețe grafice*) și sistemul "elaborează" **automat** aplicația, fie sub formă de cod-sursă, fie direct sub formă de cod executabil, programul respectiv se numește **generator de aplicații (GA)**. În capitolul 1 am prezentat câteva exemple de generatoare de aplicații pentru anumite domenii, generatoare realizate la universități sau disponibile comercial. Evident, generatoarele de aplicații au în general un domeniu de acțiune restrâns la limitele unei **clase de aplicații** (v. cap. 1). Acesta este și cazul **GA** prezentate în acest capitol.

*Pentru unele domenii specifice conducerii de proces sau instrumentației, după cunoștințele autorului tezei în literatură nu sunt tratate exemple de generatoare. În compensare, capitolul acesta propune unele soluții, prin delimitarea, în primul rând, a trăsăturilor clasei de aplicații în cadrul cărora lucrează **GA** prezentate și apoi prin expunerea principiului de funcționare al **GA** și prezentarea acestora. Generatoarele de aplicații prezentate implementează o interfață conversațională cu utilizatorul final, prin intermediul căreia sunt transmise cerințele pentru programul care trebuie obținut. Pe baza acestora, **GA** construiesc codul executabil al aplicației dorite.*

7.1. Clasa de aplicații în care acționează GA

Un domeniu descoperit, în domeniul software-lui reutilizabil, este acela al *modulelor de achiziții construite cu microprocesoare sau microcontrollere* și legate cu un calculator **PC** prin *legătură serială*. Pentru partenerul de dialog respectiv aplicația de instrumentație, găzduită pe **PC**, există multe rezolvări configurabile (a se vedea în acest sens paragraful precedent). Însă, în bibliografia consultată nu a fost găsită nici o abordare, nici pentru tehnici de programare în domeniul de mai sus, nici pentru eventuale **GA**. Explicația e simplă: există o așa de mare diversitate de situații și tehnici de măsurare, precum și modalități hardware de rezolvare a problemelor de măsurare, încât e greu de delimitat o clasă de aplicații, în sensul din [Mit87], astfel încât să fie posibilă definirea fie a unui limbaj de specificații, fie a specializărilor în

sensul din lucrarea citată. În această a doua accepțiune, poate să devină excesiv, deci inacceptabil, efortul de analiză a situațiilor, în cazul în care granițele clasei sunt prea relaxate, adică paleta de posibilități e prea mare. În această sitație, un **GA** devine inoperant.

Pentru a realiza un **GA** viabil, este necesară delimitarea clară a domeniului său de acțiune. De aceea, în cele ce urmează schițăm *portretul-robot* al unei aplicații din clasa pentru care definim trăsături și de unde vor rezulta specificații pentru un **GA** [Sto94-H]:

- sistemul este construit cu microcontroller din familia **8051** (inclusiv **PCB 80C552** sau **PCB 83C552**) ;
- structura sistemului este cea din Figura 6.6, cu sau fără tastatură și afișaj ;
- sistemul are resurse de memorie externă - **RAM** și **ROM** - suficiente pentru a încărca codul executabil al aplicației, dar insuficient pentru un cod generat de un limbaj de nivel superior ;
- comunicația cu **PC** (pe care este instalat programul de instrumentație) se realizează prin legătura serială ;
- sistemul include, opțional, un monitor serial ;
- se pot realiza o serie de măsurători: de tensiune (prin **CAN**), de frecvență, de deplasare, de turație etc.;
- încărcarea programului de aplicație trebuie să se poată realiza din programul de instrumentație activ pe **PC**;
- parametrii regimului de lucru, natura măsurătorilor etc. pot fi comandați din programul de instrumentație activ pe **PC**.

Aceste atribute sunt asemănătoare celor luate în considerare în § 6.4. Rezolvarea problemei puse în acest capitol este conexă la soluțiile prezentate anterior.

Esențială pentru proiectarea unui GA pentru domeniul specific al unei clase de aplicații, este stabilirea exactă a trăsăturilor caracteristice ale ***clasei de aplicații*** *corespunzătoare. Obiectul paragrafului de față l-a constituit chiar această enumerare a trăsăturilor specifice ale clasei, respectiv descrierea condițiilor de funcționare, care definesc astfel o clasă de mici aplicații de instrumentație sau interfață locală de proces, construite pe un microsistem prevăzut cu posibilități de interfațare cu procesul, având legătură serială cu un calculator PC. Pe baza acestei descrieri, se poate elabora setul de specificații pentru un* ***GA*** *pentru această clasă.*

7.2. Problematica generării de aplicații pentru clasa precizată

Considerăm că pentru a realiza un generator de aplicații pentru clasa precizată, trebuie rezolvate următoarele probleme:

1 - stabilirea exactă a membrilor familiei **8051** pentru care generatorul funcționează precum și a particularităților fiecăruia și a consecințelor în structura hardware / software ,

2 - stabilirea unui set maximal și a unui set minimal de comenzi pentru monitorul serial ;

3 - definirea listei de măsurători posibile și a metodelor folosite ;

4 - descrierea unui program preîncărcător care va fi rezident în **EPROM ;**

5 - definirea unei structuri (sau familii de structuri) pentru mesajele între **PC** și microsistemul de achiziții; definirea unui protocol (sau a unei familii) pentru comunicația dintre sisteme ;

6 - definirea unor restricții pentru parametrii de achiziție.

Pe baza acestei problematizări, conturăm o *rafinare* a generării de aplicații în clasa enunțată sintetizată în Tabelul 7.1. Tabelul cuprinde, practic, sub o formă originală, chiar varianta prescurtată a specificațiilor pentru generatorul de aplicații respectiv.

Cele prezentate în acest tabel constituie o formă aproape imediat implemetabilă a generatorului de aplicații, sub forma unor dialoguri ale utilizatorului cu sistemul. Programul poate fi organizat, din acest punct de vedere, asemănător cu programul **LabWINDOWS** [NIn94]. La proiectarea interfeței, se poate aplica conceptul **GUI** (*Grafical User Interface*) [Fow95]. Așa cum sunt elaborate specificațiile, microsistemul poate funcționa și într-un sistem de telemăsură/telemecanică.

Pe baza celor prezentate mai sus, descriem pe scurt principiile care au stat la baza realizării generatorului de aplicații pentru clasa descrisă. Acțiunile pe care le realizează generatorul de aplicații sunt :

a) instalarea de fragmente de program sau subrutine în cod-sursă, conform opțiunilor și serviciilor solicitate de utilizator (spre exemplu, un serviciu de măsurare, sau implementarea unui anumit set de comenzi ale monitorului serial);

b) completarea câmpurilor-argument din codul sursă, corespunzând încărcării imediate a numelor de tabele sau de subrutine, cu valori introduse de către operator (spre exemplu, alegerea parametrilor de comunicaţie serială şi completarea valorilor potrivite de programare în cuvintele de comandă din secvenţa de iniţializare, caracterele de comandă sau raportare pentru aplicaţia controlată prin legătura serială, etc.);

c) realizarea unor conexiuni (legături) între diferitele aspecte ale aplicaţiei (spre exemplu, conectarea unor zone de memorie completate de serviciile de achiziţie, cu zonele de raportare către nivelul ierarhic superior, sau legarea parametrilor de programare a achiziţiilor (*settings*) cu valorile recepţionate după comanda de setare a acestor parametri);

Pentru a construi generatorul de aplicaţii, se foloseşte principiul *"fill-in-the-blank"*, implementat prin cutii de dialog, fiecare corespunzând unei linii din Tabelul 7.1. Dialogurile din aceste cutii de dialog permit alegerea unei opţiuni (ceea ce duce la acţiuni de tip a) sau c)) sau introducerea unor valori sau şiruri de caractere (pentru acţiuni de tip c)). Codul-sursă este încărcat dintr-un fişier (*template file*) conform opţiunilor proiectantului care foloseşte generatorul, pe baza unui tabel de corespondenţă. Fragmentul de program cerut este regăsit prin căutare după o cheie. Locurile unde urmează să fie introduse valori numerice sau alfanumerice sunt marcate prin caractere non-**ASCII** şi poziţia lor este indicată prin coordonate, pentru ca rezultatele dialogului cu utilizatorul să poată fi inserate la locul corect.

O parte a problemelor de generare de aplicaţii poate fi simplu rezolvată prin **tehnici de speţa întâi** (în accepţiunea de la clasificarea (5.37)), adică prin echivalări la începutul programului, pentru selectarea variantei dorite, prin asamblare condiţionată, impunând opţiunea operatorului printr-o valoare care să determine alegerea codului dorit cu directive de asamblare de tip **if ... else ... endif**.

Modul cum este condus dialogul cu utilizatorul este sintetizat practic în Tabelul 7.1. Principial, programul realizează un dialog cu utilizatorul, pe baza tabelului prezentat, prin care descrie practic condiţiile de funcţionare cerute ale sistemului dorit (desigur, în limitele clasei de aplicaţii) şi construieşte codul-sursă, prin trei categorii distincte de operaţii, descrise în acest paragraf : instalare (inserare) de diverse servicii, compelatarea unor parametri şi conexiuni între componente.

Tabelul 7.1. Rafinarea specificațiilor pentru generatorul de aplicații

Nr crt	Elementul de specificare	Lista de specificații	Tip di-a-log	Subdiviziuni	Tip di-a-log	Caracteristici suplimentare	Tip di-a-log	Obs.
1.	Tip microcon-troller	8051, 8031, PCB80C552, ...	-	-	-	-	-	-
2.	Ceasul de timp real	-perioada -inst. citire tastatură. -instalare rutină USER	E d/n d/n	- - -	- - -	- - -	- - -	- -furnizată de USER - idem
3.	Hardware pentru măsurători	-CAN	S	-citire prin mem. mapping -citire prin porturi interne -citire prin porturi externe -CAN intern	SE	-lungime cuvînt (8,10,11,12,14, 16) -cu/fără semn -aranjare (L,H sau H,L) -citire prin IT sau polling -comanda CAN, MUX, S & H	SE d/n SE SE E	- - - - -
		-intrări de contorizare impulsuri		-pe IT externă -pe contor intern -pe circ. contoriz. ext -pe port intrare	SE	-sel. IT. ext. (dacă e cazul) -sel. contor -"descrierea" circuitului -indicare bit	SE SE E SE	- - -eventual driver -
		-întreruperi externe		-instalare rutină USER -instalare sis. măs. poziție	d/n d/n	- -	- -	- -
		-citire/scriere porturi		-prin mem mapping -prin porturi interne	SE	-adr. port extern -selectare port	E SE	- -
4.	Monitor serial	-opțiune instalare -set comenzi	d/n S	- -	- -	- -comenzi	- E	- -din set maximal
5.	Preîncărcă-tor	-generare separată	d/n	-	-	-comenzi	E	-
6.	Legătura serială	-la punere sub tensiune -în aplicație -în monitorul serial	S	-IT/polling -param. transmisie	SE SE	- - -	- - -	pt. default toate sunt la fel

Tabelul 7.1 (continuare)

7.	Tratare de evenimente semnificative	-atribuire (conectare)	S	-pt. evenimente externe -pt. evenimente interne	E E	-se apelează secvenţe de tratare	-	-
		-producere de evenimente prin program		-	-	-	-	apelabilă în secv. USER
8.	Măsurători	-achiziţie -măsurare frecvenţă -măsurare perioadă	S	-	-	-param. au atribuiţi nr. logice (prin editare)	-	corespondenţa între rezultate şi zone identificate prin nr. Logice
		-măsurare turaţie		-prin măsurarea perioadei -prin măsurarea vitezei unghiulare	SE	-indicare nr. impulsuri trad./tură -indicare timp de măsurare	E E	-eventual driver propriu
		-măsurare poziţie		-prin contorizare hard -prin contorizare soft	SE	-"descriere" circuit -indicare mod de citire	E E	idem -
		etc			.	.	...	...
9.	Structura informaţiei pe unitatea serială	-comenzi (parametri de lucru)	S	-nr. valori -lungime -atribuire nr. logice	E E E	-	-	-eventual instalare Driver
		-rapoarte (parametri de răspuns)		idem	E	-	-	
		-controlul transmisiei		ACK, NACK, ENQ, SLEEP, etc. (eventual selectare protocol)	S	- structura - nr. repetări mesaj, etc.	E	
10.	Inserare de secvenţe USER	-oriunde în program	E	-editare secvenţă	E	-	-	în orice context
11.	Mod de funcţionare	-mod slave -intempestiv	SE	-fără cod de recunoaştere -cu cod de recunoaştere	SE	-codul -perioada de transmisie	E E	-

Prescurtări : S - selectare
SE - selectare exclusivă (doar unul din listă - "*radio button*")
E - editare
d/n - opţiune da/nu

7.3. Prezentarea generatoarelor de aplicații

Generatorul de aplicații de instrumentație pentru sisteme cu microcontrollere din familia **8051** a fost realizat în trei versiuni [Sto95-11].

Prima versiune a fost concepută pentru a fi utilizată sub sistemul de operare **DOS**. Pentru a realiza rapid o interfață prietenoasă cu utilizatorul, pentru ca acesta să permită selectarea cu ușurință dintr-o paletă largă de opțiuni, a fost folosit la proiectare generatorul de interfețe **XView**. Obiectele grafice (cadre, butoane, *settings*) utilizate de interfață fac necesară folosirea limbajului **C++**. Interfața permite utilizatorului să construiască configurația specifică pentru aplicația sa prin selectarea obiectelor grafice dorite (prin *clicking*). Utilizatorul poate de asemenea completa numele subrutinelor utilizator precum și punctele de apelare. El nu are decât să adauge fișierele conținând sursele rutinelor pe care le-a definit. Odată aleasă configurația și sarcinile de realizat de către hardware, aceste elemente de structură (*settings*) sunt reținute. Programul construește codul-sursă al aplicației folosind un fișier *"template" (șablon)*. Structura acestui fișier permite transmiterea parametrilor de la generator la codul-sursă, rezultând codul-sursă final. Ca exemplu, în fișierul *"template"*, locul variabilelor este marcat cu un caracter de cod mai mare ca **0xd0**. Dacă în fișier după o variabilă este un număr, prezența acestei linii în codul-sursă este condiționată de valoarea unei variabile pointate de acel număr, în lista de variabile care constituie câmp de introducere de valori în cadrul dialogurilor cu utilizatorul.

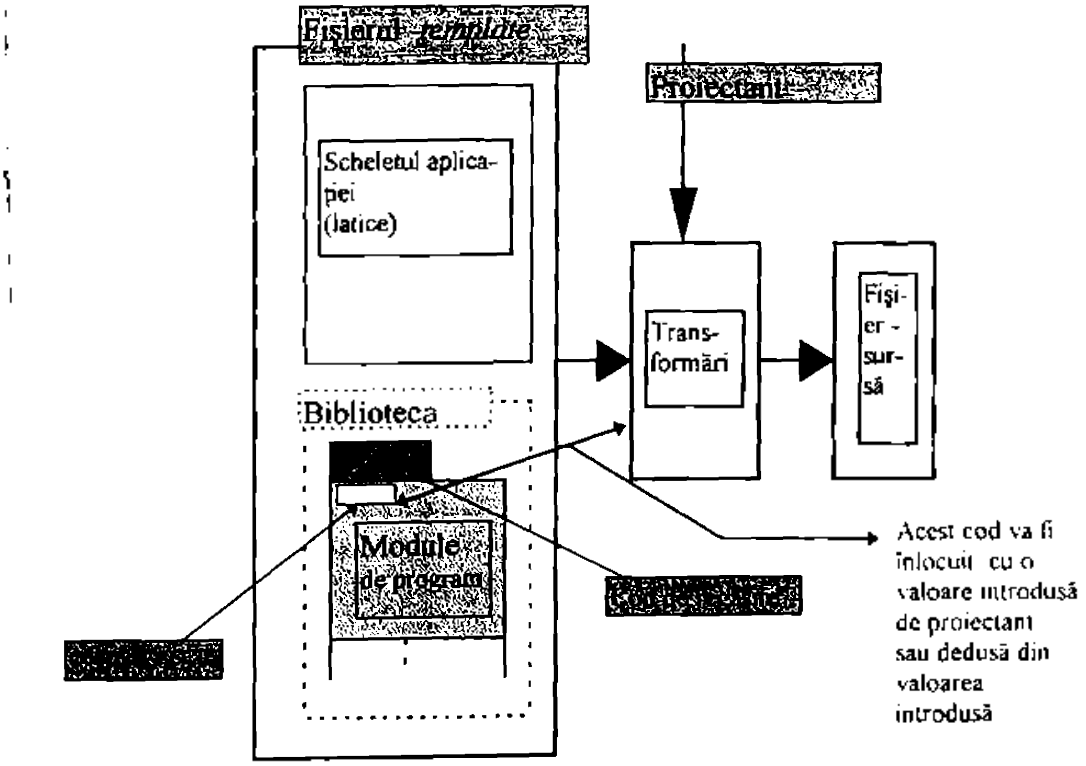


Fig. 7.1. Procesul generării aplicației

Fig. 7.1 reflectă procesul de generare a aplicației descris mai sus. Astfel, din schelet, aplicației (ca și în Cap. 1, putem să numim acesta, *latice* a aplicației) și din modulele din bibliotecă, selectate conform opțiunilor utilizatorului, se obține codul-sursă. Eventualele coduri neafișabile vor fi înlocuite cu valori introduse de la consolă de către operator sau generate pe baza unor algoritmi (spre exemplu, setarea ceasului de timp real). În timpul utilizării generatorului, este posibilă urmărirea modului cum se schimbă codul-sursă, chiar pe parcursul proiectării cu acest generator. După setarea unui atribut, prin apelul unei funcții de ieșire în fereastra cu rezultatul generării (în partea de jos a ecranului) este afișat codul-sursă corespunzător dialogului realizat (v. fig. 7.2).

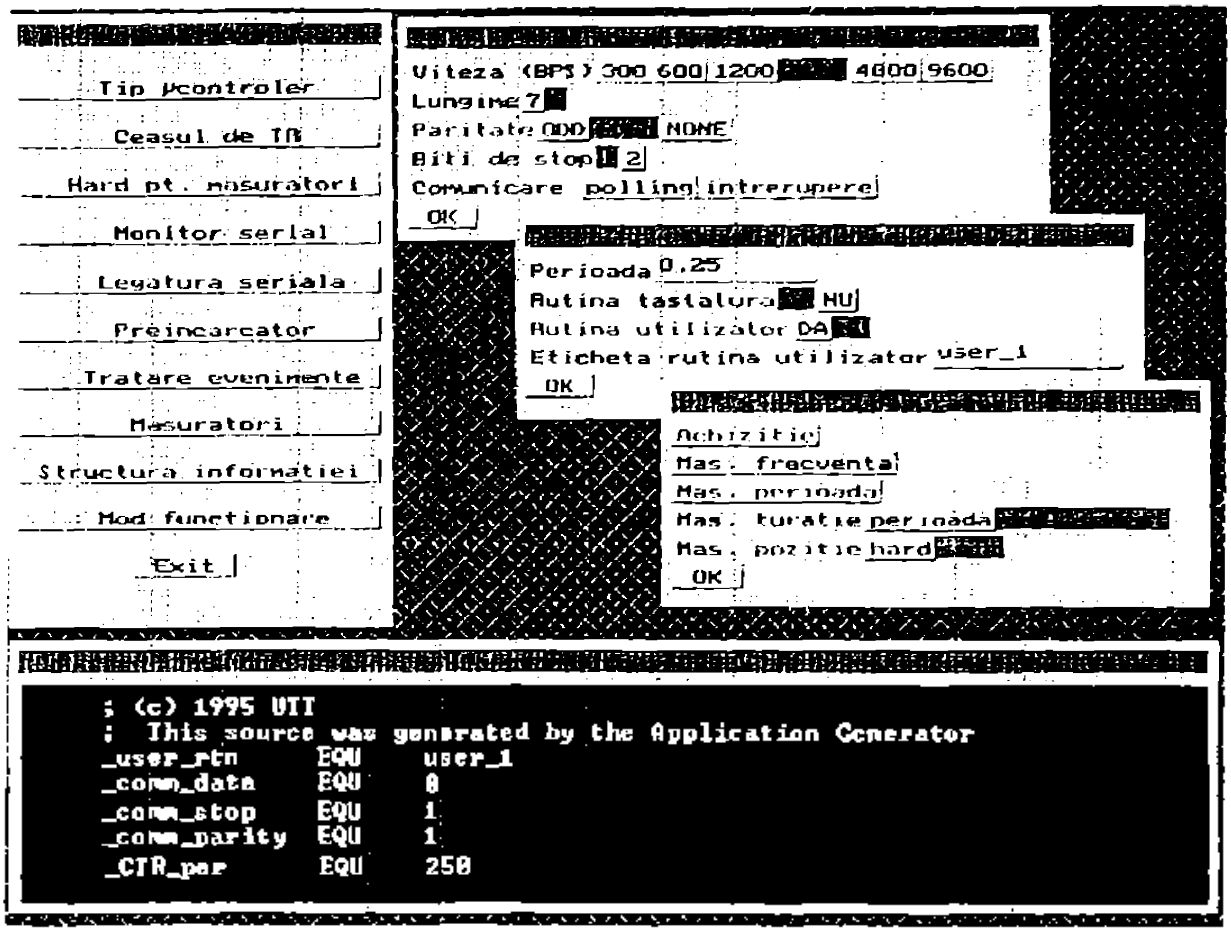


Fig. 7.2. Versiunea **DOS** a generatorului de aplicații

Datorită structurii parametrice a fișierului *"template"* este posibilă modificarea fără probleme a codului generat. Astfel, putem extinde ușor întrebuințarea generatorului și pentru alte familii de microcontrollere, nu numai pentru familia **8051**. Rămâne însă de rezolvat (și se poate rezolva cu oarecare efort de proiectare) problema redefinirii dialogurilor, altfel decât prin program (deci prin încărcarea acestora tot din fișiere).

O altă problemă majoră este legătura dintre sarcinile pe care trebuie să le realizeze aplicația și hardware-ul dat. Trebuie să luăm în considerare limitele resurselor hardware atunci când acestea sunt solicitate de diversele sarcini de realizat de către aplicație. Problema gestiunii

încărcării acestor resurse, în condițiile competiției pentru "*acaparrarea*" acestora, este de asemenea o problemă deschisă.

A doua versiune de program, numită **GAM** ("*Generator de Aplicații pentru Microcontrollere*") a fost scris în **Visual Basic**, pentru a rula sub **Windows**. Specific acestei versiuni este existența unei colecții de "*scule*" (*toolbar*), ceea ce crește viteza de proiectare. Acest "*toolbar*" face posibilă selectarea dialogului dorit printr-un simplu click pe icoana corespunzătoare (vezi fig. 7.3). Fișierul *template* are aceeași structură ca și cea folosită în versiunea anterioară. Ca o facilitate în plus, menționăm posibilitatea reținerii mai multor versiuni succesive ale aceleiași aplicații. Astfel, managementul aplicațiilor create este mai flexibil, permițând reținerea unei "*urme*" a proiectării și astfel, experimentarea mai multor "*căi*" de proiectare și reveniri ulterioare asupra unor versiuni mai vechi (reținute ca fișiere *backup*), dacă proiectantul nu este mulțumit de variantele obținute.

Prezentăm în continuare câteva figuri reprezentând ecrane tipice de dialog, pentru câteva

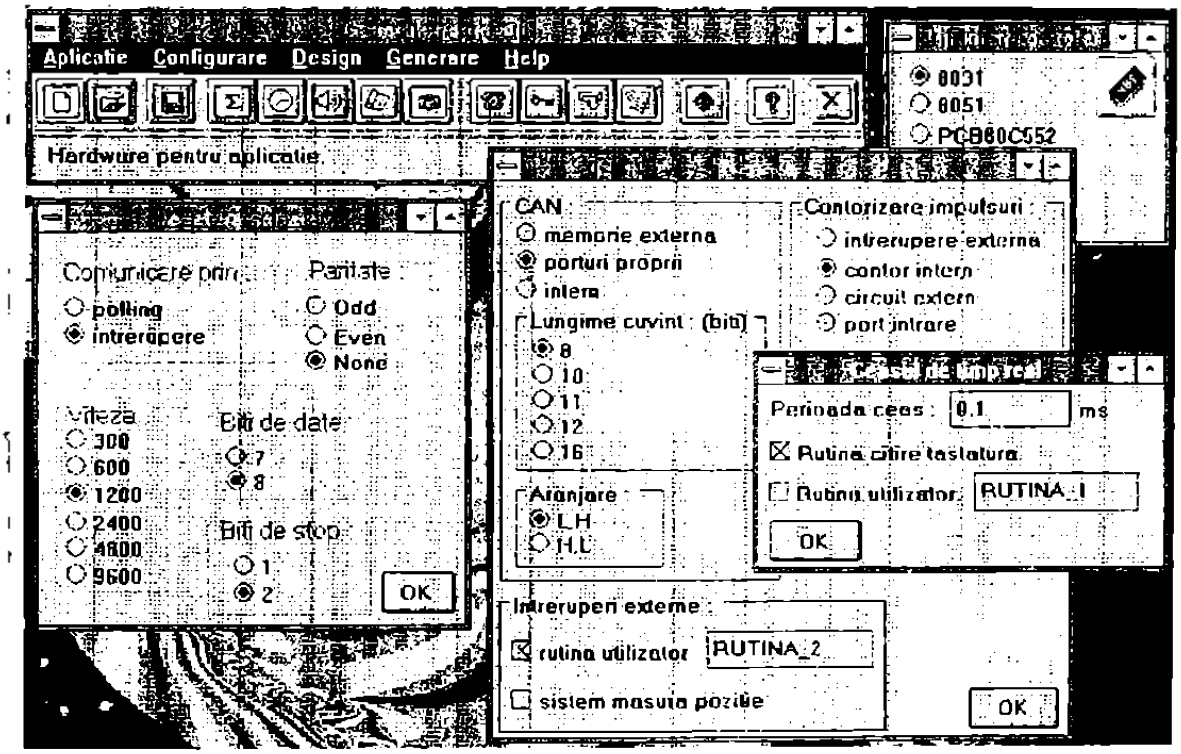


Fig. 7.3. Programul **GAM** (versiunea **Windows** a generatorului de aplicații)

linii din Tabelul 7.1, pentru o **a treia variantă** de **GA**, al cărui principiu de funționare este identic cu al celor două anterior prezentate, dar care a fost scris în **Borland C++ 3.1**, pentru **Windows**. Programul implementează doar un subset al opțiunilor ce rezultă din Tabelul 7.1.

Astfel, pentru linia 3 din tabel, aspectul dialogului este cel din Fig. 7.4. Dialogul constă într-o serie de setări solicitate de către sistem de la proiectant : fie selecții, fie completări de

câmpuri, la fel ca la **GA** precedente. Pentru dialogul de setare a datelor cerute prin linia 8 din Tabelul 7.1, aspectul consolei este cel din Fig. 7.5.

Cele două dialoguri de mai sus sunt logic legate între ele, întrucât măsurătorile care pot fi efectuate, depind de hardware - ul disponibil. De aceea, GA are incluse facilități de verificare a compatibilităților : spre exemplu, măsurarea de poziție nu este posibilă decât dacă sistemul conține facilități hardware în acest sens, setate corespunzător în dialogul anterior.

Programul generează, pe baza dialogului, un fișier care conține setările dorite. Acesta poate fi ulterior încărcat cu comanda **File** din meniul principal. Generarea de cod este declanșată de comanda **Program**, iar setările sunt apelate la comanda **Settings**. Programul este relativ simplu, nu are multe facilități dar este didactic și conține toate atributele principale și funcțiile unui **GA**.

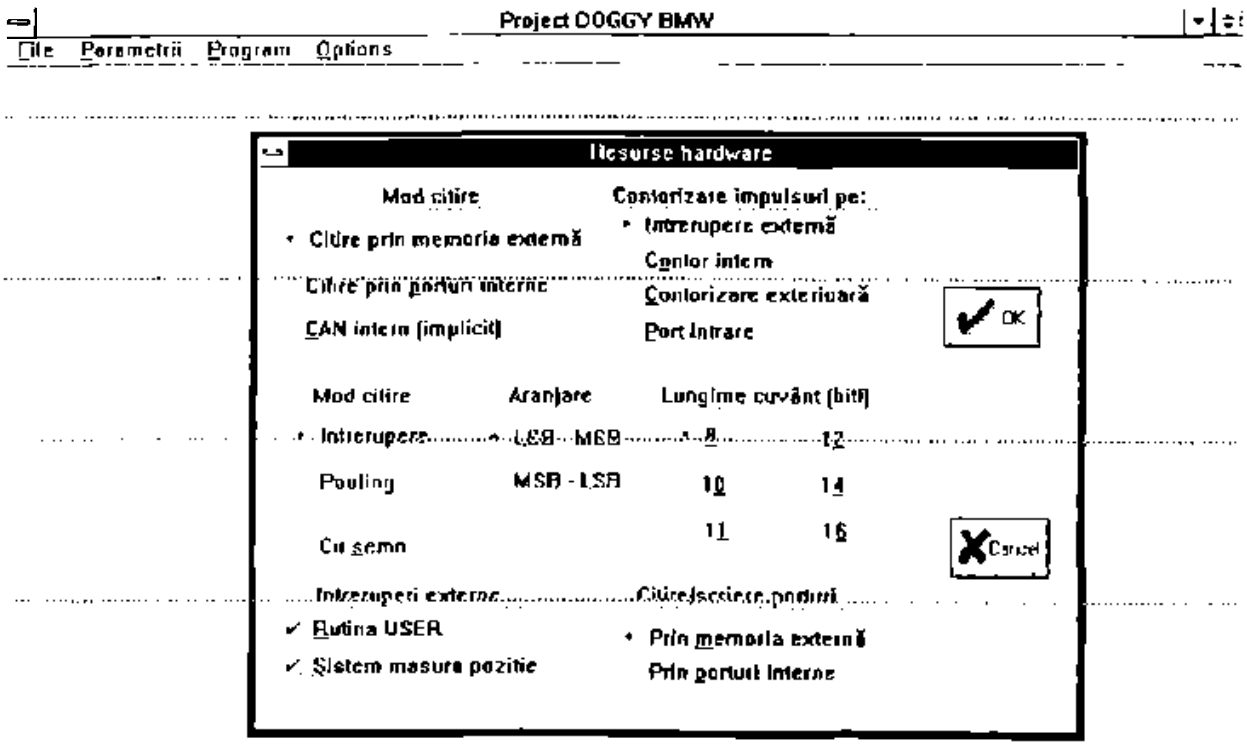


Fig. 7.4. Aspectul consolei pentru dialogul de setare a hardware-lui disponibil

*Programele descrise constituie implementări ale unor **GA** pentru clase de aplicații definite în § 7.1. Dialogul pe care îl implementează ambele variante propuse este condus după modelul Tabelului 7.1. Prin modul de realizare și prin unele soluții adoptate, programele prezintă numeroase trăsături de originalitate : clasa de aplicații căreia îi sunt destinate, structura dialogului, aspectul consolei, rezolvarea contruirii codului-sursă din fișiere - **template**, inserarea de parametri prin detectarea unor coduri non-ASCII specifice, etc.*

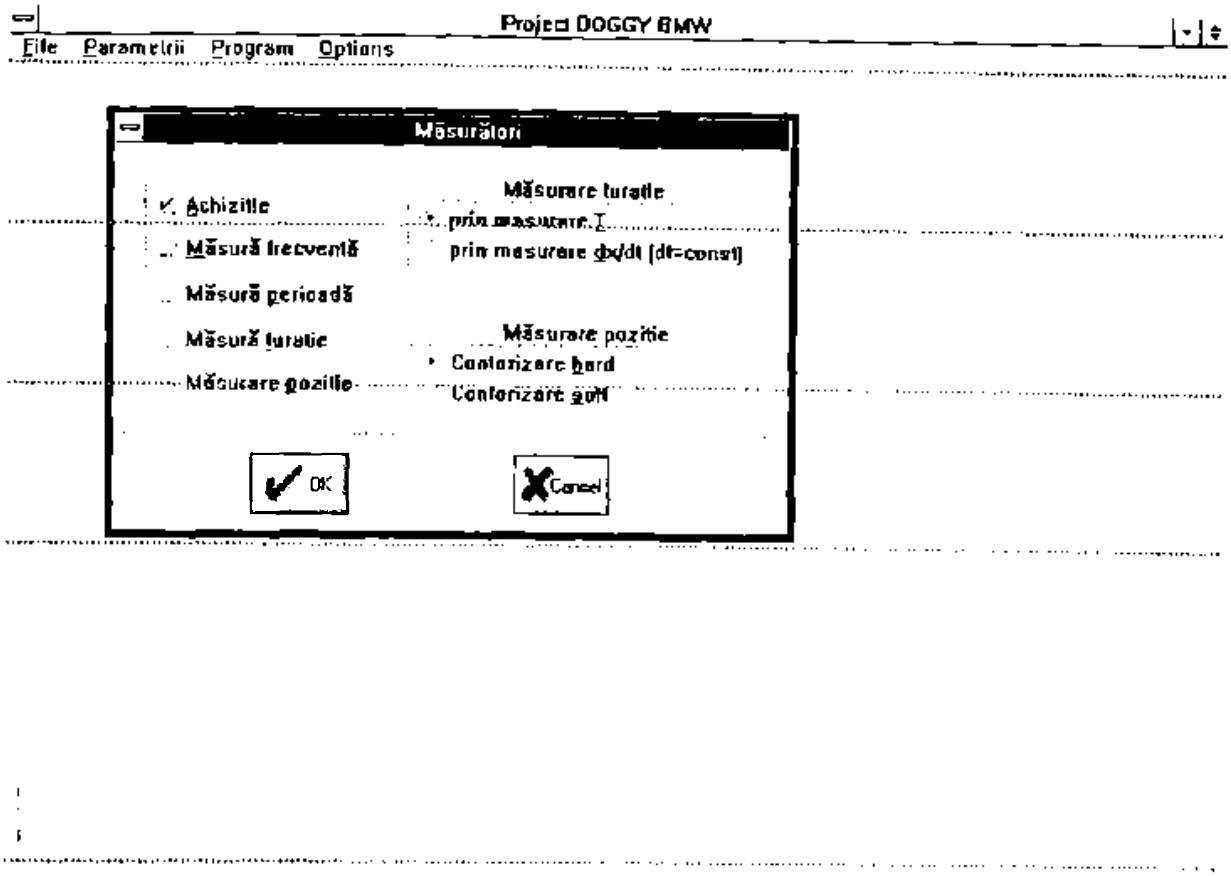


Fig. 7.5. Setarea măsurătorilor

7.4. Concluzii asupra problematicii GA

Un important efort al proiectanților software, atât din mediile universitare cât și de la firmele producătoare de software și sisteme de instrumentație, este orientat spre elaborarea **GA** sau a unor medii care au facilități de generare de aplicații. Proiectarea unui **GA** presupune multă activitate, de aceea nu se justifică decât în cazul existenței unei piețe pentru clasa căreia îi este dedicat generatorul respectiv.

Ponderea în continuă creștere a produselor software în domeniul enunțat, în contextul general al aplicațiilor comerciale este o dovadă că efortul, teoretic și practic, în acest domeniu, se justifică.

*Există încă subdomenii neacoperite, în domeniul aplicațiilor de conducere de proces, în care se pot dezvolta **GA** originale. Un astfel de domeniu este cel al partenerului pentru achiziție, într-un sistem de instrumentație. În acest capitol se propune o clasă de astfel de aplicații cu microcontroller **8051**, de asemenea sunt prezentate într-o formă sintetică specificațiile, modul de implementare și două versiuni ale generatorului.*

Unele probleme sunt lăsate deschise, deci pot constitui puncte de plecare pentru noi direcţii de dezvoltare : una este aceea a realizării generatorului astfel încât să permită trecerea uşoară la altă clasă, cealaltă este aceea a verificărilor de compatibilitate şi de încărcare de resurse relativ la specificaţiile furnizate de utilizator. Dar această a doua problemă este generală şi nu apare doar la proiectarea unui generator de programe, ci se pune în general la orice proiectare software.

O direcţie interesantă şi, credem, utilă, de dezvoltare o reprezintă perfecţionarea interfeţei cu utilizatorul, prin folosirea programării grafice, într-un mod asemănător cu **LabVIEW** [NIC92]. Astfel, generarea de aplicaţii devine accesibilă şi celor care nu cunosc limbajul de asamblare al familiei de microcontrollere al clasei respective de aplicaţii iar proiectarea devine mai rapidă şi mai plăcută.

Se mai cuvin făcute câteva comentarii legate de conexiunea ce există între problematica generării de aplicaţii de instrumentaţie şi cea a bunei definiri a unui modul **BIOS** pentru clasa respectivă, problemă tratată în § 6.4. Într-adevăr, dacă modulul **BIOS** asigură o mare varietate de funcţii pentru acoperirea majorităţii cerinţelor de măsurări, comunicaţie, funcţii generale etc., definirea unei noi aplicaţii se simplifică extrem de mult şi rolul unui generator de aplicaţii se simplifică, reducându-se în special la apelul unor funcţii **BIOS**, conform unor scheme rezultate din specificaţiile utilizatorului, rezultând astfel reutilizarea în sensul rel. (1.1,a). De altfel, un exemplu de aplicaţie simplă scrisă astfel a fost prezentat la sfârşitul paragrafului menţionat. Evident, simplitatea implementării este remarcabilă. În acest caz, practic, generatorul de aplicaţii se reduce la o interfaţă potrivită cu utilizatorul şi la implementarea unor convenţii structurale destul de simple. Vom arăta în capitolul următor cum se poate realiza generare de aplicaţii (dar pe alt principiu decât dialogul cu utilizatorul prin meniuri sau cutii de dialog, aşa cum funcţionează generatorele prezentate în acest capitol) prin folosirea unor asemenea module **BIOS** cu funcţii extinse.

GA descrise în acest capitol reprezintă soluţii clasice pentru rezolvarea problemei generării de aplicaţii pentru clasa prezentată aici şi rezolvă o problemă concretă, tipică. Multe din soluţiile adoptate sunt însă originale. Problematica generării de aplicaţii conduce însă, aşa cum am arătat mai sus, la multe alte deschideri interesante, unele tratate în continuare în lucrare, altele constituind posibile direcţii viitoare de dezvoltare.

CAPITOLUL 8.

DOUĂ INSTRUMENTE CASE PENTRU ASISTAREA PROIECTĂRII APLICAȚIILOR DE CONDUCERE DE PROCES

În acest capitol sunt prezentate două medii de programare de tip **CASE**, destinate asistării mai multor etape ale ciclului de viață al unor programe din două categorii de programe :

- **programe multitasking pentru conducere de proces**, construite în jurul unui nucleu executiv în timp real peste sistemul de operare **MS-DOS**, respectiv
- **programe de instrumentație** sau supraveghere-conducere locală de proces, destinate a rula pe microsisteme cu resurse reduse.

Mediile prezentate diferă, atât prin destinația produselor-program dezvoltate sub aceste medii, cât și prin soluțiile de proiectare, prin logica de utilizare și prin destinație. Ambele medii au implementat un subset de **facilități standard** ale unui instrument **CASE** (v. § 1.8).

Caracteristica esențială, care determină de fapt includerea lor în prezenta lucrare, este facilitatea oferită, de asistare a **generării** rapide de noi aplicații. În aceeași idee, ambele instrumente au înglobate posibilități tehnologice de asistare a reutilizării și sub alte forme decât generarea de aplicații. De asemenea, unul din instrumentele prezentate oferă o facilitate nouă față de programele similare, prin aceea că are inclusă o funcție specială de **evaluare**, care asigură calculul unor **indicatori** legați de **reutilizare** și de complexitatea aplicațiilor, indicatori preluați din literatură (v. cap. 3) sau propuși în această lucrare (cap. 4 și 5).

8.1. Instrument CASE pentru dezvoltare de aplicații multitasking de conducere de proces

În continuare prezentăm un mediu de dezvoltare destinat accelerării proiectării aplicațiilor de conducere de proces pe un suport multitasking, sub sistemul de operare **MS-DOS**. Mediul prezentat acoperă etape ale ciclului de viață al programelor, care nu țin doar

de programarea propriu-zisă. Prin modul cum este realizat, mediul înglobează şi facilităţi de bază software, deci implicit are atribute de generator de aplicaţii [Sto96-N].

Mediul prezentat realizează o serie de funcţii caracterizate succint prin câteva cuvinte, în continuare. Pe un suport standard al primitivelor oferite de un executiv (care poate fi încărcat distinct, dintr-un fişier de configurare), prin intermediul unei interfeţe grafice, utilizatorul poate trasa diagrama de flux de date a aplicaţiei dorite, în paralel fiind creat automat şi proiectul asociat, conţinând un minim deductibil de linii-sursă. Aceste surse pot fi în continuare editate imediat. Aplicaţia astfel definită este păstrată în baza software, putând apoi opera asupra ei, pentru a obţine aplicaţii asemănătoare (prin reutilizare). O comandă distinctă permite asocierea unor specificaţii fiecărei entităţi software (taskuri şi structuri de comunicaţie şi sincronizare), necesare regăsirii în vederea reutilizării.

Instrumentul prezentat are înglobate posibilităţi de evaluare software, prin funcţia sa de efectuare a unor calcule de coeficienţi specifici pentru estimarea calităţii proiectării. Calculul unora dintre aceşti coeficienţi este efectuat conform teoriilor dezvoltate în capitolele 4 şi 5 din lucrarea de faţă.

a) Concepţia generală a programului

Programul prezentat este conceput pentru a rula sub **Windows**. De aceea, structura şi aspectul programului sunt subordonate concepţiei generale a sistemului **Windows**

Instrumentul manipulează următoarele **categorii de informaţii şi fişiere asociate** :

1. fişiere - diagramă, ***.DFD**, care înmagazinează informaţiile corespunzătoare unei diagrame de flux de date (**DFD**) asociate unei aplicaţii, informaţiile necesare asocierii la acesta, a numelor şi localizării fişierelor conţinând codul-sursă al aplicaţiei, precum şi informaţiile necesare asocierii fişierelor de specificaţii pentru fiecare entitate software (task sau zonă de intercomunicaţii date) ; aceste fişiere pot fi editate, vizualizate, modificate şi sunt asociate unui **proiect** ;
2. fişiere - sursă, ***.CPP** şi ***.H**, care conţin cod în limbaj de nivel înalt, asociat aplicatiilor în lucru ; aceste fişiere sunt create şi modificate în paralel cu editarea fişierelor-diagramă, iar după ce **DFD** este finalizată şi fragmentele de cod-sursă posibil a fi deduse din **DFD** sunt deci prezente în sursă, utilizatorul poate edita şi insera ceea ce este specific aplicaţiei şi nedeductibil automat din **DFD** ; activitatea utilizatorului este în această din urmă etapă uşurată de informaţia de *specificaţii*, oricând apelabilă, asociată entităţii software aflate în editare curentă ;
3. fişiere - specificaţii, ***.SPC**, care conţin informaţii de specificaţii software, introduse de utilizator şi asociate fiecărei entităţi software din proiect (deci specificaţii referitoare la taskuri, zone de intercomunicaţii date, structuri de date, etc.) ; aceste

informaţii pot fi editate, vizualizate oricând şi modificate de utilizator (asociat entităţii software aflate, sub o formă sau alta, în atenţia utilizatorului, spre exemplu prin faptul că o anumită entitate software este sub editare, sau prin acţiunea expresă a utilizatorului de a desemna entitatea prin click pe entitate, din **DFD** afişată) ;

4. fişiere - clasă-de-aplicaţii, ***.APC**, care conţin informaţii care să permită asocierea unui set de fişiere-diagramă (şi implicit fişierele-sursă şi fişierele-specificaţii, asociate) într-o colecţie de proiecte, în sensul din [Mit87] ; unul din proiecte are regimul privilegiat de **program generic** şi poate fi vizualizat pentru a oferi un cadru pentru crearea de noi aplicaţii, prin trunchiere şi editare ;
5. fişier - istorie, ***.IIST**, conţinând informaţii despre istoria unui proiect, ceea ce permite readucerea **DFD** corespunzătoare unor stadii trecute ale proiectării ; astfel, proiectantul poate reveni dacă drumul pe care a evoluat proiectarea nu duce unde trebuie ;
6. fişier - şablon pentru specificaţii (**template** file), ***.TPL**, care conţine informaţiile de descriere a structurii informaţiilor de specificaţii ;
7. fişier - executiv, ***.XFA**, conţinând modul de apelare a primitivelor executivului (deci fragmentele de program, corespunzătoare), precum şi un model pentru structura generală a unui program construit pe un executiv dat şi convenţiile grafice asociate primitivelor executivului, care vor fi utilizate în trasarea **DFD** ;
8. fişier de configurare, ***.CFG**, conţinând setările selectate de utilizator ;
9. fişier de stare a consolei, ***.DSK**, care permite reluarea programului din punctul unde a fost întrerupt la dorinţa utilizatorului şi conţine informaţii despre starea programului la părăsirea acestuia ;
10. fişier - dicţionar ***.DBF**, care conţine surse de module software (taskuri sau funcţii) şi descrieri (specificaţii) asociate .

Programul îşi va plasa fişierele *şablon, executiv, configurare, iniţializare* şi *dicţionar* în directorul în care este instalat. Pentru fiecare clasă va crea un subdirector în directorul **APPL** (care este un subdirector al directorului de instalare), numit

APPL_nume_a ,

unde **nume_a** este numele dat de utilizator, iar fiecare proiect nou îşi va crea propriul subdirector în **APPL_nume_a**, numit

PRJ_nume_p ,

unde **nume_p** este numele de proiect dat de utilizator. Fiecare subdirector **PRJ_nume_p** va conţine *fişierul-diagramă, fişierul-sursă* şi *fişierul-specificaţii*.

Interacţiunea şi relaţiile dintre fişiere, precum şi gruparea acestora în categorii funcţionale, sunt prezentate în Fig. 8.1.

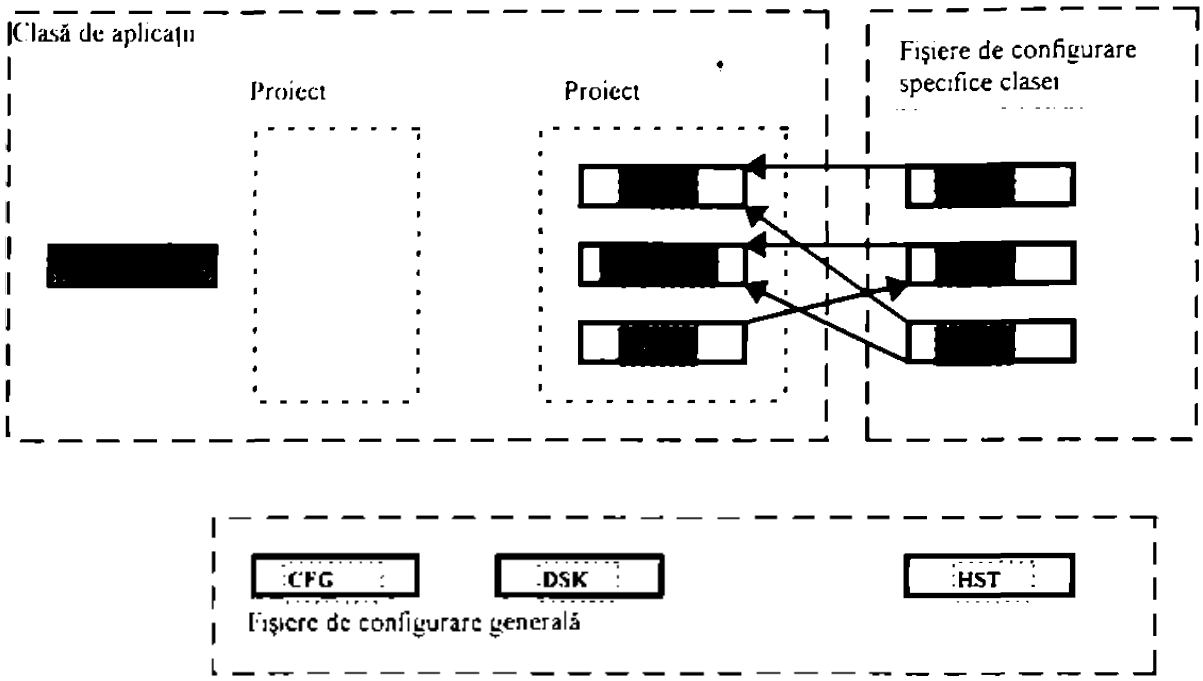


Fig. 8.1. Relaţiile dintre fişierele utilizate de program

Din punctul de vedere al utilizatorului, instrumentul **CASE** prezentată pune la dispoziţia acestuia:

- un sistem ierarhic de meniuri de tip "*pull-down*",
- un set de "*scule*" prin intermediul unei "*cutii de scule*" ("*tool-bar*") şi
- o interfaţă grafică pentru editarea şi vizualizarea diagramelor de flux de date ale aplicaţiilor asistate în proiectare.

Mai jos, prezentăm setul de **comenzi** ale meniului principal :

• **File** este comanda care determină apariţia unei liste de comenzi de submeniu, care toate manevrează fişiere - diagramă ; lista respectivă este următoarea (unele din comenzi sunt specifice aplicaţiilor **Windows**) :

- **New** - determină deschiderea unui fişier-diagramă nou (implicit, se cere editarea informaţiilor generale asociate) ;
- **Open** - deschide un fişier existent ;
- **Save** - salvează un fişier-diagramă editat ;
- **Save as** - salvează un fişier sub nume nou ; dacă o comandă de salvare are loc după un **New**, şi dacă în clasa respectivă nu este încă definit un program generic, utilizatorul este întrebat dacă nu doreşte marcarea programului salvat, ca program generic ;

- **Delete** - şterge un fişier-diagramă (implicit, proiectul asociat) ;
- **About** - afişează informaţii generale despre proiectul în lucru ;
- **Clone** - crecază o copie a proiectului în lucru ;
- **Version** - fixează un proiect ca şi versiune aparţinând clasei curente ;
- **Backtrack** - permite încărcarea unei versiuni precedente (din istoria proiectului) ;
- **Design status** - comută între etapa de specificaţii şi etapa de proiectare ; diferenţa între acestea constă în faptul că în etapa de specificaţii se asociază implicit fiecărei entităţi din **DFD** editată de utilizator, o fereastră de editare a specificaţiilor, pe când în etapa cealaltă, asocierea implicită se referă la fereastra de editare de cod-sursă aferent ; cele două stări vor fi **Specifications**, respectiv **Code** ,
- **Status** - afişează starea proiectului, referitoare la evoluţia acestuia (zile rămase până la termenul final, procentajul de completare a specificaţiilor şi codului-sursă, numărul de module definite, lista entităţilor software cu proiectarea încheiată, etc.) ;
- **Quit** - ieşire din program.

• **Class** este comanda corespunzătoare manipulării informaţiilor referitoare la gruparea proiectelor şi programelor pe **clase de aplicaţii** (în sensul din [Mit87]) ; comenzile de submeniu corespunzătoare sunt :

- **New** - deschide o clasă nouă de aplicaţii (deci şi fişierul-clasă asociat) ;
- **Open** - deschide o clasă existentă ;
- **Save** - salvează o clasă de aplicaţii ;
- **Save as** - salvează o clasă de aplicaţii cu nume nou ;
- **About** - afişează informaţii generale despre clasa curentă ,
- **Report** - afişează un sumar al clasei curente (adică, numărul de versiuni din clasa curentă, denumiri şi informaţii succinte despre fiecare) ;
- **Generic program** - este o comandă de tip comutator, care determină deschiderea fişierului-diagramă al programului generic, care va fi păstrat în fereastra de editare, cu altă culoare decât fişierul-diagramă curent, în vederea uşurării proiectării, până când operatorul nu va acţiona din nou această comandă.

• **Dictionary** este comanda de manipulare a bazei de date asociate instrumentului **CASE**, care cuprinde informaţii despre toate modulele de program (atât în ceea ce priveşte specificaţiile, cât şi în ceea ce priveşte codul-sursă) care au fost proiectate folosind acest instrument ; scopul existenţei acestui dicţionar este de a asista proiectarea prin regăsirea uşoară a unor componente

dorite din depozitul de componente (v. Fig. 1.2) în vederea reutilizării ; comenzile asociate submeniului activat sunt :

- **Browse** - permite afişarea listei de module disponibile şi selectarea modulului dorit ; operatorul poate alege diverse criterii de selecţie pentru modulele afişate ;
- **Search** - realizează căutarea unui modul de program conform unor criterii de căutare selectate de utilizator ;
- **Insert** - inserează un modul de program în dicţionar ;
- **Paste** - copiază informaţia afişată, în *Clipboard* ;
- **Edit** - permite editarea informaţiilor aferente unui modul de program, pentru ajustarea caracteristicilor şi informaţiilor în vederea îmbunătăţirii posibilităţilor de reutilizare ;
- **Delete** - determină ştergerea informaţiilor referitoare la un modul de program;
- **Report** - afişează informaţii despre modulele de program, selectate şi ordonate după diverse criterii alese de operator.

• **Verify** determină efectuarea unor verificări de corectitudine pentru un proiect realizat (deci pentru **DFD** şi modulele-program, respectiv specificaţiile asociate) ; comenzile de submeniu sunt :

- **Completeness** (completitudine) - determină verificarea completării tuturor rubricilor de specificaţii, existenţa ambilor parteneri în conexiunile din **DFD**, închiderea tuturor fluxurilor de date, etc. ,
- **Consistency** (consistenţa) - declanşează proceduri de verificare mai complexe, referitoare la corecta definire a interfeţelor, la compatibilitatea acestora, etc. ;
- **Empirical rules** - verifică respectarea, în cadrul proiectului curent, a unor reguli empirice pentru structuri ierarhice, precum cele din **DFD** (spre exemplu ca în lucrarea [Ion82]) :
- **Built** - determină declanşarea succesivă a tuturor verificărilor disponibile.

• **Evaluation** (evaluare) nu este prezentă în cadrul instrumentelor **CASE** disponibile comercial sau descrise în bibliografie ; intenţia noastră a fost de a implementa aici sistemele de evaluare a calităţii software, din punctul de vedere al reutilizării, prezentate în capitolele 3, 4 şi 5 ; de aceea, comenzile submeniului activat de această comandă vor fi :

- **Complexity** - evaluarea complexităţii software, în sensul din § 3.1, conform diferitelor criterii propuse în paragraful citat ;
- **Module reuse rate** - evaluarea gradului de reutilizare de modul, conform uneia din definiţiile din § 3.2 ;

- **Program reuse rate** - evaluarea gradului de reutilizare de program, în sensul din § 4.1 ;
- **Class reuse rate** - evaluarea gradului de reutilizare de clasă, în modul prezentat în § 4.1 ;
- **Program accesibility degree** - evaluarea gradului de accesibilitate de program, conform definiției (5.20) ;
- **Class accesibility degree** - determinarea gradului de accesibilitate de clasă, conform definiției (5.22) ;
- **Phase design speed** - calculează viteza de fază a proiectării, conform rel. (5.23) ;
- **Group design speed** - calculează viteza de grup a proiectării, aplicând relația (5.26) ;
- **Belongeness test** - verifică, pentru un program (proiect) dat, apartenența la o clasă, conform criteriului (5.28) ;
- **Consistency test** - verifică dacă o clasă este consistentă, în sensul criteriului (5.30) ;
- **Omogenity test** - verifică omogenitatea unei clase, prin testarea conform criteriului (5.31).

• **Tools** permite accesul la o serie de facilități speciale ; lista acestora este dată mai jos :

- **IDE** (de la "*Integrated Developement Environment*" - mediu integrat de dezvoltare) - permite accesul direct din mediul **CASE** în mediul de programare folosit pentru obținerea codului executabil și depanare ; astfel, odată creat codul-sursă, acesta poate fi imediat compilat și aplicația poate fi testată, fără a trebui să se iasă din programul **CASE** ;
- **Export diagram** - permite exportarea unui fișier-diagramă, sub forma unui fișier ***.BMP** ("*bitmap*") ;
- **Export code** - asigură copierea codului-sursă la destinația dorită ;
- **Export specifications** - generează un fișier-document, în format ASCII, la destinația dorită, în vederea editării sau tipăririi ulterioare, conținând toate specificațiile asociate unui proiect ;
- **User command** - lansează în execuție o aplicație a utilizatorului.

• **Set-up** este o comandă de stabilire a unor circumstanțe de funcționare a instrumentului **CASE** ; astfel, prin intermediul comenzilor de submeniu corespunzătoare, utilizatorul poate realiza următoarele acțiuni :

- **Specifications** - editarea fișierului ***.TPL**, utilizatorul putând astfel modifica rubricatura specificațiilor ;

- **Executive** - editarea fişierului ***.XFA**, astfel putându-se realiza adaptarea instrumentului **CASE** pentru a fi utilizat cu un executiv oarecare ; în cadrul acestei comenzi se poate încărca eventual un alt fişier ***.XFA**, realizând o adaptare rapidă a mediului la particularităţile unui alt executiv ;
- **Access** - permite definirea accesului comenzii **IDE** din meniul principal, la resursele mediului de programare folosit ;
- **User** - permite utilizatorului să definească numele programului lansat în execuţie de comanda **User command** ;
- **Dictionary** - permite setarea unor parametri de funcţionare ai bazei de date asociate (depozitul de componente reutilizabile) ;
- **Evaluation** - permite selectarea unor criterii de evaluare, dintr-un set prestabilit (spre exemplu, pentru evaluarea complexităţii, în § 2.1 sunt date mai multe criterii ; utilizatorul poate decide care dintre criterii este activ la comanda **Complexity**).

• **Window** şi **Help** sunt comenzi standard ale programelor care rulează sub **Windows**, oferind facilităţi standard de aranjare a ferestrelor, respectiv informare despre utilităţile instrumentului **CASE**.

Instrumentul **CASE** prezentat este scris în **Borland C 4.5** şi foloseşte ierahiile de clase **OWL** (*Object Windows Library*) oferite de acest pachet.

b) Un exemplu de utilizare a instrumentului CASE

În cele ce urmează sunt prezentate câteva aspecte ale utilizării instrumentului **CASE**. Întrucât o prezentare exhaustivă ar ocupa un loc foarte mare în structura lucrării de faţă, în continuare sunt descrise doar câteva aspecte ale dezvoltării unei aplicaţii de conducere de proces, pe scheletul oferit de executivul prezentat pe larg în [Rob94]. Pentru acest executiv lista entităţilor software *standard* este următoarea :

- task
- funcţie
- fanion
- semafor
- bloc resursă
- bloc eveniment
- bloc multieveniment
- rendez-vous

- conductă
- cutie poştală (mailbox)

"*Cutia de scule*" include, în afară de entităţile de mai sus, posibilitatea inserării în **DFD** (respectiv, automat în codul-sursă) a secţiunilor critice delimitate prin dezautorizarea întreruperilor (funcţiile **_lock()** şi **_unlock()**) şi a funcţiilor de suspendare şi relansare a taskurilor (**sleep()** şi **wakeup()**).

O **aplicaţie multitasking tipică**, pe suportul oferit de executivul amintit mai sus, arată astfel :

```
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <mem.h>
...
//
//   Urmeaza un header din pachetul executivului
//
#include "rtc86.h"
//
//   Urmeaza macro-urile de definire a descriptorilor
//   de taskuri
//
TSK_IDENT task1;
TSK_IDENT task2;
...
//
//   Urmeaza macro-urile de definire a semafoarelor si
//   zonelor de intercomunicatii date
//
SEM_IDENT semaf1;
SEM_IDENT semaf2;
...
//
//   Urmeaza definirile variabilelor globale
//
...
```

```
//
//   Urmeaza corpurile taskurilor
//
//        primul task
TASK t1(void)
{
//
//        definiri de variabile locale
//
...
//
//        urmeaza bucla infinita care este de fapt taskul
//        propriu-zis
 _repeat{
//        prelucrari specifice
          ...

//        apeluri de primitive ale executivului
   _s_wait(semaf1, 0);
          ...

 }_forever;
}

//        al doilea task
TASK t2(void)
{
...
_repeat{
     ...

 }_forever;
}
//        urmeaza celelalte taskuri
...
//
//   "Programul principal"
//
void main(void)
{
```

```
//
//   instalarea serviciilor executivului
//
   _install();
//
//   crearea structurilor asociate semafoarelor si
//   zonelor de intercomunicatii date
//
   _s_create(semaf1, 0);
   _s_create(semaf2, 1);


//
//   crearea taskurilor
//
   _create(task1, t1, 2, 8, task1);
   _create(task2, t2, 2, 8, task2);


//
//   trecerea taskurilor in stare activa
//
   _activate(task1);
   _activate(task2);

//
//   lansarea in executie
//
   _run(30000);
}
```

În continuare vom arăta în ce mod se "implică" instrumentul prezentat în *completarea* scheletului de aplicație prezentat mai sus.

Dacă luăm spre exemplu fragmentul de aplicație din Fig. 4.4, legăturile dintre taskuri se pot realiza astfel :

- între nivelul 1 şi nivelul 2, legătură prin "*rezervor*" (în sensul din [Dav86]), adică prin structură de date partajată, protejată de *bloc resursă* ;
- între nivelul 2 şi nivelul 3, legătura se poate realiza prin *conducte*, respectiv prin *mesaj* (pentru taskul 4) ;

- între nivelul 3 şi nivelul 4, legătura se poate realiza prin "*rezervor*" pentru taskurile de luare la cunoştinţă şi prin *conductă* pentru celelalte taskuri.

Utilizatorul, având o imagine aproximativă a aplicaţiei dorite (construită ca urmare a experienţei dobândite sau a vizualizării unor aplicaţii din clasa respectivă, proiectate anterior), va parcurge *etapele de proiectare* prezentate în continuare (marcate cu bile). Enumerarea şi descrierea conţinutului etapelor este completată cu exemple intercalate, pentru o mai bună înţelegere a acestor etape. Iată deci succesiunea acţiunilor utilizatorului mediului **CASE** prezentat :

- încarcă clasa de aplicaţii de "*telemecanică*" (dacă există o astfel de clasă) ;
- încarcă **DFD** asociată programului generic al clasei (dacă este desemnat un asemenea *program*) ;
- deschide un *fişier-diagramă* nou (asociat noului proiect) şi completează informaţiile generale corespunzătoare (cu **Bold** sunt evidenţiate mesajele programului, iar cu litere obişnuite sunt prezentate completările operatorului) :

Name Abbreviated DISPAT **Complete** Dispecer Apa Timisoara
Brief description Sistem de telesupraveghere pentru un front de captare a apei, avind microcalculatoare locale pentru interfata cu procesul si legatura multipunct cu calculatorul central
Keywords conducerea proceselor, telesupraveghere, telecomanda, legatura multipunct
Project manager Stoicu-Tivadar Vasile
Project design team Neagoe Catalin
Stanciulescu Virgilius
Client Departamentul de Automatica si Informatica Industriala
Project start 01.03.1996 **Project term** 01.05.1996
Target computer AT 486 Industrial PC with 1 Mb RAM
Peripherals **Standard** color VGA, HDD 160 Mb,
FDD 3.5", printer IBM
Graphics-compatible, mouse,
COM1, COM2
Non-standard Numerical output
Description Programul asigura urmatoarele functii :
- comunicarea (cu statiile din subordine si cu un nivel ierarhic superior)
- gestiunea evenimentelor din proces
- prelucrari specifice (telemasuri)
- prelucrarea comenzilor operatorului
- controlul efectuarii telecomenzilor
- gestiunea istoriei procesului

- prelucrarea comenzilor nivelului ierarhic superior
- autoteste

Vizualizarea evenimentelor din proces se realizeaza prin mesaje si prin grafica. Evenimentele sunt schimbari in starea semnalizarilor din proces (avarii si telesemnalizari de stare) si depasiri ale valorilor limita ale telemăsurilor.

Operatorul are acces la comenzi prin intermediul unui meniu si prin intermediul *mouse*-ului. Operatorul are acces la istoria procesului atit in ceea ce priveste starea procesului la un moment dat cit si la graficele de evolutie in timp ale telemăsurilor.

Nivelul ierarhic superior poate solicita rapoarte referitoare la starea actuala a procesului, la istorie sau poate cere efectuarea unor telecomenzi.

Source code DISPAT.CPP, DISPAT.H
Version 5.2 **Localisation** C:\CASE\APPL\APPL_TEL\PRJ_DISP
Implementation items **Rtclock** 10000
Others nu

- proiectează **DFD** prin trunchierea **DFD** asociate programului generic şi prin editarea **DFD** rezultate, selectând simbolurile grafice asociate diverselor entităţi software, din "*cutia de scule*" (*tool-bar*) (v Fig. 8.2) ;

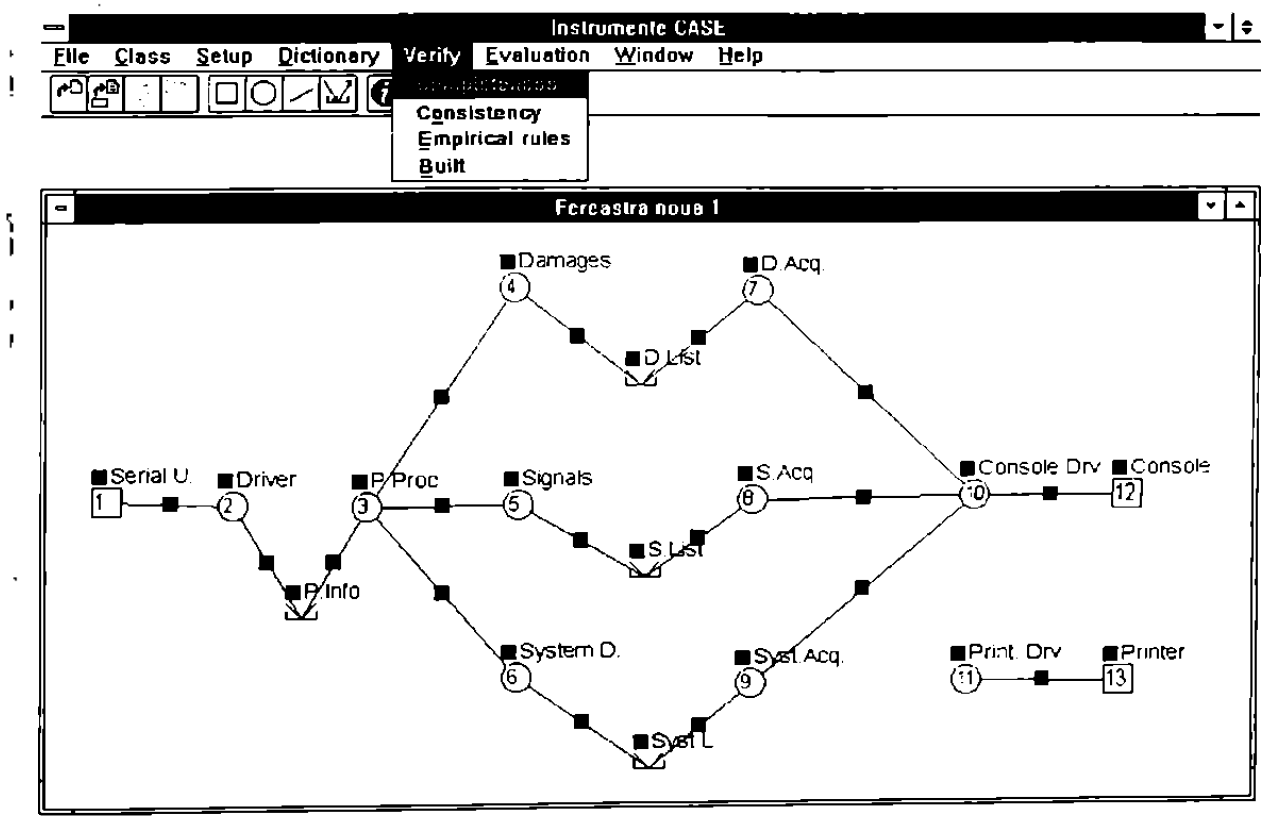


Fig. 8.2. Aspectul consolei în timpul editării unei **DFD**

- selectează la comanda **Design status** starea **Specifications** (e, de altfel, starea implicită după o comandă **New**) ; ca urmare, fereastra implicită selectată cu *mouse*-ul, prin navigare pe suprafața **DFD**, va fi fereastra de **Specifications** ;
- editează, rând pe rând, toate specificațiile, corespunzătoare fiecărei entități software.

În continuare dăm două exemple referitoare la editarea specificațiilor a două entități software distincte ca și tip (un **task**, respectiv o **conductă**) pentru a evidenția astfel utilizarea unor fișiere-*șablon* (*template*) diferite (în cadrul exemplelor prezentate, cu litere accentuate - **bold** - am marcat partea afișată de sistem, corespunzătoare informațiilor din fișierul - *șablon*) :

1. Pentru ***taskul de prelucrare a avariilor*** :

```
Entity  task
Name  Abbreviated  AVARII          Complete prelucrarea avariilor din proces
Brief description Realizeaza actualizarea starii avariilordin proces si
                  generarea mesajelor de avertizare
Keywords          avarii, avertizare
Designer          Stoicu-Tivadar Vasile
Design start      10.03.96    Design term    30.03.96
Priority          4           Sublevel priority  2
Executive services used    pipes, messages, resource blocs
Activated by      main program, ANMESJ task
Activates '       LCAV task, EXAV task, tty_drv, lpt_drv
Inputs            pipe from ANMESJ
Processing
- Taskul este blocat pina la receptia info. prin conducta de la taskul de
prelucrare primara ANMESJ ;
- verifica daca informatia este valida ;
- localizeaza avaria curenta in lista de avarii ;
- verifica basculareа de la starea precedenta, daca nu s-a produs, trimite
prin conducta informatia spre taskul tratare exceptii  EXAV ;
- daca basculareа este corecta, formeaza si trimite mesaje spre driverele de
consola si imprimanta tty drv si lpt_drv ;
- blocheaza prin bloc resursa accesul la resursa partajat "lista de avarii"
- actualizeaza starea avariei in lista ;
- inscrie in lista o cerere de luare la cunostinta ;
- pozitioneaza semaforul pe eliberare resursa lista si lanseaza taskul de
luare la cunostinta avarii proces LCAV ;
Outputs      message to tty_drv
             message to lpt_drv
             pipe to EXAV
             access to "lista_avarii"
```

2. Pentru ***structura informației din conducta de intrare*** a taskului de prelucrare a avariilor (în fapt, aceasta este informația vehiculată, și nu structura propriu-zisă a conductei, care de fapt trebuie să fie transparentă pentru utilizator ; aceeași situație este întâlnită și la mesajele trimise la cutii poștale, unde nu interesează structura cutiei poștale, ci structura și conținutul informației vehiculate spre aceasta) :

```
Entity  pipe
Name    Abbreviated  p_av     Complete info. din prelucrare
primara pt. avarii proces
Brief description Transmite taskului avarii informatiile necesare localizarii
si actualizarii unei avarii proces
Designer          Stoicu-Tivadar Vasile
Design start      12.03.96    Design term    14.03.96
Sent by           ANMESJ task
Received by       AVARII task
Structure     {
 nr_statie: integer 1..7   // numarul statiei unde s-a citit
//    avaria
 nr_avarie: integer 1..42  // numarul logic al avariei
 st_avarie: integer 0..1   // starea finala a comutarii
//    avariei
 }           ;
```

- comută prin comanda **Design status** pe starea **Code**, ceea ce înseamnă că selectarea implicită pe **DFD** va fi a ferestrelor de cod ;
- selectează rând pe rând toate entitățile software și deschide ferestrele de cod asociate; programul realizează o inițializare a acestor ferestre cu șabloane extrase din fragmentele de cod disponibile în fișierul ***.TPL**, pe baza deducțiilor posibile din **DFD** și din specificațiile introduse anterior ;
- editează și completează codul-sursă cu prelucrările necesare : pe tot parcursul editării codului-sursă, poate selecta și fereastra asociată entității curente, care afișează specificațiile entității, astfel activitatea de codificare fiind mult ușurată.

Dăm în continuare un **exemplu de cod-sursă pentru un task** (împreună cu fragmentele de cod-sursă asociate, dispuse conform schemei generale prezentate mai sus) :

```
...
TSK_IDENT i_avarii; // cod generat la definirea taskului
```

```
...
P_IDENT p_an_av;     // cod generat la definirea conductei
                     // dintre taskurile t_anmesj si t_avarii
                     // (in timpul generarii taskului t_anmesj)
P_IDENT p_av_ex;     // generat la definirea conductei dintre
                     // taskurile t_avarii si t_exav
                     // (in timpul generarii taskului t_avarii)
RES_IDENT r_av_lc;   // generat la definirea blocului resursa
                     // (in timpul generarii taskului t_avarii)
MB_IDENT mb_av_ty;   // generat la definirea mesajului
                     // (in timpul generarii taskului t_avarii)
MB_IDENT mb_av_lp;   // generat la definirea mesajului
                     //  pt. imprimanta
                     // (in timpul generarii taskului t_avarii)
...
TASK t_avarii(void) // codul corpului taskului este generat la
                     // click pe entitatea respectiva din DFD
{
typedef struct {     // aceasta structura este dedusa din
                     //      specificatii
     int  nr_statie;// numarul statiei unde s-a citit
                     //    avaria
     int  nr_avarie;// numarul logic al avariei
     int  st_avarie;// starea finala a comutarii
} p_av;              // numele acesta este copiat de la
                     //   specificatii
typedef struct {
...              // aici sunt definirile pentru structurile
                 // folosite de celelalte
                 //  functii ale executivului, apelate in cadrul
                 //   taskului
_repeat {        //   corpul propriu-zis al taskului
//   sectiunea input

     _p_receive(p_an_av, p_av); // cod dedus din specificatii
                                //     (sau din DFD)
//   sectiunea processing

     // aici sunt prelucrarile propriu-zise (vezi
     //     specificatiile) care trebuie
```

```
      //        editate de proiectant

//    sectiunea output

      _p_send(p_av_ex, p_av);  // cod dedus din specificatii
                               //    (sau din DFD) - info. trimisa
                               //     spre taskul t_exav
      _r_wait(r_av_lc, 0);     // dedus din specificatii
                               //    sau DFD
           //  aici este loc pentru manipularea listei
           //   protejate de blocul resursa
           //   definit
      _r_signal(r_av_lc);              // perechea pt. _r_wait
      _mb_send(mb_av_ty,buff_m_ty);    // mesaj spre driver
                                       //    consola
      _mb_send(mb_av_lp, buff_m_lp);   // mesaj spre driver
                                       //  imprimanta

   } _forever;        // sfirsitul corpului propriu-zis
}
...
void main(void)              // inserat la initializare
{                            // inserat la initializare
   _install();               // inserat la initializare
...
   _create(i_avarii, t_avarii, 2, 8, i_avarii); // inserat la
                             //  definirea taskului t_avarii
...
   _p_create(p_an_av, i_anmesj, i_avarii);    // inserat la
                             //  definirea taskului t_anmesj
   _p_create(p_av_ex,i_avarii,i_exav);        // inserat la
                             //  definirea taskului t_avarii
   _r_create(r_av_lc, i_avarii, i_lcav);      // inserat la
                             // definirea taskului t_avarii
   _mb_create(mb_av_ty, i_avarii, d_tty);     // inserat la
                             // definirea taskului t_avarii
   _mb_create(mb_av_lp, i_avarii, d_lpt);     // inserat la
                             // definirea taskului t_avarii
...
   _activate(i_avarii);   // inserat la definirea
```

```
                        //   taskului t_avarii
...
  _run(10000);          // inserat la initializare (cf. param.
                        // introdus de proiectant la rubrica
                        // Rtclock de la specificatii generale)
}                       // inserat la initializare
```

Evident, la crearea unui task, trebuie automat inserate în codul-sursă fragmentele de cod asociate la începutul fişierului (macro-ul **TSK_IDENT** ...), respectiv în programul *principal* (**_create**(...) şi **_activate**(...)). Astfel, la sfârşitul etapei de parcurgere a editării codului tuturor entităţilor software, aplicaţia este complet disponibilă. Apelurile macro-urilor de definire a structurilor asociate apelurilor functiilor executivului sunt si acestea automat generate, după convenţia că fiecare asemenea macro, respectiv fiecare apel de iniţializare pentru aceste structuri, este inserat în codul-sursă în momentul definirii (prin click pe taskul respectiv, în timpul proiectării codului) taskului *emiţător* sau "iniţiator" al unei comunicări sau sincronizări. Generarea de cod trebuie deci să afecteze întregul cod-sursă, nu numai corpul propriu-zis al taskului.

La setarea comutatorului **Design status** pe poziţia **Code**, se generează un fişier-cadru care cuprinde *scheletul* aplicaţiei, conform schemei generale prezentate mai sus (respectiv, cod marcat cu comentarii specifice - *generat la initializare* - în exemplul de task, prezentat. În acest cadru va fi inserat codul-sursă al aplicaţiei, în succesiunea sugerată de comentariile de mai sus şi de exemplul de task.

*Instrumentul prezentat oferă un set de facilităţi specifice pentru instrumente **CASE**, care permit dezvoltarea simplă şi intuitivă, prin programare grafică (**DFD**), a programelor multitasking pentru conducere de proces, pe baza principiului (specific lucrării de faţă) de grupare a programelor în **clase de aplicaţii**. Instrumentul oferă şi acces la o **bază software** de componente şi oferă şi **instrumente de evaluare software** care implementează unele măsuri software pentru reutilizare, **propuse în această lucrare** sau preluate din bibliografie.*

8.2. Instrument CASE pentru dezvoltarea aplicaţiilor locale de conducere de proces şi instrumentaţie, pentru microsisteme cu microcontrollere

Prezentul paragraf prezintă un al doilea mediu integrat de dezvoltare pentru aplicaţii care sunt destinate rulării pe microsisteme cu microcontrollere. Programul a fost folosit pentru microcontrollere din familia **8051** [PHi95] dar poate fi la fel de bine folosit şi pentru alte

microprocesoare sau microcontrollere, întrucât structura şi logica internă a mediului permite acest lucru. Aplicaţiile care pot fi dezvoltate sub acest mediu **CASE** sunt din categoria delimitată în § 7.1.

Instrumentul **CASE** descris este destinat rulării sub **MS Windows** şi oferă facilităţi specifice **CASE**, la care se adaugă unele funcţii necesare testării şi depanării aplicaţiilor obţinute. Astfel, utilizatorul are posibilitatea obţinerii codului executabil, apelând la facilităţile obişnuite ale unui mediu de programare : manipulare de fişiere, editare, apelul *"sculelor"* (asambloare, linkeditoare, convertoare de cod), diverse setări (inclusiv în sensul configurării denumirilor şi modului de întrebuinţare a programelor specifice microcontrollerului - respectiv cross-asambloare, linkeditoare, conversii). Apoi, pentru testarea programului obţinut, acesta este trimis în calculatorul-ţintă prin legătura disponibilă (de obicei, legătură serială standard **RS-232**), iar apoi testarea propriu-zisă are loc prin dialogul dintre microsistem şi calculatorul-host. Evident, este necesară implementarea unor comenzi specifice unor astfel de facilităţi de *mediu integrat de dezvoltare* - **MID** (***Integrated Developement Environment*** - **IDE**) pentru sisteme cuplate cu sistemul de dezvoltare.

Unele trăsături specifice instrumentelor **CASE** sunt determinate de facilitatea specifică de *navigare* printr-o bibliotecă de texte în stil *hypertext*. Aceste texte se referă la descrierea facilităţilor standard oferite de modulele software din baza proprie (dar care poate fi încărcată de utilizator, deci sistemul este foarte flexibil). La fiecare selectare a unui subiect interesant pentru proiectant (deci care corespunde unor specificaţii ale programului care trebuie proiectat) se poziţionează automat elementele unei matrici de selecţie. La ieşirea din dialog, utilizatorul poate să ceară **generare**, în care caz mediul **CASE** selectează automat din **baza software** modulele indicate prin matricea de selecţie. Astfel, o primă versiune a programului dorit este imediat disponibilă.

Programul prezentat, aşa cum s-a văzut, are atât caracteristici de instrument **CASE**, cât şi de **MID**. În cele ce urmează îl vom caracteriza prin încadrarea la categoria **MID**. Programul prezentat a fost denumit **Instrument Designer**.

a) Concepţia generală a programului

Programul prezentat este conceput pentru a rula sub **MS Windows** şi manipulează următoarele **categorii de informaţii şi fişiere asociate** [Sto96-S] [Rac96]:

- fişier - *hypertext*, ***.HTM**, care conţine baza de ***hypertext*** pentru asistarea proiectării; operatorul navighează prin baza *hypertext* şi selectează funcţiile dorite, rezultând astfel o matrice de indecşi, deduşi din poziţia în bază a funcţiilor marcate; la deschiderea unui fişier-*hypertext*, automat este deschis şi fişierul-*generic* asociat ;

- fişier - generic, ***.GEN**, conţinând o colecţie de funcţii **BIOS** (în varianta realizată, funcţiile prezentate în § 6.4), alte funcţii şi fragmente de program corespunzătoare funcţiilor descrise în baza *hypertext*, funcţii şi fragmente indexate astfel încât să corespundă descrierilor din bază; după navigarea prin baza *hypertext*, prin comanda de **generare, MID** selectează funcţiile corespunzătoare indecşilor marcaţi, obţinându-se astfel codul-sursă (cvasi)complet al aplicaţiei dorite ;
- fişier - generator, ***.DLL**, care conţine funcţiile de **generare** a codului-sursă, pe baza fragmentelor de cod conţinute în fişierul ***.GEN** şi a indecşilor selectaţi prin navigarea în fişierul-*hypertext* ; acest fişier este deschis şi încărcat odată cu fişierele prezentate anterior ;
- fişiere - sursă , ***.ASM**, care conţin cod în limbaj de asamblare, asociat aplicaţiilor în lucru ; aceste fişiere sunt create după navigarea prin baza *hypertext*, la o comandă specifică de generare, dar pot fi şi create şi editate distinct de utilizator ;
- fişiere - text, ***.TXT**, care includ acele fragmente de text din fişierul - *hypertext* care au fost selectate de utilizator la navigarea prin baza *hypertext* şi care deci corespund unor funcţii dorite de acesta şi incluse în codul-sursă ;
- fişier - informaţii - proiect, ***.IFP**, conţine informaţii generale despre un proiect în curs, inclusiv numele şi localizarea fişierului-sursă şi a eventualului fişier-text asociat ;
- fişier de configurare, ***.INI**, conţinând setările selectate de utilizator, cum ar fi numele fişierului ***.HTM**, numele fişierului ***.DBF**, parametrii comunicaţiei seriale, informaţii de specificaţii, etc.;
- fişier de stare a consolei, ***.DSK**, care permite reluarea programului din punctul unde a fost întrerupt la dorinţa utilizatorului şi conţine informaţii despre starea programului la părăsirea acestuia;
- fişier - dicţionar ***.DBF**, care conţine surse de module software (funcţii şi fragmente-tip de programe, corespunzătoare unor funcţii specifice de sisteme de măsură şi interfaţă cu procesul) ; aici, utilizatorul poate include acele module de program care i se par semnificative pentru constituirea unui depozit de module software reutilizabile şi de aici poate să extragă module în vederea reutilizării pentru proiecte noi ; fiecare fragment este însoţit de descrieri ; dacă modulul de program a fost extras din fişierul ***.GEN**, atunci automat textul asociat din baza *hypertext* va fi şi el asimilat specificaţiilor şi inclus aici ;

Din punctul de vedere al utilizatorului, mediul prezentat are aspectul unui **MID** clasic, deci prezintă o bară-meniu "*pull-down*", care permite selectarea acţiunilor dorite (v. Fig. 8.3).

Comenzile meniului sunt :

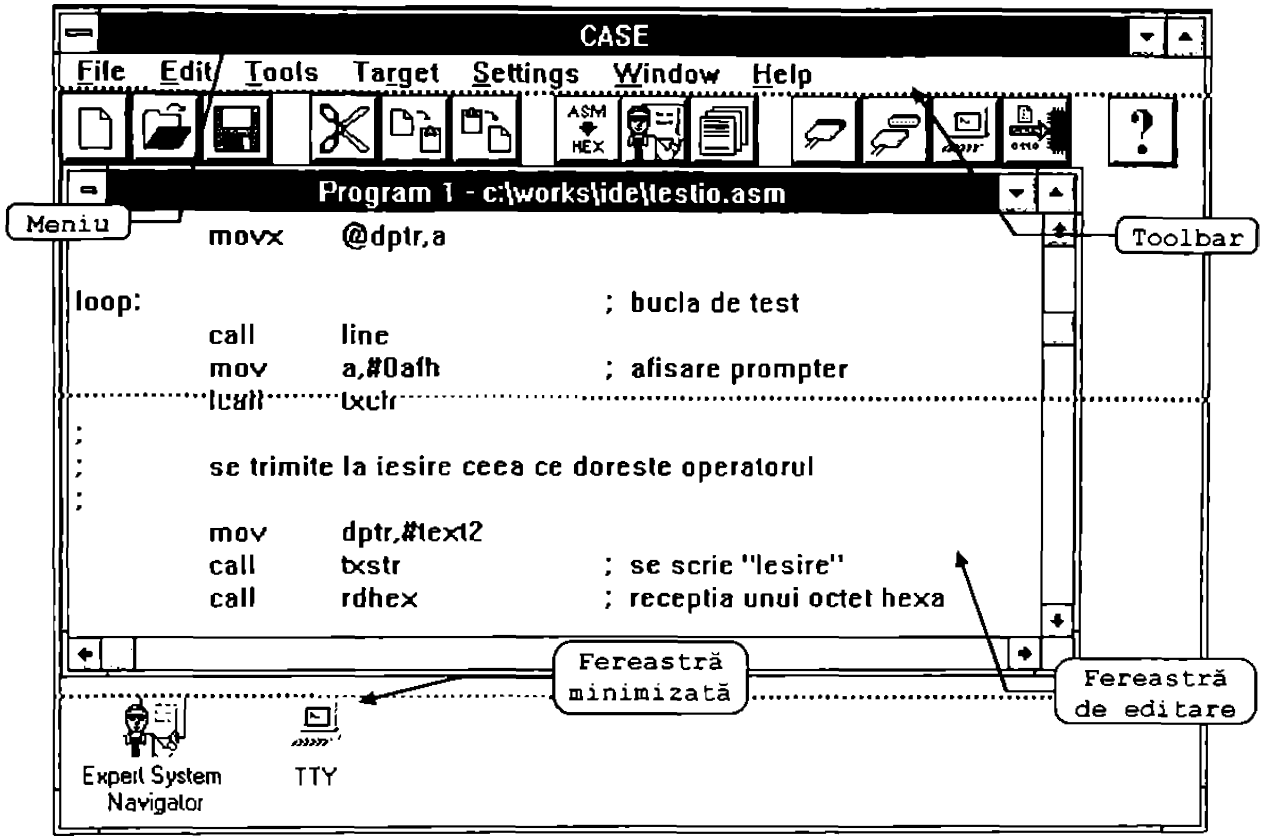


Fig. 8.3. Aspectul consolei programului **Instrument Designer** (meniul principal)

- **File** - manevrează fișiere-*sursă* și este o comandă "clasică", asemenea celor prezente în mediile **Borland C** sau în aplicațiile **Windows**, deci activează un submeniu care realizează următoarele:
  - **New** - determină deschiderea unui fișier-sursă nou (implicit, se cere editarea informațiilor generale asociate) ;
  - **Open** - deschide un fișier existent ;
  - **Save** - salvează un fișier-sursă editat ;
  - **Save as** - salvează un fișier sub nume nou ;
  - **Delete** - șterge un fișier-sursă (implicit, fișierul de informații asociat) ;
  - **Version** - fixează un fișier ca și versiune a unui proiect ;
  - **Backtrack** - încarcă o versiune precedentă (din *istoria* proiectului) ;
  - **Status** - afișează starea proiectului, referitoare la evoluția acestuia (zile rămase până la termenul final, numărul de module definite, etc.) ;
  - **Quit** - ieșire din program.

Comanda permite și deschiderea și editarea unor fișiere-text.

• **Edit** - este o comandă identică celei prezente în **MID** "clasice", deci cuprinde funcţii de editare de tipul :

- **Undo** - şterge ultima operaţie de editare ;
- **Cut** - copiază în "*clipboard*" şi şterge o porţiune de text marcată, din fişierul curent ;
- **Copy** - copiază în "*clipboard*" o porţiune de text marcată, din fişierul curent;
- **Paste** - inserează textul din "*clipboard*" în fişierul sursă ;
- **Delete** - şterge textul marcat
- **Find** - comandă de căutare a unui şir de caractere ;
- **Replace** - comandă de căutare şi înlocuire a unui şir de caractere ;
- **Next** - repetă operaţia de căutare.

• **Tools** - este o comandă care permite accesul la programele de obţinere a codului executabil, respectiv asambloare, linkeditoare, convertoare de cod, baza software şi sistemul expert:

- **Assemble** apelează asamblorul preferat al utilizatorului (care poate fi găsit pe baza informaţiilor furnizate de utilizator la o comandă de la **Settings**) ;
- **Link** lansează likeditorul ;
- **Convert** realizează conversia codului executabil al programului în cod transferabil (**HEX-INTEL**) prin legătura serială ;
- **Build** are ca efect cumularea succesivă a efectelor comenzilor de submeniu de mai sus ;
- **Export** - determină generarea unui fişier **ASCII** conţinând fişierul text şi fişierul informaţii concatenate ;
- **User command** lansează în execuţie o aplicaţie specificată de utilizator la **Settings** ;
- **Expert system** - activează navigarea prin baza *hypertext* şi generarea de cod. Afişează o fereastră de navigare cu subcomenzile:
  - **Search** - caută după un cuvânt-cheie anume ;
  - **Index** - afişează o listă de indecşi de căutare ;
  - **Generate** - genereazâ codul sursă, şi textele explicative însoţitoare separat într-un alt fişier ;
  - **Report** - afişează un sumar al denumirilor funcţiilor incluse în sursa generată .
- **Dictionary** - este comanda de acces la baza software (v. Fig. 8.4). Această comandă activează o fereastră de dialog cu o listă de funcţii disponibile, cu descrierea lor şi subcomenzile :
  - **Copy** -copiază în "*clipboard*" rutina selectată din listă ;

- **Edit** - lansează o fereastră de editare a numelui, descrierii şi a codului sursă al funcţiei selectate ;
- **Insert** - inserează o nouă funcţie în baza software, care poate fi editată mai departe cu Edit ;
- **Delete** - şterge funcţia curentă.

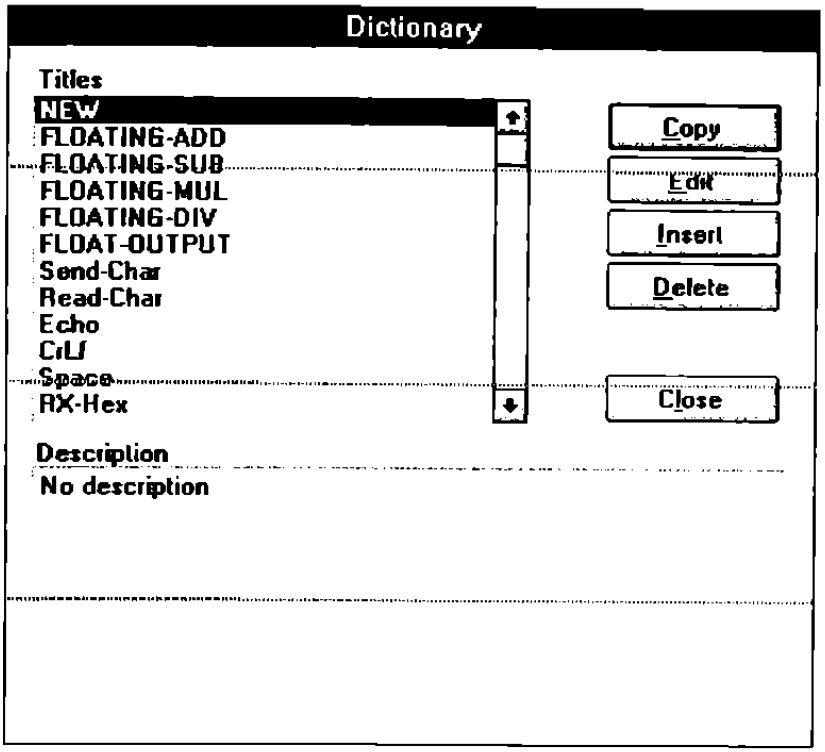


Fig. 8.4. Depozitul software (Dictionary)

Prin apelarea bazei software, se consideră ca aparţinând implicit de aceasta, funcţiile şi fragmentele de program din fişierul-generic ***.GEN** şi textele descriptive corespunzătoare din fişierul-*hypertext* ***.HTM** ;

• **Target** - este comanda care pune în legătură (prin intermediul unităţii seriale) calculatorul-gazdă a **MID**, cu microsistemul-ţintă, pe care va rula aplicaţia obţinută ; utilizatorul are la dispoziţie următoarele facilităţi :

- **Open link** - deschiderea legăturii seriale cu microcalculatorul-ţintă ;
- **Close link** - închiderea legăturii seriale ;
- **Identification** - lanseaza procesul de identificare a microcalculatorului ;
- **Open TTY** - deschiderea unei ferestre-ecou pentru unitatea serială ;
- **Load** - comanda de încărcare a unui fişier în format transferabil (**HEX**) ;
- **Monitor** - intrarea într-un regim de depanare cu comenzi standard, de nivel logic (prin intermediul fişierului ***.MON**, comenzile monitorului serial al

calculatorului-țintă sunt transformate în comenzi accesibile utilizatorului într-un mod standard, prin intermediul unor butoane), ca de exemplu:

- **Load** - încărcare fișier transferabil ;
- **Display** - afișează o zonă de memorie (internă, externă de date, externă de cod) ;
- **Bit** - realizează vizualizarea și eventual modificarea biților din zonele accesibile ca atare ;
- **Substitute** - afișcază și modifică locații de memorie (internă, externă);
- **Registers** - afișează și modifică registrele și bank-urile de registre ;
- **Go** - lansează în execuție un program ;
- **Trace** - execută un program cu trasare ;
- **Input** - vizualizare port ;
- **Output** - scriere pe port ;

Aici se impune observația că pentru diverse microprocesoare sau microcontrollere unele dintre aceste comenzi pot să nu fie active (spre exemplu, pentru microprocesorul **Z80**, memoria este structurată altfel decât la microcontrollerul **8051**, deci nu are sens tratarea distinctă a memoriei, specifică pentru **8051**) , în acest caz, comenzile respective sunt invalidate de o structură specifică de date din fișierul de configurare.

- **Master** - deschide o fereastră de afișare și o aplicație-master pentru comanda aplicației dezvoltate ; informațiile transmise prin legătura serială sunt afișate distinct în fereastra asociată.

• **Settings** este o comandă de stabilire a unor circumstanțe de funcționare a **MID** ; după activarea acestei comenzi, utilizatorul poate realiza următoarele acțiuni (v. Fig. 8.5) :

- **System** - setări specifice mediului de dezvoltare ;
- **Build** - permite definirea denumirii, căii și modului de apelare a asamblorului, linkeditorului, convertorului folosit, precum și a programului definit de utilizator (v. Fig. 8.6) ;
- **Expert sytem** - permite setarea unor parametri de funcționare a sistemului expert ;
- **Dictionary** - permite setarea unor parametri de funcționare ai bazei de date asociate (depozitul de componente reutilizabile) ;
- **Serial link** - permite definirea, de către utilizator, a parametrilor de comunicație serială cu calculatorul-țintă (viteză, paritate etc., v. Fig. 8.7) ;
- **TTY** - permite setările legate de fereastra-ecou ;

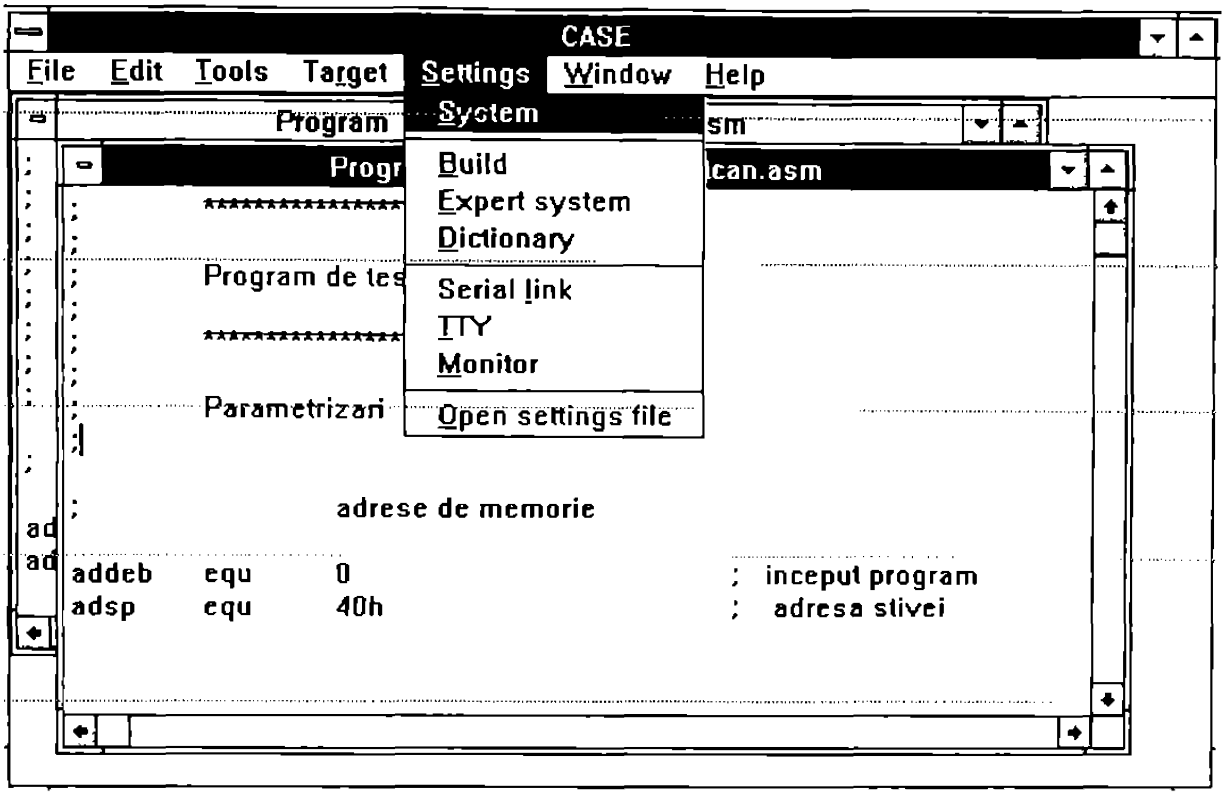


Fig. 8.5. Comanda **Settings** a programului **Instrument Designer**

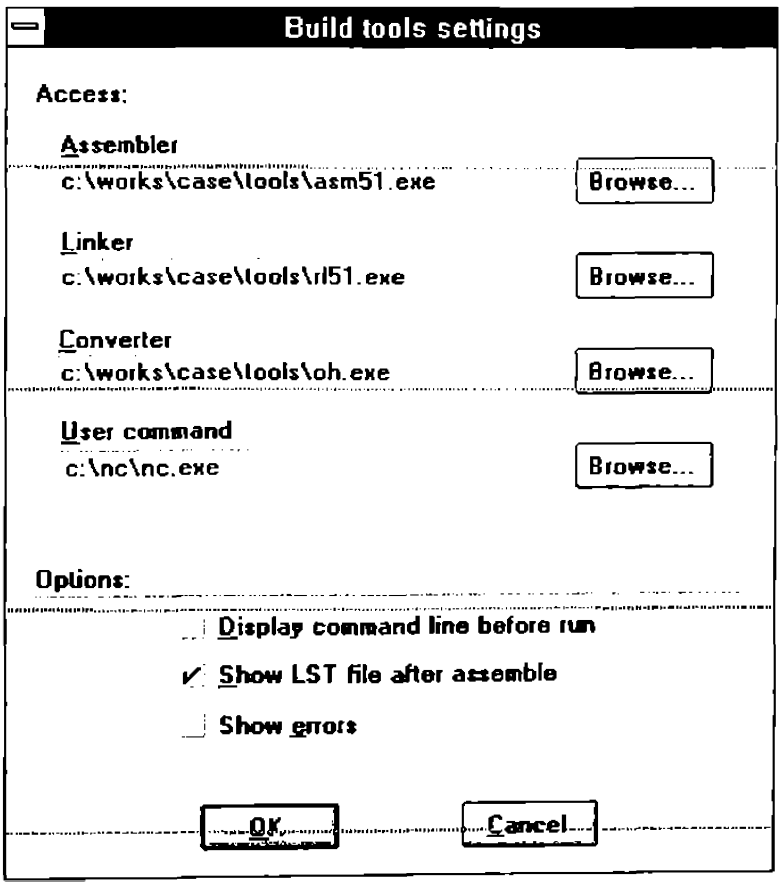


Fig. 8.6. Fereastra de dialog a setărilor "sculelor" *(Tools)*

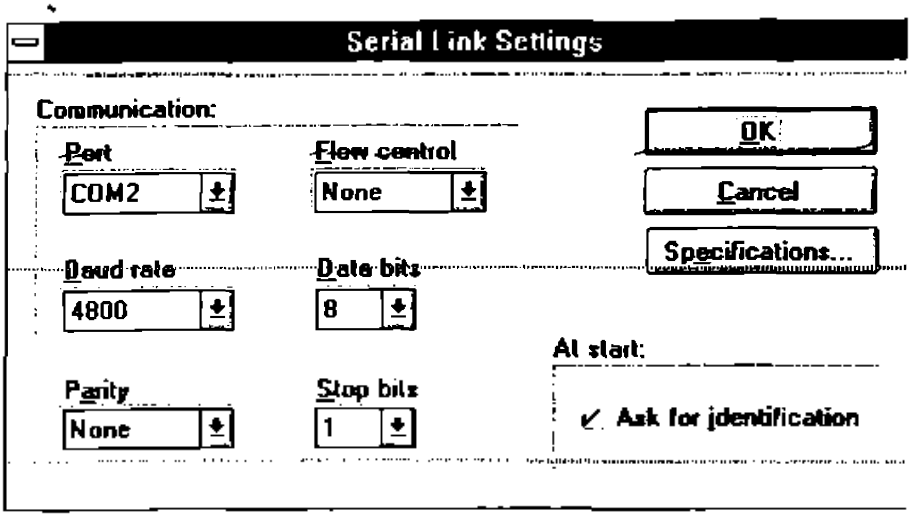


Fig. 8.7. Setarea parametrilor legăturii seriale (Serial link)

- **Monitor** - permite editarea specificațiilor legate de monitor ;
- **Open settings file** - deschide un nou fișier ***.INI**, astfel se poate realiza foarte simplu adaptarea programului la un alt tip de calculator țintă, toate setările necesare fiind conținute în acest fișier.

• **Window** și **Help** sunt comenzi standard ale programelor care rulează sub **Windows,** oferind facilități standard de aranjare a ferestrelor, respectiv informare despre utilitățile programului **Instrument Designer**

b) Structura programului

Programul este scris în limbajul **Borland C++ 3.1** pentru **Windows**, orientat spre obiecte, folosind **OWL** (*Object Windows Library*). Aceasta este de fapt o bibliotecă de clase predefinite pentru elementele grafice ale lui **MS Windows**: ferestre, meniuri, butoane, bare de rulare, căsuțe de marcare, linii de editare etc. Aceste clase încapsulează datele și funcțiile necesare gestionării elementelor de mai sus, și pe lângă asta realizează controlul mesajelor. Funcțiile membre ale claselor apelează funcții **Windows** de nivel scăzut, oferind programatorului o interfață mai prietenoasă și mai ușor manevrabilă.

Programul este alcătuit din mai multe fișiere, incluse într-un fișier "*project*", care sunt compilate separat și apoi linkeditate, și anume :

- **case.cpp** - conține modulul principal de lansare a programului ;
- **mainwin.cpp** - definește clasa ferestrei principale a programului ;
- **tlywin.cpp** - definește clasa ferestrei de ecou ;
- **dict.cpp** - definește clasa ferestrei de dialog a dictionarului ;
- **expsys.cpp** - definește clasa pentru sistemul expert ;

- **settings.cpp** - defineşte clasele pentru ferestrele de dialog ale setărilor ;
- **case.rc** - fişierul resursă, care conţine şabloanele pentru meniul principal, ferestrele de dialog, *short-cuts*, etc.
- **case.def** - descrie opţiunile de generare a aplicaţiei .

Pe lângă aceste fişiere, proiectul mai foloseşte următoarele fişiere "*header*":

- **case.h** - conţine directivele **#define** pentru definirea constantelor utilizate ;
- **classdef.h** - conţine declarările claselor şi a funcţiilor globale.

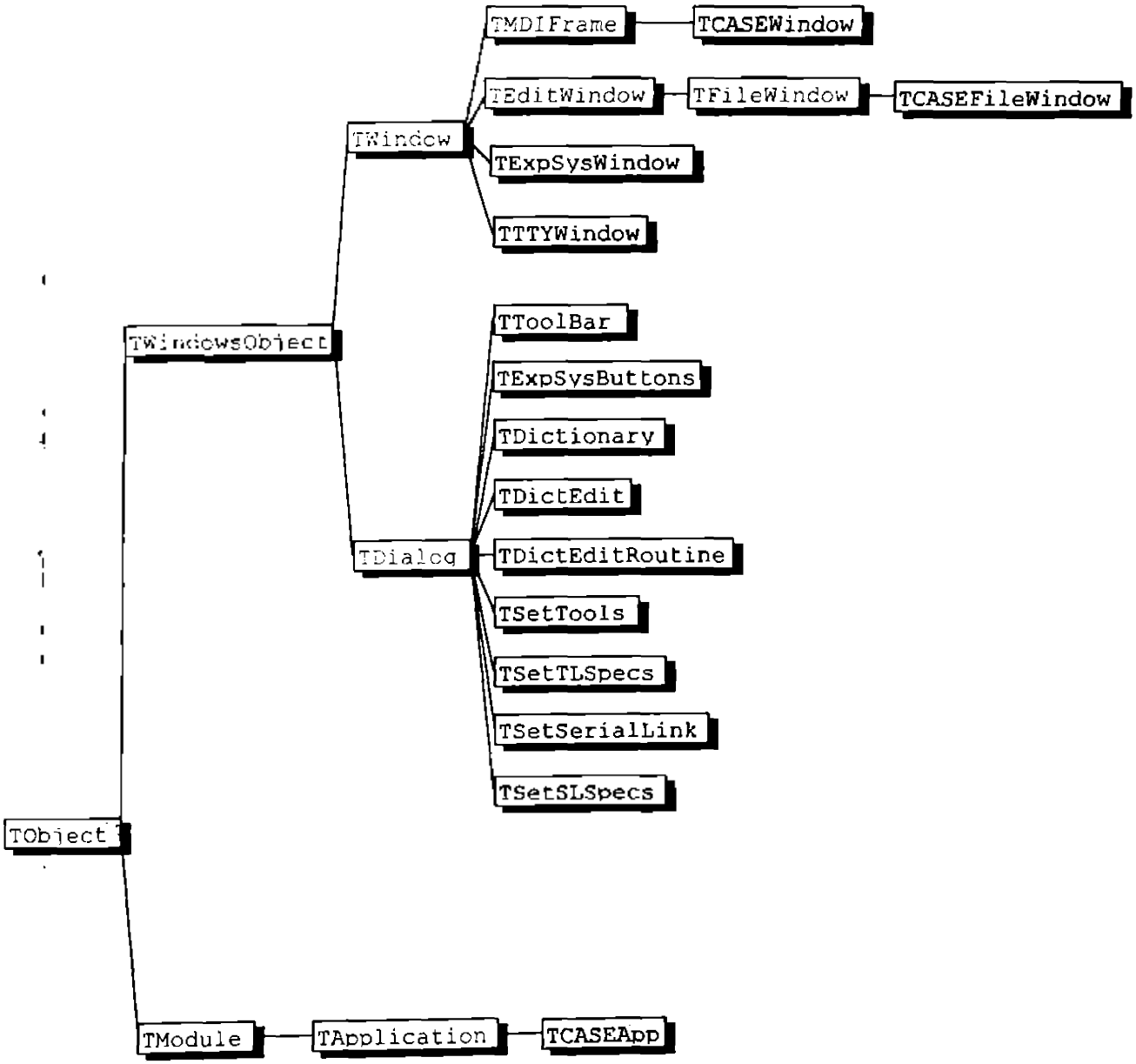


Fig. 8.8. Ierarhia de clase folosită de programul **Instrument Designer**.

În Fig. 8.8 este prezentată ierarhia de clase a programului descris. Prin dreptunghiuri albe sunt reprezentate clasele aparţinând ierarhiei **OWL** iar prin dreptunghiuri gri sunt reprezentate clasele specific definite ale aplicaţiei. O prezentare mai detaliată a claselor şi a resurselor este dată în Anexa 4.

c) Utilizarea programului

Să presupunem că proiectantul doreşte să realizeze, pentru structura tipică de sistem de achiziţie, prezentată în Fig. 6.6 (fără tastatură şi afişaj), o aplicaţie de instrumentaţie. În acest caz, proiectantul deschide, cu comanda **Expert System**, fişierul-*hypertext* asociat aplicaţiilor de instrumentaţie din clasa dorită şi navighează prin acesta pentru a întruni dezideratele dorite. După ce este mulţumit cu ceea ce a obţinut, dă comanda de generare şi automat se cere numele fişierului-sursă, care va fi completat cu fragmente de cod-sursă corespunzătoare funcţiilor dorite de utilizator

După completarea numelui fişierului, se deschide o fereastră de informaţii generale, care trebuie completată de utilizator, ca în exemplul de mai jos :

Name: - **Abbreviated:** ACHID51
- **Complete:** Sistem de achizitii de laborator cu 8051

Brief description: Sistem de achizitii comandat de PC prin legatura seriala RS 232, realizind achizitie analogica comandata sau monitorizare pe un numar precizat de intrari analogice si/sau generare de semnale de valoare comandata din PC

Keywords: instrumentatie, achizitii, microcontrollere

Project manager: Stoicu-Tivadar Vasile

Project design team: Racz Sandor

Client: Departamentul de Automatica si Informatica Tehnica

Project start: 01.03.1996 **Project term:** 01.05.1996

Target computer: 8051 cu 2 ko RAM, 8 Ko EPROM si ADC AD 582 cu MUX cu 16 intrari

Peripherals: - **Standard:** 1 x RS 232
- **Non-standard:** 16 analogic inputs: AD 582
1 analogic output: DAC 08
8 numeric inputs: I 8255
8 numeric outputs: I 8255

Description:

Programul citeste unitatea seriala si interpreteaza comenzile care vin de la PC. Raspunde la comenzi de stabilire a regimului (achizitie comandata, monitorizare, generare de semnal, combinatii), comenzi de setare de parametri (canal, numar de valori, perioada de esantionare), comenzi de raportare (a starii, a valorilor citite), comenzi de start/stop.

Daca a primit o comanda de stabilire parametri, realizeaza initializarile corespunzatoare si asteapta start. Dupa ce primeste start, programeaza ceasul de timp real si porneste achizitia apoi asteapta sfirsitul

```
achizitiei (sau generarii de semnal) si semnaleaza prin legatura seriala. La
cerere, trimite raportul prin unitatea seriala.
Source code:  ACHID51.ASM, ACHID51.H
Version:  3.2   Localisation:  C:\WORKS\APPL\APPL_ACHID
```

Din acest moment este disponibilă o primă versiune a codului-sursă . Utilizatorul poate să continue imediat prin editarea fişierului-sursă tocmai obţinut sau poate efectua orice alte acţiuni posibile cu acest program. Oricând, poate să revină la editarea fişierului-sursă, prin comenzile **File**, apoi **Open**. Editarea poate fi mai performantă folosind şi comenzile suplimentare de editare disponibile la opţiunea **Edit**.

Există şi o altă posibilitate de a obţine codul-sursă : operatorul creează un fişier-sursă prin comanda de creare **New** din opţiunea **File**. În acest caz, fereastra de informaţii generale se deschide de asemenea, pentru a fi completată de proiectant. Mai departe, utilizatorul poate să editeze fişierul "pe cont propriu" (fără ajutorul opţiunilor de asistare a proiectării) sau să apeleze la depozitul software, prin activarea accesului la acest depozit prin comanda **Dictionary**. Modulele de program găsite interesante pentru aplicaţia proiectată (respectiv, potenţial apte de refolosire) pot fi copiate în fereastra de editare prin intermediul *clipboard*-ului.

Odată codul-sursă obţinut, prin una sau alta din căile indicate, proiectantul trebuie să parcurgă etapele de obţinere a codului executabil (şi apoi, codului transferabil), de aceea va deschide fereastra de submeniu **Tools** şi va efectua asamblarea (**Assemble**). După eventualele corecturi şi reluări ale asamblării, până la eliminarea greşelilor de asamblare, se poate efectua linkeditarea (prin comanda **Link**) şi conversia în cod transferabil (**Convert**). Testarea se realizează prin transferarea codului transferabil în calculatorul-ţintă şi apoi lansarea în execuţie pe porţiuni. În acest scop, se deschide legătura serială (comanda **Open link**) şi se transferă fişierul cu comanda **Load**. Testarea propriu-zisă se poate realiza fie prin facilităţile standard **Monitor**, fie direct prin facilităţile monitorului serial disponibil pe calculatorul-ţintă (dacă operatorul este familiarizat cu acestea), prin deschiderea unei ferestre-ecou cu **Open TTY**. Ciclul de editare-obţinere de cod-transfer-testare se poate relua foarte simplu şi rapid, până la testarea completă a aplicaţiei. O facilitate în plus pentru testarea corectă o constituie comanda **Master**, prevăzută special pentru ca utilizatorul să poată lansa o aplicaţie-master pentru a lucra eventual în conjuncţie cu aplicaţia din calculatorul-ţintă, caz în care comunicaţia serială este de asemenea monitorizată şi sunt afişate informaţiile care circulă în ambele sensuri ale legăturii seriale.

Proiectantul trebuie să-şi configureze corect mediul de lucru. În acest scop, proiectantul de nivel înalt trebuie să editeze fişierele-*hypertext* şi *generic*, în funcţie de clasa de aplicaţii dorită. Utilizatorul de nivel înalt va trebui să adapteze programul la propriile condiţii prin

setarea configurației, prin aceasta realizând adaptarea la cerințele unui anumit modul calculator țintă, respectiv adaptarea la metodologia de lucru a propriei organizații software. Tot el va stabili parametrii de lucru ai legăturii seriale, conform parametrilor stabiliți la punerea sub tensiune a calculatorului-țintă, accesul la programele folosite de comenzile opțiunii **Tools** și parametrii de funcționare ai depozitului software. Toate aceste setări se realizează sub comanda **Settings**. Odată condițiile de lucru stabilite, proiectantul poate să treacă la proiectarea și testarea propriu-zisă, având la îndemână un instrument puternic de proiectare-testare. Proiectantul poate contribui la creșterea depozitului software, prin introducerea în depozit a modulelor software considerate interesante din punctul de vedere al reutilizării. Introducerea unui modul nou se realizează cu comanda **Insert** din cadrul ferestrei de dialog **Dictionary**.

Realizarea documentației este de asemenea ușurată de mediul prezentat: prin comanda **Export** de la opțiunea **Tools**, este generat un fișier **ASCII** conținând toate informațiile disponibile despre modulele de program, componente ale aplicației, care au fost preluate din depozitul software, au fost introduse acolo în timpul proiectării aplicației curente sau au fost obținute în urma navigării în baza *hypertext*. La acestea se adaugă informațiile cu caracter general, introduse de operator la crearea fișierului-sursă al aplicației. Mediul se dovedește astfel foarte versatil și util pentru dezvoltarea de aplicații mici de instrumentație sau de telesupraveghere de proces, de tipul celor din Fig. 6.6 (dar poate fi folosit foarte bine la dezvoltarea oricăror aplicații dedicate, care rulează pe microcalculatoare care dispun de o legătură serială. Programul poate fi însă ușor adaptat și pentru altfel de legături (spre exemplu legătură paralelă), dispunând și la acest capitol de multă flexibilitate.

Un exemplu concret de utilizare a mediului **Instrument Designer** este prezentat în [Rac96].

Mediul **CASE** prezentat oferă toate facilitățile necesare dezvoltării de aplicații mici de instrumentație sau supraveghere de proces în limbaj de asamblare, respectiv editarea, generarea de cod transferabil, transferarea în calculatorul-țintă, depanarea (incluzând aici ca element de originalitate monitorizarea aplicației dezvoltate în conjuncție cu o aplicație **master**), asigurând în plus o serie de facilități specifice **reutilizării software** : acces la o bază **software**, respectiv **generarea de aplicații** într-un mod original, prin navigarea într-o bază ***hypertext***.

8.3. Concluzii

Tehnologiile moderne de dezvoltare a programelor nu mai pot fi concepute fără utilizarea instrumentelor software puternice și flexibile : medii de programare sofisticate (precum ultimele produse ale firmelor **Microsoft** sau **Borland** pentru dezvoltarea programelor

în limbajul C) sau instrumente CASE care asistă cât mai multe etape ale ciclului de viaţă al programelor. Acest capitol a prezentat astfel de instrumente CASE, destinate dezvoltării aplicaţiilor din două categorii specifice interacţiunii programelor cu procesele : aplicaţii multitasking, scrise în limbajul C, respectiv aplicaţii mici de instrumentaţie sau supraveghere de proces care rulează pe microsisteme cu microcontrollere, aplicaţii scrise în limbaj de asamblare.

Cele două programe descrise îşi justifică prezenţa în această lucrare cel puţin din două motive :

- utilizarea instrumentelor ***CASE contribuie direct la creşterea gradului de reutilizare*** *al programelor, prin facilităţile tehnologice pe care le oferă în mod uzual în acest sens ;*

- programele prezentate implementează în plus faţă de facilităţi uzuale, specifice instrumentelor ***CASE, funcţii speciale*** *pentru* ***îmbunătăţirea posibilităţilor de reutilizare*** *a programelor.*

Funcţiile specifice, amintite, constau în :

1. accesul la **baze software de module reutilizabile**,
2. implementarea de concepte specifice reutilizării (**clase de aplicaţii**),
3. includerea unor facilităţi de **generare de aplicaţii** şi de **evaluare a unor indici de reutilizare**.

Programele descrise sunt actualmente în faza de finisare. Caracteristicile celor două programe sunt prezentate în Tabelul 8.1. În tabel, elementele specifice **reutilizării software** au fost scoase în evidenţă prin **accentuare**, iar elementele de originalitate au fost scoase în evidenţă prin folosirea *caracterelor italice*.

Elementele de originalitate includ în primul rând **destinaţia** celor două medii **CASE**. Tratarea programelor din punctul de vedere al **claselor de aplicaţii** este o noutate pentru acest gen de programe, în special dacă sunt destinate dezvoltării de aplicaţii de conducere de proces multitasking, ca în cazul primului program. Această trăsătură de originalitate implică şi alte aspecte : editarea **DFD** şi posibilitatea **vizualizării** simultane a **programului generic** şi a programului editat. Noutăţi sunt şi funcţiile de **evaluare** pentru **gradul de reutilizare**, prezente în primul program. **Monitorizarea simultană a două aplicaţii şi a comunicaţiei dintre acestea** este un instrument deosebit de puternic şi de eficient pentru testarea aplicaţiilor de instrumentaţie şi supraveghere de proces, de genul celor dezvoltate cu mediul **CASE** prezentat în § 8.2, constituind o contribuţie originală. **Generarea de aplicaţii noi prin navigare prin bază *hypertext*** constituie o altă propunere originală, înglobată de asemenea în cadrul celui de-al doilea program.

Tabelul 8.1. Caracteristicile programelor CASE prezentate

Destinaţie	Facilităţi standard	Bază software	Dezvoltare/ depanare	Facilităţi avansate de dezvoltare de noi aplicaţii	Evaluare software
aplicaţii multitasking pentru conducere de proces, scrise în limbaj de nivel înalt	- creare de proiecte (includ şi documentaţia) - editare **DFD** -editare fişiere-sursă -editare descriere module -generare documentaţie - istorie (backtrack)	da (taskuri, funcţii, documentaţia aferentă)	- legătura cu un **IDE** - exportul de fişiere	**-implementarea conceptului de clasă de aplicaţii (asocierea mai multor programe în clase)** **- bază de module software**	**- evaluare de indici** pentru complexitatea programelor şi pentru **gradul de reutilizare**
aplicaţii mici de instrumentaţie sau supraveghere locală de proces, pentru microsisteme cu microcontrollere	-editare fişiere-sursă -editare descriere module -generare documentaţie -istorie (backtrack)	da (secvenţe de program, subrutine, rutine şi documentaţia aferentă)	-obţinere de cod transferabil -transferul programului în microsistemul-ţintă - depanare cu monitor utilizator sau cu monitor standard - monitorizarea funcţionării în conjuncţie cu o aplicaţie master	**-generare de aplicaţii prin navigare într-o bază *hypertext*** **- bază de module software**	nu

În concluzie, cele două programe prezentate acoperă necesităţile de proiectare pentru două nişe specifice ale domeniului conducerii de proces, oferind şi facilităţi puternice pentru ***creşterea gradului de reutilizare****.*

CAPITOLUL 9.
CONCLUZII ŞI PERSPECTIVE

Lucrarea de faţă tratează câteva aspecte din problematica **reutilizării software** în domeniul aplicaţiilor de conducere de proces. Reutilizarea reprezintă un aspect al ingineriei software care devine din ce în ce mai interesant pe măsură ce producătorii de software trebuie să se adapteze cererii tot mai mari de aplicaţii dintre cele mai diverse.

9.1. Rezultatele lucrării

Sintetizăm în continuare principalele rezultate obţinute în această teză :

Tratarea problemei reutilizării, implică, aşa cum s-a arătat în **Capitolul 1**, clasificarea aplicaţiilor în *clase*. Clasele înglobează aplicaţii cu un set de trăsături comune. Împărţirea aceasta, precum şi tot ceea ce e legat de manipularea noţiunii de clasă (variante, grad de reutilizare, etc.), este "vagă" (*fuzzy*) şi subiectivă. De aceea, în **Capitolul 4** se propune o abordare *fuzzy* pentru calculul gradului de reutilizare. Perspectiva *fuzzy* asupra domeniului, poate fi extinsă şi la alte concepte (s. ex. apartenenţa unui program la o clasă, relaţii *fuzzy*, matrice *fuzzy*, etc.). Asocierea unor mărimi la aspectele calitative şi măsurarea acestor mărimi nu este un concept nou. Principalele abordări ale determinării gradului de reutilizare, precum şi alte aspecte legate de măsuri software, sunt trecute în revistă în **Capitolul 3**, ceea ce constituie o bună fundamentare a dezvoltării de la **Capitolul 4**.

Abordarea **POO** face obiectul **Capitolului 2**. A fost necesară o astfel de delimitare întrucât această modalitate de proiectare este diferită de conceptele "clasice" ale programării structurate. Pe de altă parte, capitolul este necesar în structura lucrării deoarece **POO** este cea mai nouă abordare în reutilizare şi deci o parcurgere a principalelor repere ale acesteia este necesară într-o lucrare cu un asemenea subiect.

În Capitolul 5 este prezentat un început de formalizare a proiectării pentru reutilizare. Din punct de vedere practic au semnificaţie atât diversele măsuri software introduse în acest capitol cât şi clasificarea propusă în ultimul paragraf al capitolului. Ca urmare, vorbim de tehnici de speţa întâi (parametrizări în codul-sursă), a doua (tehnici şi structuri de program pentru reutilizare) şi a treia (**GA**). Tehnicile de speţa întâi, considerate elementare, nu au fost tratate în lucrare. În schimb, în

Capitolul 6 au fost date mai multe exemple privind utilizarea tehnicilor de speța a doua, cu ajutorul cărora aplicațiile din aceeași clasă pot fi obținute cu efort minim de reproiectare. Acest efort va viza de cele mai multe ori doar redefinirea unor structuri de date care descriu particularități de funcționare, mergând de la descrieri de interfețe până la implementarea diagramei de funcționare inclusiv. S-a dezvoltat și un exemplu referitor la realizarea decuplării dintre partea generală și cea specifică a unei aplicații. Sunt prezentate de asemenea elementele definitorii ale unor module **BIOS** pentru sisteme de instrumentație sau de interfață cu procesul, module concepute special pentru a permite implementarea cu efort minim a unei mari varietăți de aplicații.

Tehnicile de speța a treia sunt din ce în ce mai prezente, atât ca preocupare științifică, cât și pe piață, ceea ce se poate observa din cele câteva exemple prezentate în Capitolul 1. Programele **GA** sunt complexe. Proiectarea lor trebuie justificată prin cererea mare în domeniul acoperit. În **Capitolul 7** se propune o listă de cerințe pentru o clasă de aplicații de instrumentație cu microcontroller din familia **8051** și sunt descrise trei astfel de generatoare.

Capitolul 8 reprezintă o descriere a două medii **CASE** pentru accelerarea proiectării aplicațiilor de conducere de proces multitasking. Aceste medii **CASE** implementează o serie de funcții care contribuie la creșterea gradului de reutilizare.

În concluzie, structura lucrării urmărește o dezvoltare de tipul : **cunoștințe despre domeniu** (Cap. 1, 2, 3), **dezvoltări teoretice** (Cap. 4 și 5) respectiv **aplicații** (Cap. 6, 7, 8). Prezentul capitol de **Concluzii** încheie teza.

Pe scurt, **rezultatele** tezei sunt deci următoarele :

- **sinteza** în domeniu ;
- contribuțiile **teoretice** în domeniul abordării *fuzzy* a domeniului ,
- contribuțiile **teoretice** în definirea unui **formalism sistemic**, original, pentru domeniul abordat ;
- definirea unor **măsuri software** originale, cu aplicabilitate în proiectarea pentru reutilizare, măsuri definite pe baza **logicii** *fuzzy* sau prin prisma abordărilor teoretice **sistemice** ,
- descrierea unor **tehnici** de programare pentru aplicații mici de conducere de proces, care contribuie substanțial la creșterea gradului de reutilizare al programelor din această categorie;
- realizarea unor **GA** pentru domeniul instrumentației ;
- realizarea unor instrumente **CASE** puternice, care implementează facilități avansate pentru creșterea gradului de reutilizare.

Aceste rezultate sunt reflectate în contribuțiile originale ale tezei, prezentate în continuare.

9.2. Contribuţii originale

Studiul realizat înglobează o serie de tratări originale ale problemelor care constituie obiectul lucrării. Categorisim aceste contribuţii astfel :

A. Contribuţii în domeniul **sintezei** informaţiei din problematica reutilizării

B. Contribuţii cu caracter **teoretic** :

- în domeniul **măsurilor software**
- în domeniul **abordărilor *fuzzy***
- în domeniul **formalizării procesului de proiectare pentru reutilizare**

C. Contribuţii cu caracter **aplicativ** :

- în domeniul **tehnicilor şi structurilor de program pentru aplicaţii mici de conducere de proces scrise în limbaj de asamblare**
- în domeniul **generatoarelor de aplicaţii pentru sisteme de instrumentaţie**
- în domeniul **instrumentelor CASE pentru aplicaţii multitasking şi pentru sisteme de instrumentaţie**

Vom enumera în continuare contribuţiile originale pentru toate categoriile menţionate :

1. În domeniul **sintezei** :

1.1. Realizarea unei sinteze originale asupra aspectelor de actualitate ale domeniului reutilizării software, cu accent pe **clase de aplicaţii, SGBS, GA, instrumente CASE, modele în proiectare, avantaje şi probleme asociate** reutilizării software, aspecte proprii **sistemelor în timp real** (Cap. 1) ;

1.2. Completarea clasificării **Biggerstaff** cu instrumentele **CASE** (§ 1.11.1) ;

1.3. Formularea unor **concluzii** care să contureze principalele **direcţii** de investigare ale prezentei lucrări (§ 1.12, 1.13) ;

1.4. Elaborarea unei sinteze originale asupra trăsăturilor de reutilizabilitate care pot fi întâlnite în utilizarea paradigmei **TOO** (Cap. 2) şi a unei **scheme originale** de reprezentare a influenţei diverselor aspecte ale **TOO** asupra reutilizării (Fig. 2.5) ;

1.5. Formularea unei sinteze originale referitoare la **măsurile software** care pot fi utile în abordarea pentru reutilizare, atât în cadrul paradigmelor clasice, cât şi în cadrul **TOO**

(Cap. 3), bazată pe o evaluare **critică** a diverselor definiri (Cap. 3) ;

1.6. Realizarea unui studiu de sinteză asupra principalelor surse de informaţii în domeniul reutilizării şi al metricilor software, pe **INTERNET** (Anexa 1.3) ;

2. În domeniul **măsurilor software** :

2.1. Îmbogăţirea sensului a două măsuri software, în vederea utilizării acestora pentru evaluarea gradului de reutilizare (§ 3.2, A şi C) ;

2.2. Definirea unor noi măsuri software pentru gradul de reutilizare - **gradul de reutilizare de program** şi **gradul de reutilizare de clasă** - calculate prin folosirea logicii *fuzzy* (§ 4.1) şi definirea **gradului de reutilizare de la un program la altul** (§ 4.5) ;

2.3. Definirea unor măsuri software originale, utile în estimarea performanţelor proiectării pentru reutilizare în cadrul claselor de aplicaţii : **efortul de acces, efortul mediu de acces, gradul de accesibilitate de program, gradul de accesibilitate de clasă, viteza de fază a proiectării, viteza de grup a proiectării** (§ 5.3) ;

2.4. Definirea şi caracterizarea unei **semimetrici** peste clasele de aplicaţii (rel. (5.14) şi Teorema (5.25)) ;

2.5. Definirea unui **factor de calibrare** pentru *gradul de reutilizare de clasă* (5.11,a) şi a unei tehnici de determinare (calibrare) a acestui factor (5.36) ;

2.6. Definirea **normei reciproce de reutilizare** şi demonstrarea faptului că relaţia *fuzzy* indusă de aceasta este relaţie de echivalenţă (Teorema (4.14)) ;

2.7. Extinderea folosirii **distanţei Hamming** la submulţimile *fuzzy* de proprietăţi constituite din clase de aplicaţii (Definiţia (4.6)) .

3. În domeniul **abordărilor *fuzzy*** :

3.1. Propunerea unor **principii *fuzzy*** pentru calculul gradelor de reutilizare definite şi evidenţierea avantajelor lor (§ 4.2) ;

3.2. Propunerea unei modalităţi de **calcul recursiv** a gradelor de reutilizare definite în Cap. 4, prin aplicarea recursivă a unui algoritm bazat pe principiile menţionate mai sus, având avantajul că această modalitate de calcul ţine cont de structura unui sistem care poate fi reprezentat prin **DFD** (spre exemplu, sisteme de conducere de proces multitasking) ;

3.3. Definirea unui set de **reguli de inferenţă** (deduse din Tabelul 4.1) şi alegerea unui principiu de agregare pentru algoritmul *fuzzy* menţionat mai sus ;

3.4. Propunerea unor **criterii** pentru desemnarea setului de **programe semnificative** pentru aplicarea algoritmului de la § 4.2 ;

3.5. Identificarea, în cadrul claselor de aplicaţii, a unor mulţimi care pot fi asimilate

submulțimilor *fuzzy* de proprietăți din teoria *fuzzy* (Definiția (4.3)) ;

3.6. Delimitarea, în cadrul claselor de aplicații, a unor mulțimi care pot fi asimilate **mulțimilor *fuzzy*** din teoria *fuzzy* (Definiția (4.4)) ;

3.7. Descrierea unei modalități de **cuantificare** pentru **apartenența** unei variante la o submulțime *fuzzy* de proprietăți (§ 4.4) ;

3.8. Asimilarea modificărilor ce survin în timp în cadrul unor clase de aplicații, noțiunii de **submulțime *fuzzy* de proprietăți dependentă de timp** din teoria *fuzzy* și sublinierea caracterului **dinamic** al modelului *fuzzy* de apartenență a programelor la o clasă de aplicații ;

3.9. Definirea unui **criteriu de coeziune** pentru o clasă de aplicații, pe baza **distanței Hamming** (Criteriul (4.7)) ;

3.10. Folosirea gradelor de reutilizare definite anterior, pentru definirea unor **matrice *fuzzy*** (§ 4.5) ,

3.11. Stabilirea faptului că matricele *fuzzy* de la pct. 3.11 sunt exprimări ale unor **relații** de **reutilizare** (§ 4.5) ;

3.12. Demonstrarea unei **proprietăți** referitoare la valori numerice caracteristice pentru matricea *fuzzy* de la pct. 3.11 (Proprietatea (4.8)) ;

3.13. Interpretarea consecințelor practice ale **Teoremei de descompunere** (4.10) ;

3.14. Investigarea unor proprietăți (4.12) ale relației *fuzzy* propuse : demonstrarea proprietății de **reflexivitate**, respectiv demonstrarea, prin construirea unui **contraexemplu**, a faptului că relația nu este nici simetrică, nici antisimetrică;

3.15. Interpretarea aplicativă a proprietăților (4.12) în sensul că în practică nu poate fi realizată o **ordonare** a variantelor în cadrul clasei ;

3.16. Definirea unei relații *fuzzy* de **similitudine (echivalență)** (conform Definiției (4.13)) pe baza **normei reciproce de reutilizare** ;

3.17. Demonstrarea (teoremei (4.14)) proprietății de similitudine a relației *fuzzy* induse de definirea anterioară (adică demonstrarea proprietăților de **reflexivitate, simetrie** și **tranzitivitate** ale *normei reciproce de reutilizare*) ;

3.18. Interpretarea și realizarea **conexiunii** practice a consecințelor Teoremei (4.14) cu principiile de clasificare și **browserele** folosite în **SGBS** ;

4. În domeniul **formalizării procesului de proiectare pentru reutilizare** :

4.1. Abordarea originală a procesului de proiectare pentru reutilizare, materializată prin descrierea unui **formalism original**, bazat pe conceptele **sistemice** (Cap. 5) ;

4.2. Descrierea unui **model formal intrare-stare-ieșire** pentru proiectarea pentru reutilizare (§ 4.1) ;

4.3. Definirea **claselor deschise** (Observația (5.1,a)) ;

4.4. Propunerea unor criterii de **indexare** (ordonare) în cadrul clasei (Observația (5.1.b)) ;

4.5. Propunerea folosirii unui **timp abstract** pe axa timpului (Observația (5.2)) ;

4.6. Prezentarea a două **scheme de modelare** a procesului de proiectare pentru reutilizare (Fig. 5.1 și 5.3) ;

4.7. Descrierea **spațiului calităților** (Fig. 5.2) ;

4.8. Definirea unor **transformări** în *spațiul calităților* (5.7) ;

4.9. Găsirea unui *izomorfism conceptual* între *spațiul calităților* și *spațiul variantelor* ,

4.10. Interpretarea practică a unor situații ce decurg din analiza schemei propuse în Fig. 5.3 ;

4.11. Încadrarea proiectării pentru reutilizare în cadrul unei clase de aplicații ca un **sistem dinamic** numit **SPC** (Definiția (5.3)) ;

4.12. Extrapolarea unor concepte din teoria sistemelor, referitoare la fază accesibilă, fază controlabilă și controlabilitate completă la specificul **SPC** (§ 5.2) ;

4.13. Enunțarea condițiilor în care un **SPC** devine **automat finit** (Proprietatea (5.9)) ;

4.14. Interpretarea procesului de proiectare de către un operator uman ca o deplasare prin transformări **ortogonale** în *spațiul calităților* (§ 4.2) ;

4.15. Enunțarea și **demonstrarea** Teoremei (5.11) referitoare la accesibilitatea **SPC** ;

4.16. Demonstrarea proprietății de **observabilitate** a modelului trasnformărilor din *spațiul calităților* (§ 4.2) ;

4.17. Propunerea unei **clasificări** proprii pentru **SPC** (Clasificarea (5.12)) .

4.18. Dezvoltarea unei teorii referitoare la faze care aparțin aceleiași *clase de observare* sau *aceleiași clase de contructie* (§ 4.2) și demonstrarea faptului că *programul generic* este *neobservabil și necontrolabil* (Lema (5.17)) ;

4.19. Definirea unui operator în raport cu care o clasă este simultan *clasă de observare* și *clasă de construcție* (Teorema (5.18)) și interpretarea practică a acestei situații ;

4.20. Enunțarea Problemei fundamentale a proiectării în clasă (5.27) ;

4.21. Definirea unui **criteriu** *a posteriori* de apartenență a unui program la o clasă (Criteriul (5.28)) ; introducerea mărimii **normă de apartenență de clasă ;**

4.22. Definirea consistenței claselor de aplicații prin enunțarea unui **criteriu** de consistență (5.30) și introducerea noțiunilor : **deviație de consistență**, partea **tare consistentă** a unei clase ;

4.23. Definirea omogenității unei clase prin enunțul unui criteriu de **omogenitate** (5.31) și introducerea noțiunilor : **vecinii** unei variante, **normă de omogenitate locală,** **ecart de omogenitate,** apartenența **de drept** la o clasă ;

4.24. Definirea super-consistentei unei clase ca proprietate cumulativă (5.33) a celor anterioare și interpretarea practică a acestei situații ;

4.25. Definirea nucleului unei clase (Definiția (5.35)) și introducerea noțiunii de **eroare de merit** ;

4.26. Propunerea unui algoritm iterativ pentru construirea *nucleului* unei clase (§ 5.7) ;

4.27. Propunerea unei **clasificări** pentru **SPC** (Clasificarea (5.37)), folosită mai departe în aplicațiile practice prezentate.

5. În domeniul **tehnicilor și structurilor de program pentru aplicații mici de conducere de proces scrise în limbaj de asamblare :**

5.1. Descrierea unor structuri simple de date și de program care permit simularea unor concepte specifice POO, în domeniul aplicațiilor mici de conducere de proces, scrise în limbaj de asamblare, structuri concepute pentru a îmbunătăți gradul de reutilizare al programelor (§ 6.1) și descrierea unor **aplicații funcționale**, care folosesc aceste structuri ;

5.2. Descrierea unor structuri de date care implementează informațiile necesare rulării unui program complex de tip *panou de operare* (§ 6.2), care asigură un grad de reutilizare de clasă aproape unitar, definirea noțiunii de **context** și prezentarea unor implementări concrete ;

5.3. Descrierea unor structuri de program și a unor tehnici și metodologii de proiectare care permit realizarea unei **decuplări** a programelor de interfață locală de proces, de structura și volumul informației din proces (§ 6.3) ;

5.4. Descrierea unui **modul BIOS extins** pentru microsisteme de instrumentație sau interfață de proces (§ 6.4) concretizată prin descrierea structurii sistemului, definirea conceptului de **BIOS extins**, clasificarea originală a funcțiilor necesare, **etapizarea proprie** a proiectării modulului, prin descrierea etapelor succesive de rafinare a proiectării, prezentarea *in extenso* a unei propuneri de set de funcții **BIOS extins** și prezentarea unei implementări concrete.

6. În domeniul **generatoarelor de aplicații pentru sisteme de instrumentație** :

6.1. Identificarea problemei și **elaborarea temei** de proiectare pentru un **GA** în domeniul stațiilor sistemelor de telemecanică (§1.7.1,c) ;

6.2. Identificarea și conturarea unei probleme de generare de aplicații în domeniul instrumentației (§ 7.1, 7.2) și definirea unor specificații pentru acest **GA** (§ 7.2) .

6.3. Precizarea unei modalități originale de **prezentare a specificațiilor** pentru un **GA** (Tabelul 7.1);

6.4. Prezentarea a trei generatoare de aplicații realizate pentru rezolvarea problemei conturate mai sus (§ 7.3) ;

6.5. Descrierea unui principiu original de **generare de aplicații** plecând de la un fișier-*șablon* având o structură care permite selectarea fragmentelor de program necesare și

inserarea parametrilor introduşi de utilizator (§ 7.3) ;

7. În domeniul **instrumentelor CASE pentru aplicaţii multitasking şi pentru sisteme de instrumentaţie** :

7.1. Identificarea a două domenii de aplicaţii, pentru care uzual nu sunt disponibile instrumente **CASE** adecvate (Cap. 8) : aplicaţii de conducere de proces multitasking şi aplicaţii de instrumentaţie pentru sisteme mici ;

7.2. Prezentarea **specificaţiilor** referitoare la conţinutul fişierelor folosite şi la interfaţa cu operatorul, pentru un instrument **CASE** pentru prima categorie de aplicaţii de mai sus (§ 8.1), descrierea funcţionării prin exemplificare pe un caz concret şi orientarea funcţionării instrumentului **CASE** spre aplicarea consecventă a principiilor de proiectare pentru reutilizare (folosirea bazelor software, a unor facilităţi de editare avansate etc.) în vederea creşterii însemnate a productivităţii proiectării ;

7.3. Implementarea, în mediul **CASE** de mai sus, a conceptelor specifice claselor de aplicaţii : clasificarea proiectelor în clase de aplicaţii, editarea **DFD** a unui program pleeând de la diagrama programului generic, posibilitatea afişării în permanenţă a **DFD** etc.

7.4. Implementarea, în mediul **CASE** de mai sus, a unor instrumente de evaluare software orientate spre reutilizare, respectiv, a unor măsuri precum cele prezentate în Cap. 4 şi 5 ;

7.5. Realizarea concretă a programului de pct. 7.2 ;

7.6. Prezentarea **specificaţiilor** referitoare la conţinutul fişierelor folosite şi la interfaţa cu operatorul, pentru un instrument **CASE** pentru a doua categorie de aplicaţii de mai sus (§ 8.2), descrierea funcţionării prin exemplificare şi orientarea funcţionării instrumentului **CASE** spre aplicarea consecventă a principiilor de proiectare pentru reutilizare (folosirea bazelor software) :

7.7. Implementarea în mediul **CASE** de mai sus a unui **generator de aplicaţii** care funcţionează pe baza unui principiu **original**, prin **navigare printr-o bază *hypertext*** şi folosirea unui fişier asociat ***.DLL** ;

7.8. Implementarea posibilităţii de supraveghere a traficului de dialog dintre o aplicaţie *master* de pe calculatorul-gazdă şi o aplicaţie dezvoltată cu instrumentului **CASE** de mai sus ;

7.9. Realizarea concretă a programului de la pct. 7.6 ;

7.10. Asigurarea unei **flexibilităţi** deosebite pentru ambele programe prin principiile şi structurile de program şi de fişiere folosite.

9.3. Direcţii de dezvoltare

Domeniul abordat are o dinamică foarte pronunţată, datorită consecinţelor practice pe care le are adoptarea tehnicilor şi instrumentelor software asociate. Lucrarea de faţă lasă mult loc şi multe direcţii deschise unor viitoare dezvoltări. Enumerăm doar câteva dintre acestea:

- extinderea **cadrului de abordare** *fuzzy* în domeniul proiectării pentru reutilizare ; astfel, se pot încerca definiri pentru noi norme care să inducă relaţii de ordine *fuzzy* ; de asemenea, se pot investiga în continuare posibilităţile de descompunere teoretizate de Teorema de descompunere (4.10) ;

- extinderea **setului de reguli de inferenţă** folosite pentru calculul gradului de reutilizare ;

- lărgirea cadrului conceptual pentru **abordarea sistemică** a proiectării pentru reutilizare ;

- introducerea unor noi definiri de **metrici software** şi de indicatori calitativi pentru evaluarea calităţii reutilizării programelor ;

- imaginarea unor noi tehnici de determinare a **programelor semnificative**, care să fie aplicabile în timpul evoluţiei clasei, nu doar *a posteriori* ;

- dezvoltarea **tehnicilor şi structurilor de program** de tipul celor prezentate în Cap. 6 ;

- construirea unor **preprocesoare** pentru generarea automată a structurilor de date prezentate în Cap. 6 ;

- construirea unui mediu integrat care să permită adaptarea modulului **BIOS extins** descris în § 6.4 la diverse cerinţe hardware ;

- definirea unui limbaj de nivel foarte înalt care să permită definirea de aplicaţii de instrumentaţie folosind modulul **BIOS extins** ;

- dezvoltarea de noi **GA** cât mai flexibile ;

- dezvoltarea facilităţilor oferite de cele două instrumente **CASE** prezentate în lucrare ;

- construirea de fişiere folosite de **GA** prezentate (sau de facilităţile de generare de aplicaţii

incluse în instrumentele **CASE** prezentate) pentru mai multe clase de aplicații.

Domeniul este foarte vast şi este de lucru pentru încă 20 de ani, aşa cum a afirmat Freeman în lucrarea [Fre87]. Şi avea dreptate !

BIBLIOGRAFIE

[Abb83] Abbott,R.J. - *Program Design by Informal English Description*
Communications of the ACM, Volume 26, no. 11, nov. 1983

[Ban94] Banker,D.R., Kauffman, J.R., Wright,C., Zweig,D. - *Automating output size and reuse metrics in a repository-based computer-aided software engineering (CASE) environment*
IEEE Trans. on Software Engineering, vol. 20, no. 3, march 1994

[Bar94] Barkakati, N. - *Borland C++ 4 Developer's Guide*
Sams Publishing, Carmel, 1994

[Bas95-F] Bassett,P.G. - *Frame Based Engineering: Cost Effective Systems Through Cost Effective Reuse*
Prentice Hall, 1995

[Bas95-M] Basili,V.R., Briand, L.C. - *Measuring the Impact of Reuse on Quality and Productivity in Object-Oriented Systems*
Communications of ACM, july, 1995

[Big84] Biggerstaff,T.J., Perlis, A.J. - *Foreword (Special issue on software reusability)*
IEEE Trans on Software Engineering, vol. SE-10, no 5, sept. 1984

[Ble94] Blechar,M. - *IEAdvantage: A IE-CASE Tool With an E-CASE Mentality*
Applications Development & Management Strategies, Research Notes, Stamford, USA, oct. 1994

[Boo86] Booch,G. - *Object-Oriented Development*
IEEE Trans. on Software Engineering, vol SE-12, no.2, febr. 1986

[Boo93] Booch,G. - *Object-Oriented Analysis and Design with Applications*
Second Edition, Benjamin/Cummings, 1993

[Boy84] Boyle,J.M., Muralidharan,M.N. - *Program Reusability Through Program Transformation*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept 1984

[BRG91] *** - *Technology Implementation Index*
Business Research Group, Newton, Cahners Publishing Company, USA, june 1991

[Bro90] Browne,J.C., Lee,T., Werth,T. - *Experimental Evaluation of a Reusability-Oriented Parallel Programming Environment*
IEEE Trans. on Software Engineering, vol. 16, no. 2, feb. 1990

[Bur80] Burstall,R.M., Goguen, J.A. - *The semantics of CLEAR, a specification language*
Proc. 1979 Copenhagen Winter school on Abstract Software specification (Lecture Notes in Computer Science, vol. 86), Springer-Verlag, New-York, 1980

[Bur90] Burns, A., Wellings,A. - *Real-time systems and their programming languages*
Addison-Wesley Publishers, Wokingham, 1990

[Cec94] Cecal,L. - *Programare vizuală orientată pe obiect*
PC WORLD, București, apr. 1994

[Che84-L] Cheng,T.T., Lock, E.D., Prywe,N.S. - *Use of Very High Level Languages and Program Generation by Management Professionals*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Che84-R] Cheatham,T.E. - *Reusability Through Program Transformations*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Coa92] Coad,P., Yourdon,E. - *Object Oriented Design*
Prentice Hall, 1992

[CPN94] *** - *Computer Product News*
Reed Elsevier Publication, Brussels, Belgium, march-may 1994

[CPN95] *** - *Computer Product News*
Reed Elsevier Publication, Brussels, Belgium, march-may 1995

[Dah70] Dahl, O., ş.a. - *Simula 67 Common Base Language*
Norwegian Computing Center, Oslo, Norway, 1970

[Dav86] Davidoviciu,A., Bărbat,B. - *Limbaje de programare pentru sisteme în timp real*
Editura tehnică, București, 1986

[Dav94] Davis,M.A., Sigal,R., Weyuker,E.J. - *Computability, Complexity, and Languages*
Academic Press, London, 1994

[Dia88] Diaconescu,S., Bereczky,A., **Tivadar,V.** - *Program pentru aplicații de proces cu panou de operare*
Sesiunea de comunicări a tinerilor absolvenți, IPTV Timișoara, 1987

[DiF93] Di Felice,P. - *Reusability of mathematical software: a contribution*
IEEE Trans. on Software Engineering, vol. 19, no. 8, aug. 1993

[Dod87] Dodescu,Gh., ş a. - *Informatica*
Editura ştiinţifică şi enciclopedică, Bucureşti, 1987

[Dom95] Domanski,P.D., Arabas, J. - *On generating the fuzzy rule base by means of genetic algorithm*
EUFIT Proceedings, vol. 1, Aachen, Germany, aug. 1995

[Dum90] Dumke,R. - *Software Metrics in the Software Design*
Proceedings of the Workshop Computer Aided Software Evaluation, Technical University Magdeburg, oct. 1990

[Dum93-G] Dumke,R., Goetzke, C., Winkler,A. - *Total Quality Management in Object-Oriented Software Development*
Proceedings of the Third International Conference on Software Quality, Lake Tahoe, Nevada, oct. 1993

[Dum93-S] Dumke, R. - *Software-Metriken in der objektorientierten Software-Entwicklung*
Workshop of the German Software Metrics Group, Stuttgart, sept. 1993

[Dum93-X] Dumke,R., Xiros,N. - *Software Measurement in Objectworks Smalltalk*
Study, University of Magdeburg, University of Athen, aug. 1993

[Dum94] Dumke,R., Kuhrau,I. - *Tool-Based Quality Management in Object-Oriented Software Development*
Proceedings of the Third Symposium on Assessment of Quality Software Development Tools, Washington DC, june 1994

[Ebe85] Eberle,L., Tivadar,V. - *O nouă concepţie despre software pentru roboţi industriali de generaţia I*
Sesiunea de comunicări tehnico-ştiinţifice "TEHNIC 2000", Intr. Electrotimiş, vol. IV, pp. 1057-1064, Timişoara, 1985

[Ebe93] Ebert,C. - *An Approach to Fuzzy Data Analysis for Software Quality Control*
EUFIT Proceedings, Aachen, Germany, 1993

[Ele91] Eleş,P., Ciocârlie,H. - *Programarea concurentă în limbaje de nivel înalt*
Editura ştiinţifică, Bucureşti, 1991

[Fen91] Fenton, N.E. - *Software metrics : A rigorous Approach*
Chapman & Hall, London, 1991

[Fil91] Filli,Th. - *Fuzzy - Logik*
Franzis - Verlag Gmbh, München, 1991

[Fow95] Fowler,S.L., Stanwick, V.R. - *The GUI style guide*
Academic Press Proffesional, Boston, 1995

[Fra94] Frakes,W.B. - *Advances in Software Reusability: Proceedings of the Third International Conference on Software Reuse*
November 1994, Rio De Janeiro, Brazil, IEEE Computer Society, 1994

[Fre87] Freeman,P. - *A Conceptual Analysis of the Draco Approach to Constructing Software Systems*
IEEE Trans. on Software Engineering, vol. SE - 13, no. 7, july 1987

[Fre94] Frenţiu,M., Pârv,B. - *Elaborarea programelor*
Editura ProMedia, Cluj, 1994

[Gas81] Gaşpar,D. - *Analiză funcţională*
Editura Facla, Timişoara, 1981

[Gib87] Gibbs,N.E., Failey, R.E. (editors) - *Software Engineering Education*
Springer - Verlag, New - York, 1987

[Goe93] Goetzke,C. - *Analyse der Smalltalk-Methodenkomplexitaet*
Study, University of Magdeburg, july 1993

[Gog78] Goguen,J.A., Ginali,S. - *A Categorical approach to general systems theory*
Applied General Systems Research, G.Klir, Ed. Plenum, New - York, 1978

[Gog84] Goguen,J.A. - *Parametrized Programming*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Gol83] Goldberg,A., Robson,D. - *Smalltalk-80: The Language and its Implementation*
Addison-Wesley, Reading, MA, 1983

[Hoo91] Hooper,J.W., Rowena, O.C. - *Software Reuse: Guidelines and Methods*
(Software Science and Engineering) Plenum Publishing, 1991

[Hor84] Horowitz,E., Munson, J.B. - *An expansive View of Reusable Software*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[ICS92-R] *** - *Software Reuse: Past, Present, and Future*
IEEE Computer Society Press, New-York, 1992

[ICS92-S] *** - *Standard no. 1045/1992 - Standard for Software Productivity Metrics*
IEEE Computer Society Press, New-York, 1992

[IES93] *** - *IE:Advantage. The World's Best Universal CASE Tool*
Information Engineering Systems Corporation, Alexandria, USA, 1993

[Ion82] Ionescu,T. - *Sisteme şi echipamente pentru conducerea proceselor*
Editura Didactică şi Pedagogică, Bucureşti, 1982

[Ion85] Ionescu,V. - *Teoria sistemelor*
Editura Didactică şi Pedagogică, Bucureşti, 1985

[Jac92] Jacobson, I., ş.a. - *Object-Oriented Software Engineering : A Use Case Driven Approach*
Addison-Wesley, Workingham, England, 1992

[Jon84] Jones,T.C. - *Reusability in Programming: A Survey of the State of the Art*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Kal75] Kalman,R.E., Falb,P.L., Arbib,M.A. - *Teoria sistemelor dinamice*
Editura tehnică, Bucureşti, 1975

[Ker84] Kernighan,B.W. - *The UNIX System and Software Reusability*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Kit87] Kitchenham,B. - *Towards a constructive quality model, Part I : software quality modelling, measurement and predictions*
Software Engineering Journal, vol. 22, no. 4, 1987

[Kli88] Klir,G.J., Folger, T.A. - *Fuzzy sets, uncertainty, and information*
Prentice-Hall International, Inc., New-York, 1988

[KMe93] *** - *DATA ACQUISITION. Catalog & Reference Guide*
Vol. 26, Keithley Metrabyte, 1993

[Kru90] Kruzella,I., Brorsson,M. - *Human aspects and organization issues of software reuse*
Preprints of UNICOM Seminar on Software Reuse and Reverse Engineering in Practice, London, dec. 1990

[Kuh93] Kuhrau,I. - *Softwaretechnische Analyse von Borland C*
Study, University of Magdeburg, july 1993

[Lan71] MacLane,S. - *Categories for the Working Mathematician*
Springer-Verlag, New-York, 1971

[Lim95] Lim,C.W. - *Managing Software Reuse*
Prentice Hall, 1995

[Lip88-M] Lipan,L., **Tivadar,V.** - *Mediu de punere la punct şi testare a aplicaţiilor de informatică distribuită pentru conducerea proceselor*
Simpozionul de calculatoare şi conducere automată a proceselor, I.P.T.V. Timişoara, vol. II, pp. 79-82, 1988

[Lip88-S] Lipan, L., **Tivadar, V.** - *Sistem de programe pentru poziționarea cilindrilor laminorului bluming 950 de la C.S. Reșița*
Simpozionul de calculatoare și conducere automată a proceselor, I.P.T.V. Timișoara, vol. II, pp. 65-72, 1988

[Lip94] Lipovan,O. - *Capitole de matematici speciale. Mulțimi fuzzy și elemente de matematică discretă*
Curs litografiat, partea I, Universitatea Tehnică din Timișoara, 1994

[Lis86] Liskov,B., Guttag,J. - *Abstraction and Specification in Program Development*
The MIT Press, Cambridge, MA, 1986

[Lit84] Litvintchouk,S.D., Matsumoto,A.S. - *Design of Ada Systems Yielding Reusable Components: An Approach Using Structured Algebraic Specification*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Mat96] Matei,C. - *Formatul grafic PCX*
PC Report, Computer Press Agora, Tîrgu-Mureș, nr. 45, iunie, 1996

[Mcc93] McClure,S. - *Information Engineering Systems Corp.'s IE:Advantage, Version 6.1. Multi-User Tool Has Intelligent Model Analysis*
CASE Trends, Northboro, USA, dec. 1993

[McG92] McGregor,J.D., David, A.S. - *Object-Oriented Software Development: Engineering Software for Reuse*
(VNR Computer Library) Thomson Computer, 1992

[Man88] Mancu,D., Daday,H. - *Generare de aplicații pe baza unei descrieri de nivel foarte înalt*
Proiect de diplomă, Facultatea de Electrotehnică, I.P.T.V. Timișoara, 1988

[Mat84] Matsumoto,Y. - *Some Experience in Promoting Reusable Software : Presentation in Higher Abstract Levels*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Mat87] Matsumoto,Y. - *Software Reuse in Industrial Process Control*
2-nd International Conference on Software Production, IEEE Tutorial, P. Freeman, editor, 1987

[MET93] *** - *METKIT - Metrics Educational Toolkit*
Information and Software Technology, Volume 35, No. 2, feb. 1993

[Mey88] Meyer,B. - *Object Oriented Software Construction*
Prentice-Hall, Englewood Cliffs, 1988

[Mey94] Meyer,B. - *Reusable Software: The Base Object-Oriented Component Libraries*
(Object-Oriented Series) Prentice Hall, 1994

[Mih95] Mihalache,A. - *Când calculatoarele greşesc. Fiabilitatea sistemelor de programe (software)*
Editura didactică şi pedagogică, Bucureşti,1995

[Mil95] Mili,H., Mili,F., Mili, A. - *Reusing Software: Issues* and Research Directions
IEEE Trans. on Software Engineering, vol. 21, no. 6, june 1995

[Mit87] Mittermeir,R.T., Oppitz,M. - *Software Bases for the Flexible Composition of Application Systems*
IEEE Trans. on Software Engineering, vol. SE-13, no. 4, apr. 1987

[Mun77] Muntean,I. - *Sinteza automatelor finite*
Editura tehnică, Bucureşti, 1977

[NAT92-D] *** - *Development of Reusable Software Components*
NATO Standard, ASSET-528, NATO Communications and Information Systems Agency, Brussels, Belgium, 1992

[NAT92-M] *** - *Management of Reusable Software Components Library*
NATO Standard, ASSET-529, NATO Communications and Information Systems Agency, Brussels, Belgium, 1992

[NAT92-S] *** - *Software Reuse Procedures*
NATO Standard, ASSET-529, NATO Communications and Information Systems Agency, Brussels, Belgium, 1992

[Nei84] Neighbors,J.M. - *The DRACO Approach to Constructing Software from Reusable Components*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Neu91] Neumann,K. - *Konzeption und Prototyploesung von Tools zur Qualitaetsbewertung eines objektorientierten Software-Entwurfs*
Study, Technical University Dresden / Technical University Magdeburg, jan. 1991

[Neu92] Neumann,K. - *Implementation von Software-Metriken im objektorientierten Software-Entwurfssystem SMALLTALK*
Master's Thesis, Technical University Magdeburg, 1992

[NIA93] *** - *Solutions. A Guide to Third-Party Products and Consultants for LabVIEW and LabWindows*
National Instruments Alliance Program, june 1993

[NIC92] *** - *LabVIEW for Windows. Demonstration Guide*
National Instruments Corp., Nov. 1992

[Nie92] Nielsen, K. - *Object-Oriented Design with Ada*
Bantam Books, New York, 1992

[Nie95] Nielsen, K. - *Software Development with C++. Maximizing Reuse with Object Technology*
Academic Press Professional, Cambridge, 1995

[NIn94] *** - *IEEE 488 and VXIbus Control, Data Acquisition, and Analysis*
National Instruments, 1994

[Phi95] *** - *Data Handbook IC20, 80C51-Based 8 Bit Microcontrollers*
PHILIPS, 1995

[Pop81] Popescu,T., ş.a. - *Dicţionar de informatică*
Editura ştiinţifică şi enciclopedică, Bucureşti, 1981

[Pop84] Popescu,N. - *Sisteme informatice cu funcţionare în timp real*
Editura militară, Bucureşti, 1984

[Pop89] Popescu,M, Gyulay,A., Ionică,T., Popescu,S., Munteanţ,M., Kiss,I. - *Echipament de comandă numerică pentru maşini de tăiere termică NUMEROM 633*
Tehnic 2000, nr. 5, Timoşoara, 1989

[Pre96] Prelipceanu,D., Boia,C. - *System of Automatic Monitoring by Means of Calculator of a Well Field Using LabWindows Media*
Proceedings of Automatic Control and Testing Conference - A'96 - TIRETA 10, Cluj-Napoca, mai 1996

[Pri93] Prieto-Diaz,R., Willian, B.F. - *Advances in Software Reuse : Selected Papers from the Second International Workshop on Software Reusability*
IEEE Computer Society, march, 1993

[Rac94] Racko, R. - *Inside inheritance*
Software Development, may, 1994

[Rac96] Rácz, S. - *Instrument CASE pentru programe de instrumentaţie*
Lucrare de licenţă, Univ. "Politehnica" Timişoara, Departamentul de Automatică şi Informatică Tehnică, Timişoara, 1996

[Rad79] Radu,O., Săndulescu,Gh. - *Filtre numerice. Aplicaţii*
Editura tehnică, Bucureşti, 1979

[Ram88] Ramamurthy,B., Melton,A. - *A Synthesis of Software Science Measures and the Cyclomatic Number*
IEEE Trans. on Software Engineering, vol. 14, no. 8, aug. 1988

[Rob94] Robu,N. - *Mecanismele programării concurente în timp real*
Editura "*Helicon*", Timişoara, 1994

[Rum91] Rumbaugh,J., Blaha,M., Premerlani,W., Eddy,F., Lorensen,W. - *Object-Oriented Modelling and Design*
Prentice-Hall International, Inc., 1991

[Rus85] Rusu,R., Cucu,L., Fantaziu,R., Eberle, I., **Tivadar, V.** - *Sistem de programe pentru comanda roboţilor industriali de tip REMT*
Sesiunea de comunicări ştiinţifice "Electromotor", vol.I, partea a II-a, Timişoara, 1985

[Sav96] Savii,G.G. - *Proiectarea asistată de calculator*
Editura "*Helicon*", Timişoara, 1997

[Sch94] Schafer,W., et al. - *Software Reusability*
(Ellis Horwood Workshop) Ellis Horwood, 1994

[She95] Shepperd, M. - *Foundations of Software Measurement*
Prentice-Hall, Englewood Cliffs, 1995

[Sil88] Silea,I., Balazs,H., Eberle,I., Bereczky,F., Bereczky,A., Popa,Gh., Socol,S. - *ESAROM - Echipament de supraveghere a proceselor de aşchiere*
TEHNIC 2000, Timişara, sept. 1988

[Soc89] Socol,S., Stoicu,I., Popa,Gh., **Tivadar,V.** - *Echipament de comandă numerică pentru poziţionarea cilindrilor unui laminor blooming*
TEHNIC 2000, nr. 5, Timişoara, 1989

[Sol84] Soloway,E., Ehrlich,K. - *Empirical Studies of Programming Knowledge*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Som93] Somnea,D., Turturea,D. - *Iniţiere în C++. Programarea orientată pe obiecte*
Editura Tehnică, Bucureşti, 1993

[Spi95] Spireu,C., Lopătan,I. - *POO. Analiza, proiectarea şi programarea orientate spre obiecte*
Editura *Teora*, Bucureşti,1995

[Sta88] Stancovic,J.A. - *Real-time computing systems : the next generation*
Technical Report, University of Massachusetts, 1988

[Sta96] Stanciu, E. - *Reutilizarea software : cum se apreciază*
PC World, nr. 1-2, Bucureşti, ian/feb. 1996

[Sta84] Standish,T.A. - *An Essay on Software Reuse*
IEEE Trans. on Software Engineering, vol. SE-10, no. 5, sept. 1984

[Sto93] Stoicu-Tivadar, L., **Stoicu-Tivadar, V.** - *Some aspects in position control at a class of non-linear systems*
Buletinul ştiinţific şi tehnic al Universităţii Tehnice Timişoara, tom 38(52), 1993

[Sto94-A] **Stoicu-Tivadar,V.** - *Asupra abordării sistemice a unor probleme de inginerie a programării pentru o clasă de aplicaţii de conducere de proces*
Referatul nr. 2 pt. doctorat, Catedra de Automatică şi Informatică Industrială, Universitatea Tehnică Timişoara, 1994

[Sto94-D] **Stoicu-Tivadar,V.**, Dragomir,T.L. - *On evaluation of reuse rate for process control software*
Proceedings of Conference on Computer Science and Technical Informatics - CONTI '94, Timişoara, 1994

[Sto94-H] **Stoicu-Tivadar,V.**, Haag,Zs. - *Instrumentation, and process surveillance program generators*
Buletinul ştiinţific UTT 1994, Tom 39(53), Seria Automatizări şi Calculatoare, Timişoara, 1995

[Sto94-R] **Stoicu-Tivadar,V.** - *Reuse rate increase program structures in process control small applications*
Proceedings of Conference on Computer Science and Technical Informatics - CONTI '94, Timişoara, 1994

[Sto94-S] **Stoicu-Tivadar,V.**, Stoicu-Tivadar,L. - *About several software measures useful in reusable software engineering for process control*
Buletinul ştiinţific UTT 1994, Tom 39(53), Seria Automatizări şi Calculatoare, Timişoara, 1995

[Sto95-A] **Stoicu-Tivadar,V.** - *An attempt of formalization in software reusability*
XXXIX Konferencija ETRAN - a, Zlatibor, Yugoslavia, 1995

[Sto95-H] **Stoicu-Tivadar,V.**, Haag,Zs. - *A program generator for instrumentation systems and process monitoring, using 8051 family microcontrollers*
Proceedings of CSCS '10, Bucureşti,1995

[Sto95-O] **Stoicu-Tivadar,V.** - *Asupra apartenenţei programelor la clase de aplicaţii*
Analele Universităţii din Oradea, 1995

[Sto95-S] **Stoicu-Tivadar,V.** - *O abordare sistemică a proiectării pentru reutilizare software*
Acta Universitatis Cibiniensis, vol. XXII, Seria Tehnică, Sibiu, 1995

[Sto96-N] **Stoicu-Tivadar,V.** Neagoe,C. - *A CASE tool for local process control and instrumentation applications* Proceedings of Automatic Control and Testing Conference - A'96 - THETA 10, Cluj-Napoca, mai 1996

[Sto96-S] **Stoicu-Tivadar,V.,** Stoicu-Tivadar,L., Rácz, S. - *INSTRUMENT DESIGNER - A Software Tool for Instrumentation Applications Development* Proceedings of The International Conference of Technical Informatics - CONTI'96, Timişoara, 1996

[Str91] Stroustrup,B. - *The C++ Programming Language* Second Edition, Addison-Wesley, Reading, MA, 1991

[Ter94] Tervonen,I. - *Quality-driven assessment : a pre-review method for object-oriented software development* Department of Information Processing Science, University of Oulu, Research Papers, Series A19, Oulu, Finland, 1994

[Tiv85] **Tivadar, V.** - *O abordare sistematică a problemei algoritmilor de poziționare la roboți industriali* Sesiunea de comunicări tehnico-ştiinţifice "TEHNIC 2000", intr. Electrotimiş, vol. II, pp. 342-351, Timişoara, 1985

[Tiv87-B] **Tivadar,V.**, Botoca,S., Lipan,L., Bereczky,A., Diaconescu,S., Manciu,M., Neamţu,D. - *Cîteva posibilităţi de restructurare la generare şi de reconfigurare dinamică pentru aplicaţii informatice de proces* Al IV-lea Colocviu de sisteme-modele-informatică-cibernetică. Bucureşti, 1987

[Tiv87-N] **Tivadar,V.**, Neagu,V., Botoca,S., Lipan,L., Manciu,M., Neamţu,D. - *Sistem informatic de proces pentru dispecerizarea unui front de captare a apei* Al IV-lea Colocviu de sisteme-modele-informatică-cibernetică. Bucureşti, 1987

[Tra88] Tracz,W. - *Software reuse maxims* A.C.M. SIGSOFT, 13, 4, October 1988

[Uni94] *** - *UnixWorld's Open Computing* febr. 1994

[Ung96] Ungureanu,G., Gal-Bancsi,T. - *Programmable Automaton Made with Microcontrollers for Industrial Applications - Automatic Burner Controls* Proceedings of Automatic Control and Testing Conference - A'96 - THETA 10, Cluj-Napoca, mai 1996

[Vas85] Vasilescu,P. - *Discrepanţa generaţiilor în informatică* Editura ştiinţifică şi enciclopedică, Bucureşti, 1985

[Wal93] Walton,P, Neil,M. - *Integrated Software Reuse: Management and Techniques* (Unicom Applied Information Technology Series) Ashgate Publishing Company, 1993

[Wey90] Weyuker,E.J. - *The Cost of Data Flow Testing : An Empirical Study* IEEE Trans. on Software Engineering, vol. 16, nr. 2, feb. 1990

[Wie95] Wieringa,R.B., Feenstra,R.J. - *Information Systems: Correctness and Reusability* Selected Papers from the IS-CORE Workshop, Amsterdam, Sept. 1994, World Scientific, 1995

[You79] Yourdon,E., Constantine,L.L. - *Structured design* Prentice-Hall, Englewood Cliffs, 1979

[Yua95] Yuan,Y., Zhuang, H. - *Using a genetic algorithm to generate fuzzy classification rules* EUFIT Proceedings, vol. 1, Aachen, Germany, aug. 1995

[Zad73] Zadeh,L.A., Polak,E. - *Teoria sistemelor* Editura tehnică, Bucureşti, 1973

ANEXE

ANEXA 1.1

Câteva puncte de vedere referitoare la reutilizare (§ 1.6)

1) După Horowitz [Hor84], abordările reutilizării software pot fi stratificate astfel :

- medii de dezvoltare care să favorizeze reutilizarea (**QuickC**, **Visual C++**, aplicații de tip spread-sheet, programare vizuală,etc.) ;
-facilități hardware care permit accesul la resursele reutilizabile (s.ex. lucrul la rețele de calculatoare) ;
- limbaje de programare cu facilități de reutilizare (s.ex. **C++**) ;
- **prototipizarea** : se referă la definirea unui prototip, din care, prin interacțiunea cu clientul pentru a rafina cerințele, se va obține produsul final prin transformări succesive ;
- reutilizarea unui program existent : dacă doar puține modificări sunt necesare, este mult mai ieftină modificarea programului existent decât dezvoltarea unui sistem complet nou ;
- utilizarea codului reutilizabil ;
- practica proiectării reutilizabile, pe baza unui set de concepte și termeni care sunt utilizați în analiza domeniului ;
- **generatoare de aplicații** destinate ajutării utilizatorilor finali pentru a construi aplicații într-un domeniu dat ;
- specificarea formală și sistemele de transformare.

În articolul citat, se subliniază unele aspecte și dificultăți corespunzătoare diverselor fațete ale reutilizării. Două dintre acestea sunt :

- Reutilizarea codului presupune evidențierea dificultăților legate de această abordare, precum și trecerea în revistă a mecanismelor necesare pentru aplicarea sistematică a acesteia. Sunt amintite aici 3 dificultăți majore :

  - este imposibil de creat un sistem parametrizat eficient, sigur și convenabil pentru o clasă largă de sisteme ;
  - nu întotdeauna pot fi ocolite proprietățile mașinii (fie hardware, fie virtuale) ;
  - apar dificultăți referitoare la catalogare și definirea descriptorilor care să permită unui utilizator regăsirea modulului potrivit.

Relativ la aceste dificultăţi, **problemele de rezolvat** sunt următoarele :

- de identificare a componentelor, adică, răspunsul la întrebarea : cum se pot determina acele componente general utile care să fie folosite în proiecte diferite ; de obicei, răspunsul restrânge domeniul de aplicabilitate al reutilizării unor module specificate, la o categorie mai îngustă decât se doreşte ;
-de specificare a componentelor ; aceasta se referă la modalitatea de a descrie componentele astfel încât să fie cât mai perceptibile ;
- de aspect formal al componentelor, respectiv de limbajul de programare folosit;
- de catalogare a componentelor ; se poate ivi chiar necesitatea catalogării pe categorii pe baza unui metalimbaj.

Dificultăţile reutilizării determină lărgirea ariei de aplicare nu numai asupra reutilizării de cod ci şi asupra altor etape din ciclul de viaţă al programelor (v. Tab. A1-1). Astfel,

Tabelul A1-1. Creşterea potenţială a productivităţii prin reutilizare / ciclu de viaţă software

	Costuri curente (% din total)	Îmbunătăţire %	Cost net
Cerinţele sistemului	2	0	2
Cerinţele hardware	8	25	6
Cerinţele software	10	20	8
Proiectare software	12	40	7
Codificare	13	75	3
Testare module	24	50	12
Testare de integrare	13	30	9
Documentaţie	6	30	4
Testare sistem	12	25	9
Total	100	40	60

reutilizarea devine cu adevărat rentabilă, ducând la reducerea costurilor cu până la 40 %. La nivelul reutilizării codului mai există o dificultatea majoră, anume necesitatea calificării speciale a proiectanţilor, în vederea realizării modificărilor de cod în modulele (atomii) selectate.

- Sisteme producătoare de program de nivel înalt.

Acestea procesează un limbaj adecvat domeniului abordat sau chiar un limbaj mai general, care să ofere o descriere a aplicaţiei. Schema generală a unui asemenea sistem este dată în fig. A1-1 :

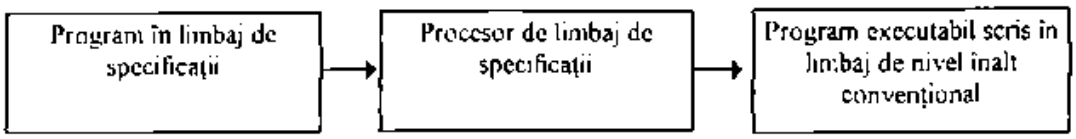


Fig. A1-1. Schema generală a unui sistem producător de program de nivel înalt

Unele dificultăţi trebuie surmontate şi de asemenea trebuie făcute unele opţiuni atunci când se defineşte un astfel de sistem :

- dacă se face prelucrare secvenţială de tip "batch" pentru specificaţiile introduse în sistem sau dacă acestea se nasc interactiv ;
- dacă se pot detecta incompletitudinea şi/sau inconsistenţa specificaţiilor şi dacă se poate garanta că s-a realizat în acest sens o analiză şi o detecţie completă ;
- cât de eficient este procesorul de specificaţii ?
- cât de eficient este codul rezultat ?

Un exemplu de astfel de sistem este **DRACO** [Nei84][Fre87], prezentat pe larg mai departe.

2) Jones prezintă, în [Jon84], câteva tendinţe de aplicare a tehnicilor de reutilizare (referitoare la structuri de date, arhitecturi de programe, reutilizare în proiectare, module şi subrutine refolosibile, etc). creşterea estimată pentru 1990 (la nivelul anului 1984), pentru reutilizarea software este de 50 % părţi refolosite din volumul total al programelor comerciale. În articolul citat sunt date câteva cifre statistice interesante . astfel, 75 % din funcţiile din aplicaţiile bancare şi de asigurare sunt folosite în mai multe aplicaţii. Alte surse indică o estimare de doar 30 % din cod dedicat specificului problemei, în majoritatea aplicaţiilor. Dacă pentru 1983 erau inventariate resursele umane şi rezultatele muncii lor, la nivelul de 3.250.000 programatori pe plan mondial, care au produs un număr de 6.500.000.000 linii-sursă, la nivelul anului 1990 erau estimaţi 7.695.000 programatori, care să producă 15.292.000.000 linii-sursă. Tendinţa în orizontul anului 2000 este de reducere la 10-15 % a ponderii părţii specifice unei aplicaţii. Într-adevăr, folosirea largă a tehnicilor **POO** face ca această previziune să se împlinească, iar efortul de proiectare să fie împins din ce în ce mai mult spre utilizatorul final. Cumulativ la perioada 1955-1990, se presupune că s-au produs 100 miliarde linii-sursă, dar din acestea, doar 25 miliarde sunt inventariate. Este evident că o refolosire relativă la acest volum uriaş de programe (şi la nivelul de creştere în perspectivă) are un impact economic major.

În articolul analizat se enumerau câteva domenii în care autorul excludea folosirea tehnicilor de reutilizare : sistemele în timp real (deci cu restricţii de timp ţinând de specificul aplicaţiei), sistemele de fabricaţie (care sunt extrem de "personalizate"), orice alte sisteme cu restricţii importante referitoare la performanţe şi memorie ocupată. Se presupunea că aceste

restricții constituie doar o problemă tranzitorie, ceea ce s-a adeverit, în prezent reutilizarea software întinzându-se și asupra acestor domenii de aplicații.

3) În articolul [Sta84] sunt de asemenea prezentate cifre și previziuni legate de evoluția domeniului analizat. Analiza tendințelor de pe piața produselor software arată că prețul acestora tinde să crească exponențial. De aceea reutilizarea software se dovedește cu atât mai importantă. Tendințele de creștere a activității de proiectare software, la data elaborării articolului, arătau că la nivelul anului 1990 era așteptată o creștere a numărului de proiectanți software la nivelul de 1,2 milioane, dacă nu se iau măsuri speciale pentru creșterea gradului de reutilizare al programelor. Autorul considera că următoarele aspecte trebuie analizate pentru a obține noi tehnologii de reutilizare software :

a) Regăsirea informației : trebuie găsite noi modalități de organizare, indexare, descriere și referire efectivă a componentelor software reutilizabile.

b) Generatoarele software : aici trebuie căutată o cale sistematică de generare a instanțelor concrete ale modulelor software abstracte prin substituirea unor parametri specifici.

c) Modele de compunere a modulelor : aici e vorba de a da răspuns la întrebarea referitoare la metodele prin care putem construi sisteme prin compunerea unor module. Sunt amintite aici tehnici de apelare încuibată, conducte UNIX ("*pipes*"), interfețe partajate, "*ascunderea*" informației, etc.

d) Înțelegerea programelor : înainte de a refolosi anumite feluri de software (prin modificare sau îmbunătățire) trebuie să înțelegem cum funcționează. În cadrul ciclului de viață al programelor, înțelegerea poate să ocupe un volum însemnat. Spre exemplu, dacă în cadrul unui ciclu de viață "clasic", deja costul întreținerii este de 70-90 % din prețul total, în cadrul acestei etape, costul înțelegerii programelor în sine poate să urce la 50-90 % din cel al întreținerii. Adică, dacă din start ne bazăm pe reutilizare de software, ne putem imagina că în economia proiectării, partea rezervată înțelegerii programului va ocupa un loc însemnat. Rezultă implicit importanța scrierii programelor astfel încât să poată fi cât mai ușor înțelese.

e) Analiza beneficiilor : trebuie să ne punem problema avantajelor pe care le așteptăm în urma aplicării diverselor tehnici de reutilizare software, precum reutilizare directă a unor module concrete, reutilizare prin "rafinare" (îmbunătățire) și reutilizare prin modificări. În ce măsură într-un sistem nou se regăsesc sintetizate diversele forme de reutilizare ? Sau, pentru diverse familii de sisteme, cât anume din programele componente se poate refolosi eventual, în continuare ? La aceste întrebări se poate răspunde cel mai bine prin analiza unor situații practice. Autorul se așteaptă la beneficii în special la reutilizarea în domenii mai înguste, unde este o cerere de versiuni asemănătoare, adaptate la cerințele unei categorii largi de clienți.

4) În [Lan84] se insistă pe importanța **standardizării** în vederea reutilizării software. Este adevărat că articolul se referă în special la aplicații comerciale dar concluziile sale sunt perfect utilizabile în orice alt domeniu de aplicații. Astfel, autorii subliniază că standardizând funcțiile în aplicațiile contabil-comerciale, sub forma unor structuri logice și a unor module funcționale reutilizabile, productivitatea dezvoltării de noi aplicații a crescut cu 50 %, permițând programatorilor reorientarea activității lor spre rezolvarea problemelor care necesită creativitate, mai degrabă decât activitatea redundantă. Astfel, organizațiile pot să-și reorienteze 60 - 80 % din resurse spre dezvoltare de noi aplicații. Concluzia care se desprinde de aici (și de altfel și din alte articole citate în acest paragraf) este că unul din principalele obiective ale reutilizării software este corecta întrebuințare a resurselor umane, așa de greu de găsit în domeniul ingineriei software performante, adică folosirea inginerilor software în adevărata muncă de creație, pentru dezvoltarea părților cu adevărat noi ale aplicațiilor.

5) Articolul [Mat84] este o privire mai generală asupra domeniului reutilizării software, prin aceea că aici autorul definește mai multe nivele de abstractizare pentru a asigura un mai mare grad de reutilizare. În acest sens, procesul de proiectare este stratificat pe acele nivele. Experimentele de reutilizare pot fi deci clasificate în funcție de aceste nivele, astfel:

a) **programe descrise prin limbaje de nivel foarte înalt**. În acest caz, programul-țintă este generat din descriere prin analiza acesteia și extragerea unor module de cod dintr-un "*rezervor*", adaptându-le apoi problemei abordate.

b) **structuri logice** - precum diagramele, schemele logice ori de transformare, abstractizate din coduri existente - ce urmează să fie **adaptate** propriilor necesități, prin modificarea codului-sursă existent.

c) **probleme descrise în limbaje de nivel foarte înalt** ; din aceste descrieri, utilizatorul obține codul-sursă prin aplicarea de demonstrări de teoreme și de procese de inferență.

În articolul citat, prin **module software** se înțeleg mulțimi de stări cuprinzând atât codul-sursă cât și cerințele și specificațiile de proiectare și programare. Nivelele de abstractizare folosite primar sunt cele corespunzăare cerințelor, proiectării și programului. Aici este introdus conceptul de **prezentare**, care este definit drept o cerință a programului, conectată cu specificarea domeniului de variație (măsura în care programul poate fi schimbat relativ la această cerință) când programul este refolosit în alte aplicații. Deoarece o prezentare descrie un program existent la un nivel înalt de abstractizare, la nivelul cerințelor, rezultă două avantaje ale folosirii acesteia în practica proiectării :

- claritate asupra comportării programului
- maximizarea creşterii productivităţii

Într-adevăr, proiectantul care dezvoltă un program, caută o prezentare care să se potrivească cerinţelor acestui program. Dacă prezentarea unui alt program se potriveşte, atunci acesta poate fi adaptat cerinţelor programului dorit şi deci poate fi refolosit.

Modulele software reutilizabile trebuie să aibă următoarele caracteristici majore :

a) **generalitatea** - este gradul în care utilizatorii unor module software (care nu au participat la proiectarea acelor module) pot înţelege obiectivele acelor module, precum şi relaţiile între acestea şi algoritmii folosiţi

b) **determinarea** - este legată de claritatea scopurilor modulelor, performanţelor, constrângerilor, interfeţelor şi resurselor cerute. De aceea, modulele trebuie totdeauna însoţite de descrierea (sau precizarea) :

i) modulelor apelate sau strict necesare executării modulului respectiv
ii) limbajului, sistemului de operare, a utilităţilor, dispozitivelor folosite
iii) întreruperilor care afectează modulele
iv) memoriei necesare.

c) **portabilitatea** - este gradul de simplitate al transferării pe diverse tipuri de calculatoare.

d) **recuperabilitatea** - măsoară gradul în care modulul poate fi uşor manevrat (selectat, memorat, întreţinut şi adaptat) de utilizatori care nu au cunoştinţe aprioriee despre existenta acestuia.

Pentru a putea prezenta teoria abordării priectării pentru reutilizare în viziunea autorului articolului citat, trebuie să facem recurs la un model al procesului de proiectare prezentat în [Dij72]. Proiectarea software este un proces iterativ de rafinare prin care cerinţele specificate în domeniul problemei sunt transformate gradat în programe apte a fi rulate pe un calculator-ţintă. Transformările nu se pot realiza într-un singur pas. În modelul propus în [Dij72], ***procesul proiectării*** este descris astfel :

O maşină **M(i)** şi un program **P(i)** aflate pe nivelul abstract **i** (astfel încât satisfac condiţia că execuţia **P(i)** pe maşina **M(i)** satisface scopul programului **P** care trebuie executat pe maşina-ţintă (reală) **M**), suferă prin transformare spre un nivel mai jos **i+1**, o modificare în **P(i+1)** care poate fi executat pe maşina **M(i+1)**. Dacă nivelul curent este cel mai de jos nivel, **i+1=L**, atunci **M(L)** este maşina reală O fază a procesului de rafinare este un pas care transformă **P(i)** în **P(i+1)**.

Există trei modele de ciclu de viață al programelor, care sunt dinamice, decurgând în coordonata timp, prin trecerea procesului proiectării de la o fază la alta. Cele trei modele sunt :

- modelul "**cascadă**" (modelul "clasic") care presupune respectarea unei secvențialități stricte; modelul este adesea utilizat în proiectarea aplicațiilor industriale; câteodată în procesul de dezvoltare a aplicațiilor se refolosesc experiența dobândită precum și unele module de program, într-un mod nesecvențial ceea ce determină o ieșire a proiectării din limitele acestui model;

-modelul "**omiterii**" , care presupune faze definite într-o ordine a utilizatorului, mai degrabă decât în secvență fixă; astfel, fazele care nu sunt necesare sunt omise;

- modelul "**unificat**" care presupune că :

  - toate fazele sunt definite independent
  - fiecare fază are interfețe bine definite cu alte faze pentru a asigura interconectarea fazelor care au interfețe compatibile ;
  - fazele selectate pot fi interconectate între ele de către proiectant, arbitrar, secvențial sau în altă ordine.

Așadar, fiind dat un proces de proiectare software peste N nivele abstracte, un model secvențial tipic poate fi reprezentat astfel :

$$\mathbf{Form\ (\ i\ ) := Trans(\ i, Form(i\text{-}1)\ )} \qquad (A1.1)$$

pentru i = 1,2,...,N

unde:

i -numărul nivelului abstract

Form (i) - model descris în limbajul "i" de specificații : reprezentare formală a nivelului "i" de abstractizare

Form (0) - cerințele clientului (condițiile inițiale)

Form (N) - codul-sursă al programului

Trans (i,Form(i-1)) - transformarea în pasul "i" sau procesul adaptării (trecerii) fiecărui obiect de pe nivelul "i-1" în obiect de pe nivelul "i"

De obicei procesul proiectării este divizat în 4 transformări (N=4), respectiv 4 nivele abstracte:

- nivelul cerințelor
- nivelul proiectării
- nivelul programului
- nivelul codului-sursă

Pentru a permite comunicarea bidirecțională între diferite niveluri, fiecare formă de specificare de pe diverse nivele trebuie să asigure "**trasabilitatea**", adică acea proprietate din care decurg următoarele : dacă un modul software în Form(i) va fi refolosit, proiectantul va trebui să poată refolosi modulul și în forma Form(j) de-a lungul drumului în ciclul de viață al programelor.

Formalizarea prezentată mai sus permite mai ușoara urmărire a abordării sistematice, în viziunea autorului articolului [Mat84], a conținutului celor 4 etape de transformare enumerate mai sus, care constituie procesul de proiectare :

5,a) <u>Nivelul cerințelor</u>

La acest nivel, obiectul transformării este exterior programului care trebuie dezvoltat. Acesta trebuie pus în conexiune cu obiecte interne acestui program. O descriere a cerințelor, **Form(1)**, conține 6 entități majore :

a) obiecte
b) relații între obiecte
c) luarea deciziilor
d) transformări intrare ieșire
e) constrângeri
f) facilități date

Un exemplu complet de descriere a cerințelor, conform celor de mai sus, pentru un domeniu abordat de autor în [Lip88-S][Soc89][Sto93], este prezentat în continuare [Mat84] (v. fig. A1-2) :

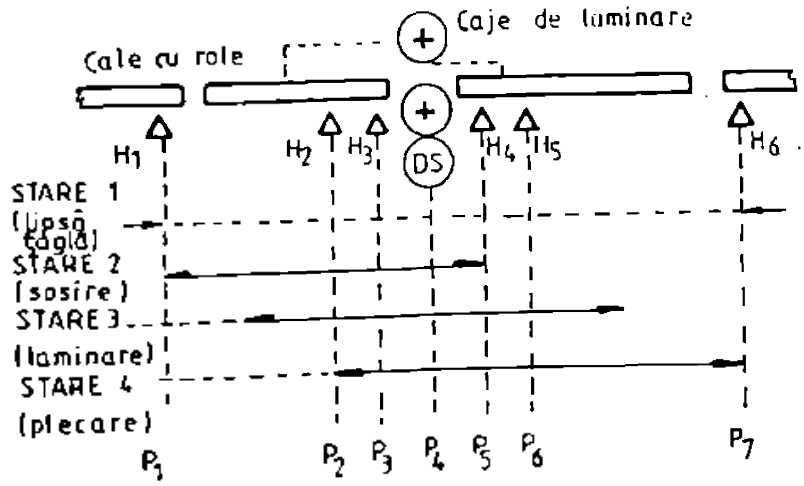


Fig. A1-2. Laminorul din exemplul tratat

Se consideră un laminor (plecând de la o aplicație concretă) având un ciclu de funcționare cu mai multe treceri ale semifabricatului între cajele de laminare. Programul care va fi elaborat va trebui să controleze viteza semifabricatelor și a cajelor. În fig. A1-2, cu **H** sunt marcați detectori termici, iar cu **DS** este desemnat un senzor de sarcină.

Descrirea cerinţelor este făcută într-un limbaj "ca şi **ADA**", pentru uşurinţa unificării notaţiilor în diversele etape ale proiectării. Descrierea este compusă dintr-o specificaţie, sau interfaţă, şi un corp al descrierii. Cerinţa este scrisă într-un stil încapsulat, pentru a simplifica eventuala reutilizare a fiecărui modul de cerinţe şi corespondenţele cu modulele de pe alte nivele. Relaţiile cu alte module de cerinţe şi cu obiectele externe sunt definite în partea de interfaţă.

Descrierea la acest nivel, pentru aplicaţia prezentată, este dată în continuare :

Exemplu de descriere la nivelul cerinţelor

```
01 descrierea cerintelor pentru CONTROL-VITEZA-LAMINOR
02 with
03 object:TAGLA:EXTERNAL_ENTITY;
      --O bucata de otel pentru a fi laminata de caje.
04 object:CAJE:EXTERNAL_ENTITY;
      --O pereche de role care se invirt unidirectional.
05 object:ROLE1, ROLE2, ROLE3, ROLE4:EXTERNAL_ENTITY;
      --Cai cu role care transporta taglele. Acestea sint
      --localizate de ambele parti ale cajelor,
      --numerotate in ordinea 1..4 de la stinga la dreapta.
      --ROLE1 si ROLE2 sint in stinga.
06 object:P1, P2, P3, P4, P5, P6, P7:EXTERNAL_ENTITY;
      --Puncte de control (pozitii) succesive.
07 object:CONTROLLER_INTRARE_PROCES:INPUT_INTERFACE;
08 object:CONTROLLER_VITEZA_LAMINOR:OUTPUT_INTERFACE;
09 object:GESTIUNE_TIMP_REAL:INPUT_INTERFACE;
10 end with;
11 requirement CONTROLLER_VITEZA_LAMINOR is
12 interface relationship is
13 trigerred by GESTIUNE_TIMP_REAL
      with INTERVAL_TIMP_REAL:event;
14 use CONTROLLER_INTRARE_PROCES
      to PRELEVEAZA(VALOARE_SENZOR:VALOARE_LAMINOR);
15    type DATA_LAMINOR is
        record
        H1, H2, H3:analog;
        DS:digital;
        H4, H5, H6:analog;
        STP:digital;
        end record;
      end type;
16 acknowledge CONTROLLER_VITEZA_LAMINOR
      with GATA:event;
17 used by CONTROLLER_VITEZA_LAMINOR which
      OBTINE(VITEZA_LAMINOR:VALOARE_VITEZA);
        type VALOARE_VITEZA is
        (REDUSA, MICA, MARE);
```

```
        end type;
18 end interface relationship;
19 requirement body is
20 object:OBTINE(VITEZA_LAMINOR:VALOARE_VITEZA);
      PROCES is
21 object:H1, H2, H3, H4, H5, H6:DATA;-- termocuple,
      --H1 este in P1, H2 in P2,
      --H3 este in P3, H4 in P5,
      --H5 este in P6, H6 in P7.
22 object:DS:DATA;--senzor de sarcina
23 object:STP:DATA;--buton de stop
24 object:STARE:INTERNAL_ENTITY;
      --In STARE1 nu exista tagla intre P1 si P7.
      --In STARE2 exista o tagla intre P1 si P5.
      --In STARE3 exista o tagla in regiunea care satisface
      -- conditia ca un capat al taglei este dupa P5 iar
      -- celalalt capat este inainte de P2.
      --In STARE4 exista o tagla in regiunea care satisface
      -- conditia ca un capat este inainte de P7 iar celalalt
      -- dupa P2.
25 begin
    estimate STARE of the TAGLA using H1..H6,DS and STP;
    if TAGLA is in STARE1 then
     VITEZA_LAMINOR:=REDUSA;
    elseif TAGLA is in STARE2 or STARE4 then
     VITEZA_LAMINOR:=MICA;
    elseif TAGLA is in STARE3 then
     VITEZA_LAMINOR:=MARE;
    end if;
    acknowledge CONTROLLER_VITEZA_LAMINOR
     with GATA;
  end;
26 end requirement body;
27 end requirement CONTROL_VITEZA_LAMINOR;
28 end description;
```

Descrierea aceasta reprezintă doar o parte din specificații. Acestea mai trebuie să cuprindă:

- drumul (urma) obiectelor în nivelul de abstractizare următor
- constrângerile
- caracteristicile statice, cinetice şi dinamice ale obiectelor exterioare
- resursele date pentru implementare

5,b) Nivelul proiectării

Trans(2) reprezintă tranziția de la cerințele **Form(1)** la specificațiile de proiectare ale **Form(2)**. La această fază se definesc structurile de date, funcțiile, fluxurile de date, structurile de control.

În această fază, spre exemplu, la cerința "CONTROL_VITEZA_LAMINOR" corespund 7 module distincte: DRIVER_INTRARE_PROCES, HANDLER_INTRARI, SELECTARE_VITEZE, CONTROL_VITEZA, HANDLER_ERORI, HANDLER_ALARME și DRIVER_IESIRE_PROCES. În articolul analizat este prezentat in extenso unul din module. La acest nivel folosim diagrame de tranziție între stări și tabele de configurații de intrare (v. art. cit.).

5,c) Nivelul programului

Trans(3) determină de fapt proiectarea structurilor externe ale modulelor de program. Sunt aici proiectate diagrame de flux de date, structuri de date, diagrame de control (scheme logice), astfel definindu-se configurația programului, structurile de fișiere și interfețe. Ca urmare a derulării tranzițiilor acestui nivel, rezultă următoarele :

- Este proiectată structura generală a programului :
  - taskurile sunt determinate (delimitate) pentru programare concurentă ;
  - se determină subprogramele (funcțiile) și pachetele (fișierele în sensul modulelor compilate împreună) folosite ;
  - se definesc structurile fișierelor ;
- Se scriu specificațiile pentru pachete ;
- Sunt proiectate structurile interioare ale pachetelor și structurile de date.

Pentru exemplul urmărit, modulele HANDLER_INTRARI, SELECTARE_VITEZE și CONTROL_VITEZA sunt transformate în taskuri, iar celelalte, implementate prin funcții. În articolul citat, este prezentată detalierea specificațiilor acestei etape, în limbajul **ADA**.

Subiectul de interes major pentru lucrarea de față, reutilizarea software, este, desigur, tratat în articolul analizat prin aceea că este în amănunt descrisă modalitatea de conversie a modulelor de program concepute după metodologia prezentată, în module reutilizabile. Acestea sunt rescrise într-o formă mai generală, în care numele de entități și relații asociate sunt înlocuite de denumiri de o natură mai generală. Astfel, modulul CONTROL_VITEZA_LAMINOR devine CONTROL_DE_STARE, iar fiecare variabilă care poate suferi schimbări este evidențiată prin delimitarea cu **&(...)&**. Un fragment de astfel de program reutilizabil este prezentat în continuare :

Exemplu de fragment de program reutilizabil

```
'Form(1,CONTROL_DE_STARE) is
with
  object:SPATIU:EXTERNAL_ENTITY;
  object:ZONA:EXTERNAL_ENTITY;
      --SPATIU este divizat in ZONA1, ZONA2, ZONA3 si ZONA4
  object:INTRARE_MASURATA:EXTERNAL_ENTITY;
      --M1, M2, M3, M5, M6, M7:analog;
      --M4, M8:digital;
      --(in ZONA1) => (M1 < valoare_de_prag, M2 <...);
      --(in ZONA2) => (M1 > valoare_de_prag,...);
      --(in ZONA3) => (M1 ...);
      --(in ZONA4) => (M1 ...);
  object:OBIECT_MOBIL:EXTERNAL_ENTITY;
      --se misca de la ZONA1 la ZONA2, ZONA3, ZONA4.
  object:CONTROL_INTRARI_PROCES:
      INTREFATA_INTRARE;
      --citeste M1, M2, M3, M4, M5, M6, M7, M8.
  object:GESTIUNE_TIMP_REAL:INTERFATA_INTRARE;
  object:VALOARE_IESIRE:EXTERNAL_ENTITY:stepped_analog;
  object:CONTROL_IESIRE:INTERFATA_IESIRE;
      --genereaza iesirea.
  end with;
requirement CONTROL_DE_STARE is
  interface relationship is
      trigerred by GESTIUNE_TIMP_REAL
        with &(INTERVAL_TIMP_REAL)&:event;
      use CONTROL_INTRARI_PROCES
        to PRELEVEAZA $(M1, M2, M3, M, M6, M7:analog,
                        M4, M8:digital)$;
      acknowledge CONTROL_IESIRE
        with GATA:event;
      used by CONTROL_IESIRE
      to OBTINERE(VALOARE_IESIRE:VALOARE_IN_TREPTE);
      type VALOARE_IN_TREPTE is
        $(IESIRE_FOARTE_MICA, IESIRE_MICA, IESIRE_MARE)$
        of analog;
  end interface relationship;
requirement body CONTROL_DE_STARE is
  OBTINERE(VALOARE_DE_IESIRE:VALOARE_IN_TREPTE) is
      object:$(M1, M2, M3, M5, M6, M7:analog, M4, M8:digital)$;
      object:STABILIRE_PUNCT:VALOARE_IN_TREPTE;
        type:VALOARE_IN_TREPTE is
          $(IESIRE_FOARTE_MICA, IESIRE_MICA, IESIRE_MARE)$
          of analog;
      object:STARE:INTERNAL_ENTITY.1;
        type INTERNAL_ENTITY.1 is
          $(STARE1, STARE2, STARE3, STARE4)$;
        --cind (OBIECT_MOBIL este in ZONA1) => STARE1.
```

```
        --cind (OBIECT_MOBIL este in ZONA2) => STARE2.
        --cind (OBIECT_MOBIL este in ZONA3) => STARE3.
        --cind (OBIECT_MOBIL este in ZONA4) => STARE4.
      begin
        PRELEVEAZA$(M1, M2, M3, M4, M5, M6, M7, M8)$
        estimate STARE
          using $(M1, M2, M3, M4, M5, M6, M7)$;
        $(
        case INTERNAL_ENTITY.1 is
          when(STARE1 => true, others => false) =>
            (VALOARE_DE_IESIRE:=IESIRE_FOARTE_MICA);
      ...
      ...
      ...
        end OBTINERE;
      begin
        --initializare;
        accept INTERVAL_TIMP_REAL;
        acknowledge CONTROL_IESIRE
          cu GATA;
  end requirement body CONTROL_DE_STARE;
mapping is
  => (package CONTROL_DE_STARE
      which consists of HANDLER_INTRARI, SELECTARE_IESIRE,
CONTROL_IESIRE)
      where
        (M1, M2, M3, M4, M5, M6, M7, M8) =>
            (X1, X2, X3, X4, X5, X6, X7, X8);
            --in SELECTARE_IESIRE
        (IESIRE_FOARTE_MICA, IESIRE_MICA, IESIRE_MARE) =>
            (FOARTE_MIC, MIC, MARE)
            --in SELECTARE_IESIRE
        (STARE1, STARE2, STARE3, STARE4) is invariant
      ...
      ...
      ...
  end mapping;
  required facilities are
    --sistem de operare;
    --subsistem utilitar;
    --procesor de limbaj;
    --marimea memoriei;
  end required facilities;
end requirement CONTROL_DE_STARE;
```

Se observă că în ultima secțiune este cuprins blocul de corespondențe (mapping block), unde sunt cuprinse numele variabilelor, tipurile și operațiile care instanțiază cerințele. De asemenea, o secțiune expresă cuprinde resursele (facilitățile) necesare.

În continuare, mai rămâne de demonstrat în ce mod poate fi refolosit pachetul CONTROL_DE_STARE şi prezentarea sa, respectiv cum poate fi instanţiat pentru o aplicaţie diferită de prima. Articolul prezintă o aplicaţie numită ANALIZA_TELEGRAMA. Un fragment din descrierea cerinţelor pentru această aplicaţie este prezentat în continuare :

Exemplu de instanţiere a programului, pentru aplicaţia ANALIZA_TELEGRAMA
(prezentare prescurtată)

```
requirements description for ANALIZA_TELEGRAMA
with
object:MagneticTape;
      --este inregistrat un fisier; acesta are mai multe
      --inregistrari, fiecare avind o dimensiune MaxRecordSize
      -- diferita.

object:TELEGRAMA;
      --TELEGRAMA este inregistrata in MagneticTape.
      --un TEXT consta in mai multe inregistrari (RECORDs).
      --un CUVINT nu poate fi separat pe mai multe RECORDs.

type TELEGRAMA is
  record
      for all EMITATORI
        loop
          record;
            Description:TEXT;
            'ZZZZ';
          end record;
        end loop;
        'EOF';
    end record;
  end TELEGRAMA;
object:TEXT;
type TEXT is
  for all CUVINTs
  loop
      record
        blankspace;
        characterstring;
        end record;
    end loop;
  end TEXT;
object RAPORT;
...
...
...
```

```
requirements AnalizaTelegrama is
  interface relationship is
      trigerred by command "START AnalizaTelegrama";
      use MagneticTape
        to OBTINE(INTRARE:TELEGRAMA);
      acknowledge LinePrinter
        with GATA:event;
      used by LinePrinter which will
        OBTINE(DOCUMENT:RAPORT);
      end interface relationship;

requirements body AnalizaTelegrama is
  relationship OBTINE(DOCUMENT:RAPORT) is
      object:CUVINT;
      object:SimbolIntrare;
        --type of SimbolIntrare is
        --(CuvintOrdinar, CuvintDepasit).
      object:STARE;
        --type of STARE is (STARE1, STARE2, STARE3,
        --STARE4, STARE5, STARE6).
        --when(stare initiala) => STARE1
        --when(este receptionat un CUVINT cu marime ordinara)
        --                    => STARE2
        --when(este receptionat un CUVINT cu marime depasita)
        --                    => STARE3
        --when(trebuie generat un RAPORT) => STARE4
        --when(sfirsit normal de fisier)  => STARE5
        --when(sfirsit ilegal de fisier)  => STARE6
  begin
      OBTINE(INTRARE);
      compute SimbolIntrare;
      if in STARE1 then
        initializa the system;
      elseif in STARE2 then
        count up CUVINT Ordinar;




  end requirements body AnalizaTelegrama;
end requirements AnalizaTelegrama;
```

Constatând că AnalizaTelegrama poate fi modelată prin tranziții de stări, comparăm descrierea cerințelor de mai sus, cu prezentarea pentru CONTROL_DE_STARE, forma **Form(1,Q*)** prezentată anterior. Din comparație rezultă că se pot realiza următoarele asocieri între entitățile și relațiile din cele două forme :

(event INTERVAL_TIMP_REAL) =>
(event COMANDA("START AnalizaTelegrama"),
(event CuvintulUrmatorVaRog);
(M1, M2, M3, M4, M5, M6, M7, M8) => CUVINT;
(event CALCUL_VITEZA(INTRARE_NOUA:CADRU)) =>
(event StareCuvintNou(STARE:MARIME));
(event STABILIRE_VITEZA(VITEZA_CAJE)) =>
(print NumarTelegrama, MarimeCuvint, NumarCuvinteDepasite);
(NU_TAGLA, VENIND LAMINARE, IESIND, ILI., STP) =>
(INIT, CuvintOrdinar, CuvintDepasit, GenRaport, EOF, SfTlegEOF);
(I1,...,I11) => (MarimeOrdinara, MarimeDepasita, SfirsitText, EOF);

Folosind aceste convenţii, prin înlocuire se poate obţine programul complet pentru aplicaţia AnalizaTelegrama, plecând de la CONTROL_DE_STARE în forma de la nivelul program. Exemplul complet este prezentat în articol.

Algoritmul complet de refolosire arată în concluzie astfel :

- **Etapa realizării proiectului general** :
  - se parcurg etapele **Form(1) ... Form(3)** pentru un proiect complet ;
  - se construieşte o **prezentare** a formei **Form(1)** care este o generalizare a acesteia ;
- **Etapa reutilizării** :
  - la aplicaţii bănuite a fi din aceeaşi clasă, se scriu cerinţele sub forma **Form(1)** ;
  - se compară forma obţinută cu prezentarea şi se extrag corespondenţele ;
  - se înlocuiesc entităţile şi relaţiile din forma **Form(3)**, conform corespondenţelor stabilite, obţinând astfel aplicaţia nouă.

Folosirea acestui algoritm de reutilizare software înseamnă, desigur, în primul rând, respectarea unor reguli care ţin de organizarea activităţii de proiectare :

- instituirea, în cadrul organizaţiei, a obligativităţii scrierii modulelor pentru reutilizare.
- promovarea scrierii prezentărilor;
- reţinerea prezentărilor în forma **Form(1, Q*)** şi a formei corespondente **Form(3, Q*)**, de către mediul de programare folosit (un instrument CASE);
- când sunt dezvoltate produse noi, se cere fiecărui proiectant să urmeze următoarea procedură:
  - după ce proiectantul obţine forma **Form(1,P)** pentru un program, acesta trebuie să caute în biblioteca de module de program la care are acces, cod existent pentru o posibilă reutilizare;

- dacă se găseşte un modul potrivit, proiectantul selectează **Form(1, Q*)** respectiv corespondentul **Form(3, Q*)** şi transformă pe aceasta din urmă în **Form(3,P).**

În concluzie, *refolosirea unor module la niveluri înalte de abstractizare creşte posibilităţile de reutilizare*. În articolul analizat a fost folosit un limbaj cu sintaxa **ADA**, la toate nivelurile, pentru a simplifica utilizarea şi a reduce costul instrumentelor software şi a pregătirii programatorilor. **Prezentarea** modulelor reutilizabile se realizează la nivelul cerinţelor întrucât astfel este mai uşor de înţeles scopul modulelor. În aceste prezentări, domeniul fiecărei descrieri care poate fi adaptate la aplicaţie, este clar vizualizat. Utilizând aceste prezentări şi **urmele** (conexiunile) spre nivelul 3, memorate într-un sistem de arhivare, proiectantul poate uşor regăsi modulele pe care le poate reutiliza.

6) Articolul [Ker84] se ocupă cu reutilizarea sub sistemul de operare **UNIX**. Filosofia **UNIX** încurajează reutilizarea. După autorul articolului, nivelele de reutilizare sunt în acest caz :

- nivelul bibliotecilor (cazul relaţiei (1.1,a) preluată din [Mit87]);
- nivelul limbajului de programare;
- nivelul programelor;
- nivelul sistemului;
- nivelul conceptual.

7) Parametrizarea, ca şi tehnică de reutilizare, este analizată în [Gog84]. Prin această tehnică, sunt adaptate interfeţe foarte generale pentru modulele de program, la condiţiile specifice prin parametri care descriu (localizează) aceste condiţii. Astfel, de fapt, interfeţele includ mai mult decât informaţii sintactice. Programele noi se obţin prin instanţierea parametrilor fiecărui modul. Se foloseşte şi "încapsularea" codului în module.

De multe ori, modulele de program nu au o formă imediat utilizabilă în aplicaţii noi. În acest caz, asupra codului se aplică unele transformări prin :

- adăugarea de linii-sursă, corespunzătoare unor noi funcţionalităţi;
- redenumiri de elemente de program;
- eliminarea unor funcţionalităţi prin eliminarea liniilor-sursă corespunzătoare.

Pentru ca un limbaj şi un mediu de programare să suporte programare parametrizată, ca trebuie să îndeplinească anumite condiţii, să permită implementarea facilităţilor de rezolvare a unor probleme specifice reutilizării. Câteva din aceste condiţii şi facilităţi sunt :

- modularitatea;
- să permită definirea unor structuri ierarhice ;
- să permită utilizarea unor biblioteci de funcţii ;
- să conţină mecanisme de definire şi verificare a tipurilor de variabile ;
- să conţină mecanisme care să permită o parametrizare puternică, adică să permită nu numai adaptări prin modificări de numere sau indecşi diferiţi, ci eventual să permită substituirea unor module întregi de program, diferite, pentru diverse instanţe ;
- cerinţele pentru parametri trebuie astfel satisfăcute încât programul rezultat să prezinte siguranţă în funcţionare în condiţiile concrete pentru care a fost instanţiat ;
- "*Teorii*" şi "*Vederi*" : este util ca să fie posibilă declararea acelor proprietăţi ale modulelor care pot fi utilizate ca parametri în alte module ; "*Teoriile*" declară astfel de proprietăţi, iar "*Vederile*" indică modul în care module date satisfac "*teorii*" date ;
- "*ascunderea*" informaţiei ("*hiding*") înseamnă ascunderea detaliilor de implementare ;
- posibilităţi de modificare facilă a modulelor de program ;
- simplitatea, importantă pentru înţelegerea uşoară a programului, deci şi a facilitării reutilizării ;
- facilităţi de evidenţiere a semanticii - cresc perceptibilitatea programului ;
- dezvoltare interactivă a programului.

Nu toate aceste atribute sunt simultan necesare şi de obicei un limbaj şi un mediu-suport nici nu le asigură pe toate. În articolul citat se defineşte un limbaj de programare, OBJ, şi un mediu asociat. Limbajul este orientat spre obiecte (lucrează cu date şi cu operaţii asociate acestora) şi implementează cerinţele de mai sus, folosind, faţă de alte limbaje, două noi concepte (amintite mai sus) :

a) **Teoriile** - au ca scop exprimarea proprietăţilor unui modul (sau interfaţă de modul) ca un întreg şi conţin definiţiile proprietăţilor cerute unui parametru actual pentru a putea înlocui parametrul formal al unui modul parametrizat dat.
b) **Vederile** - exprimă posibilitatea de satisfacere a unei teorii date într-un anumit mod de către un anumit modul; acestea sunt necesare întrucât este posibil ca modulele să satisfacă teorii în mai multe moduri posibile. Deci o vedere arată explicit cum un modul dat satisface o teorie dată.

În articolul citat sunt prezentate exemple de teorii şi vederi pentru programe scrise în **OBJ**. Aceste abordări sunt inspirate din lucrarea [Gog78], citată în articol. Spre exemplu, o "vedere" a unui obiect **A** la teoria **T** constă în corespondenţa de la categoriile din **T** la categoriile din **A** care conservă relaţiile dintre subcategorii, şi în corespondenţa de la operaţiile din **T** la operaţiile din **A** care păstrează toate atributele, astfel încât fiecare ecuaţie din **T** poate fi verificată pentru fiecare model al lui **A**. O vedere este deci un **morfism** al unei teorii.

Al treilea concept important introdus de abordarea din [Gog84] constă în **expresii de module**, care produc noi module prin modificarea celor existente. Aceste expresii pot fi considerate un fel de generalizare a transformărilor de program, configurate structural (corespunzând relației (1.1.,c) citate din [Mit87]).

8) Articolul [Lit84] este consacrat prezentării unei metodologii bazate pe algebra formală, care permite proiectarea prin refolosirea unor componente reutilizabile, scrise în **ADA**. Astfel, folosind teoria categoriilor, pot fi dezvoltate limbaje de specificații care permit exprimarea claselor de mediu (de context) pentru componentele reutilizabile. Autorii consideră că sunt necesare două formalisme distincte pentru reprezentarea și tratarea proiectării și respectivmanipulării componentelor reutilizabile, din pricina cerințelor deosebite pentru specificațiile de intracomponente și intercomponente. Aceste noțiuni sunt definite astfel :

- **punctul de vedere intramodul** - este proiectarea structurii interioare a unui modul de program ;
- **punctul de vedere intermodul** - exprimă și gestionează relațiile dintre module văzute ca și "cutii negre"; aceasta consideră comportarea modulului din punctul de vedere al specificațiilor exterioare ale modulelor.

În plus, trebuie menținută **consistența** între aceste două puncte de vedere

Mecanismele de încapsulare oferă posibilitatea definirii elementelor de interfață exterioară a modulelor software, deci satisfacerea cerințelor intermodul. Satisfacerea cerințelor intramodul este posibilă pe baza trăsăturilor clasice de program de nivel înalt. Articolul citat se referă la utilizarea limbajului **ADA**, dar concluziile pot fi extinse și asupra limbajului C++, prin facilitățile și construcțiile pe care acesta le pune la dispoziția utilizatorului.

Metodologia pentru reutilizare, dezvoltată de autori, este de tip *top-down* și presupue folosirea componentelor generice. Esența metodologiei este presupunerea că o componentă lucrează prin schimb de informații între instanța sa și mediul înconjurător, corespunzător acestei instanțe. Fiecare componentă este implementată la două nivele ierarhice imbricate, astfel încât trebuie de fapt dublu instanțiată, în ordinea utilizării : în prima etapă, schimbul de informații dintre instanță și mediu privește tipul de interes reciproc al informațiilor schimbate, iar la nivelul al doilea de instanțiere, funcțiile de legătură corespunzătoare acestor tipuri sunt pasate de la mediu spre componentă. Astfel, componenta poate fi referită "cu înțeles" de către mediu, iar facilitățile care trebuie exportate către mediu sunt acum bine definite și disponibile pentru a fi întrebuințate.

Pentru a urma această metodologie, componentele reutilizabile trebuie să posede interfețe vizibile standard atât pentru includerea în contextul mediului cât și pentru a accepta componente de nivel scăzut (nivelul al doilea). Întâi, pentru a asigura posibilitatea refolosirii la orice nivel în cadrul sistemului, este necesar ca metoda de proiectare să determine potrivirea

interfețelor vizibile între componentele de pe nivelele succesive. Un limbaj precum ADA permite construcția modulelor generice (parametrizate) dar nu permite ca un tip formal generic particular să fie reprezentat printr-o structură algebrică particulară. Adică, nu este posibilă definirea unei semnatici dorite pentru tipurile și funcțiile pe care modulul trebuie să le importe (această cerință este rezolvată în C++ prin supradefinire). Rezolvarea acestei probleme este prezentată mai departe.

Există limbaje de specificații care sunt potrivite pentru exprimarea specificațiilor intercomponente prin exprimarea functorilor definiți pe structuri algebrice. Astfel, [Gog84] (v. pct. 7), amintește de **morfisme de teorii**, care de fapt sunt functori între categorii ale teoriilor structurate. În sintaxta limbajului **CLEAR** [Bur80], există unele elemente interesante, în sensul care interesează lucrarea de față :

- **theory X ... endth** descrie o teorie ; o astfel de teorie descrie un set de algebre, fiecare din acestea :
  - asigură un set de elemente de date corespunzând fiecărui sort al teoriei;
  - asigură o funcție de acele seturi corespunzând fiecărui operator;
  - satisfac axiomele teoriei;
- operatorul **combine** crează coprodusul a două teorii ; operatorul acesta permite teoriilor să utilizeza subteorii comune (astfel, dacă ambele teorii conțin o teorie **BOOL**, coprodusul lor va conține o singură copie **BOOL**, nu două) :
- operatorul **enrich E by X enden** crează o nouă teorie **E** adăugând noi elemente specificate de **X** (de fapt, cu acest operator, fiecare teorie poate fi privită ca o îmbogățire a unei teorii vide) :
- operatorul **derive D from E by M : D --> E endde** desemnează teoria obținută din câtul teoriei **D** cu imaginea inversă din **M** a ecuațiilor lui **E** ; acest operator poate fi utilizat după construcția teoriilor cu **combine** și **enrich** , dacă teoria rezultată conține mai mult decât este de dorit

Acești operatori permit construirea de noi teorii în terminii celor definite anterior. Pentru a manevra adecvat subteoriile, fiecărei teorii i se asociază numele tuturor subteoriilor de care aceasta depinde. Corespondențele acestor nume la teoriile propriu-zise se numește **baza** teoriei iar teoria propriu-zisă este numită **teorie cu bază**. Limbajul **CLEAR** permite utilizatorului crearea propriilor proceduri de construire de teorii, plecând de la operatorii de construcție definiți mai sus. Aceste proceduri proprii au ca argumente teorii satisfăcând un **meta-sort**, un model (o paradigmă) pentru o clasă de teorii. Procedura este scrisă folosind meta-sortul ca și parametru formal, iar apelul acesteia este realizat cu argument o teorie actuală, asociind sorturile și operațiile teoriei actuale cu cele ale meta-sortului. Articolul citat prezintă posibilitatea implementării în **ADA** a unor operatori din cei propuși.

9) Lucrarea [Che84] tratează tehnica *prototipizării rapide* şi ajustarea la cerinţele clientului ("*Custom tailoring*"). Ajustarea la cerinţele clientului înseamnă dezvoltarea, plecând de la un program abstract, unei familii de programe concrete, fiecare potrivit unui mediu-ţintă specific. După cum se vede, această definiţie aduce cu ceea ce peste 3 ani avea să numească Mittermeir [Mit87] clasă de aplicaţii, obţinută plecând de la un program generic. Un program abstract ajunge astfel ascendentul unei familii de variante, iar fiecare variantă poate să suporte mai multe cicluri de prototipizare înainte de a obţine forma finală. În articolul citat este prezentat un mediu de dezvoltare - **PDS** (***Program Development System***) - care va fi prezentat la paragraful care tratează generatoarele de aplicaţii (§ 1.7.1). Autorul este preocupat şi de dezvoltarea unui limbaj pe două nivele : o componentă de nivel foarte înalt (**VHLL** - Very High Level Language) şi una de nivel de bază (limbaj convenţional de nivel înalt).

ANEXA 1.2

Câteva generatoare comerciale de aplicații

a) **XFaceMaker** [CPN94] este un program **UIMS** (*User Interface Management System*) care poate fi utilizat la proiectarea interactivă a interfețelor grafice. Este deci un **generator de interfețe**, o parte consistentă a oricărei aplicații software (deci este un **GA** pentru un subset al funcțiilor unui sistem).

Oferă funcții avansate de definire a comportării interfeței și posibilitatea de a crea șabloane (*templates*). Permite crearea de **clase** de interfețe grafice, **automat**, conform standardului **X-Intrinsics** și oferă suport pentru aplicații multi-display.

b) **CAPTOOLS** [CPN94] este un **pachet specificator de sisteme** care permite crearea aplicațiilor concurente în timp real. Printr-unul din programele pachetului (**"Cap cg"**) oferă și servicii de **GA**, acel program permițind generarea codului **C documentat** al aplicației.

Suportă structuri *pipe-line, date distribuite, modele master slave*. Utilizînd o interfață grafică, se poate defini diagrama software și arhitectura hardware a sistemului. O componentă a pachetului (**"Cap simu"**) permite simularea mediului țintă, în absența hardware-lui respectiv.

c) **VisualMira C++** [CPN94] este un mediu interactiv de dezvoltare sub **Windows 3.1**, care integrează mai multe instrumente, incluzînd un editor cu analizor sintactic pentru limbajul de specificație **MIRA**, un **generator de cod** și un sistem de interfață grafică. Poate să lucreze cu **Visual C++** sau **Borland C++**.

d) **iXBUILD3** [CPN94] este un **generator de interfețe GUI** (*Graphical User Interface*) modulare și reutilizabile pentru aplicații MOTIF.

Oferă facilități deosebite de apelare de funcții și astfel simplifică puternic testarea unei aplicații. Folosește biblioteci de clase, ca de altfel toate generatoarele comerciale amintite în anexa aceasta, precum și majoritatea celor existente pe piață.

e) **Progress** [CPN94] este un pachet de aplicații de tip **CASE**, conceput special pentru medii care sînt în continuă schimbare și reorganizare, fiind un mediu complet de dezvoltare pentru aplicații mari, portabile și reconfigurabile, folosind **cod reutilizabil**. Pachetul este alcătuit din următoarele componente: **Progress ADE** - care este un mediu de dezvoltare a

aplicațiilor, **Progress RDBMS - administrator de bază de date** (similar **SGBS**) și **Progress DataServer Architecture** - care permite utilizatorilor să dezvolte aplicații independente de baze de date construite cu **Progress RDBMS**. Sistemul permite de asemenea conexiuni cu **User Interface Builder** (*Constructor de Interfețe Utilizator*) - care este un mic **generator de interfețe**.

Sistemul folosește filosofia **POO** pentru a putea utiliza intensiv facilitățile de **reutilizare** rezultate, dar fără ca utilizatorul să simtă complexitatea dezvoltării, asociată acestei filosofii. Sistemul, în versiunea 7.3, include cadre de **cod reutilizabil** (templates), proceduri persistente, multe modele de interfețe utilizator (*layouts*), versiuni grafice pentru un sistem de dezvoltare a părților de program care elaborează rapoarte (**Progress Report Builder**). Astfel, noile aplicații pot fi dezvoltate grafic, în plus se poate trece foarte ușor la diverse seturi de caractere.

f) **XView** este un mic **generator de interfețe** prietenoase, care permite definirea unor programe care să utilizeze obiecte grafice (cadre, butoane, setări, meniuri, etc.), prin folosirea facilităților oferite de limbajul **C++**. Printr-un dialog simplu, ușor de condus, utilizatorul își dimensionează și amplasează ferestrele, își definește comenzile de meniu și denumirile butoanelor, precum și numele funcțiilor apelate în situațiile acționării comenzilor definite. Codul generat conține funcțiile de tratare a diverselor comenzi, pregătite pentru a fi completate de utilizator cu secvențele de program care să efectueze acțiunile dorite. Programul se obține prin compilarea codului obținut, după ce a fost completat cu partea de cod specifică, apoi prin linkeditarea cu o bibliotecă proprie a generatorului. Generatorul menționat este disponibil pe Internet, iar informații se pot afla de la autorul programului, Antonio Carlos Moreirao De Queiroz, Department of Electronic Engineering and COPPE, Federal University of Rio de Janeiro, Brazil, email: acmq@coe.ufrj.br.

ANEXA 1.3

Colective de lucru şi bibliografie disponibilă

Reutilizarea software este, din motive pe care le-am detaliat pe parcursul prezentului capitol, o preocupare majoră pentru teoreticienii şi practicienii software, mai ales la nivel organizaţional şi academic. Numeroase grupuri puternice de lucru sunt angrenate în activităţi de teoretizare, definire, conturare, standardizare, a unor concepte şi metodologii legate de **reutilizarea software**. Astfel, preocupări majore există în acest sens în cadrul "Software Engineering Institute" (din cadrul Universităţii "Carnegy Mellon") din S.U.A. (adresa **URL** : **http://sw-eng.falls-church.va.us**), la "NATO Communications and Information Systems Agency" (**NACISA**) - Brussels, "Reuse Group" - S.U.A. (adresa **URL** : **http://www.reuse.com**), etc.

Tematica tezei de faţă include şi domeniul **măsurilor software**. De acest domeniu se ocupă de asemenea grupe de lucru la Universitatea Tehnică din Berlin (adresa **URL** : **http://www.cs.tu-berlin.de**), la Universitatea din Rochester (ftp: **ftp.cs.rochester.edu**), la "Software Engineering Management Research Laboratory" din cadrul Universităţii Quebec din Montreal, Canada, la "Software Engineering Institute" - S.U.A., în cadrul Laboratorului de Măsurări Software de la Universitatea Tehnică din Magdeburg, Germania, la Organizaţia Internaţională pentru Standardizare (**ISO**), la "Centrul Naţional de Cercetare pentru Tehnologia Informaţiei" - Germania (**http://www.gmd.de**), la "The US Airforce Software Technology Support Center" (**STSC**), grupul de lucru pentru proiectul **METKIT** finanţat de Comunitatea Europeană [MET93], grupul de lucru **Grubstake** (cercetători de la City Umversity şi South Bank Polytechnic, Londra, Colorado State University, Kansas State University - v. [Fen91]), etc. În cadrul grupurilor de ştiri **INTERNET**, există unul consacrat măsurilor software (**RFD : comp.software.measurement**), de asemenea, articole cu referire la acest domeniu sunt incluse în grupul de ştiri de inginerie software (**RFD: comp.software-eng**).

Instituţii specializate definesc **standarde** pentru a uşura utilizarea tehnicilor, metodologiilor şi instrumentelor asociate, pentru reutilizare şi metrică software. Astfel, Institutul Internaţional de Standardizare (Standardele **ISO 8402, ISO/IEC DIS 9126, ISO 9000-3**), IEEE Computer Society [ICS92-S] şi "NATO Communications and Information Systems Agency" (**NACISA**) - Brussels [NAT92-D][NAT92-M][NAT92-S] au fost definite

standarde referitoare la nomenclatură, tehnici, evaluare software, abordări organizaționale referitoare la reutilizare software, respectiv măsuri software.

Numeroase **manifestări internaționale** sunt consacrate domeniilor precizate. Numai pentru anul 1996, din lista manifestărilor (destul de cuprinzătoare), selectăm pe cele mai importante : International Workshop on Systematic Reuse (ian., Liverpool), ACM Computing Week (febr., Philadelphia), Software Technology Conference (apr., Salt Lake City), 4th International Conference on Software Reuse (apr., Orlando), Object World (mai, Boston), REUSE'96 (iulie-aug., Morgantown),International Conference on Software Engineering - ICSE (martie, Berlin, având un "workshop" dedicat unui Simpozion de Metrici), Software Quality Week (mai, San-Francisco), The Tenth European Conference on Object-Oriented Programming ECOOP'96 (iulie, Linz), The International Conference on Software Maintenance ICSM'96 (nov., Monterey), etc

În domeniile de interes ale lucrării de față, au fost publicate numeroase cărți și lucrări de bază. Astfel, câteva dintre lucrările de **teoretic a programării** mai noi, din domeniile de interes, sunt [Bas95-F], [ICS92-R], [Lim95], [Lie95] etc. Exemple de câteva lucrări cu valoare de întrebuințare **practică** imediată sunt : [Hoo91], [McG92], [Mey94], [Wal93] Unele din lucrările interesante, prezentate la **conferințe** științifice de prestigiu, sunt strânse în volumele : [Pri93], [Sch94], [Wie95], [Fra94].

După cum se remarcă, o întreagă comunitate științifică internațională, deosebit de activă, aduce contribuții continue la dezvoltarea domeniului reutilizării software și a metricilor în programare. Această preocupare susținută este o pledoarie în plus pentru importanța domeniului abordat și pentru actualitatea domeniului.

ANEXA 2.1

Exemplu de calcul al indicatorilor Halstead prezentați în § 3.1

Prezentăm în cele de mai jos, un exemplu de aplicare a relațiilor (3.1), pentru un fragment de program foarte simplu [Mih95] :

IF (A < B) THEN X = A;
ELSE X = B;

Din fragmentul de program de mai sus se deduce :

η_1 = 4 deoarece în program sunt 4 operatori : **if then else**, >, = ;
N_1 = 6 deoarece operatorii de mai sus apar de 6 ori :

η_2 = 3 deoarece în program apar 3 operanzi : **A**, **B**, **X** ;
N_2 = 6 întrucât acești operanzi apar în total de 6 ori :

Odată calculate aceste numere, ceilalți indicatori se determină prin aplicarea directă a relațiilor (3.1) :

a) **vocabularul** programului : $\eta = \eta_1 + \eta_2 = 4 + 3 = 7$
b) **lungimea observată** a programului : $L = N_1 + N_2 = 6 + 6 = 12$
c) **lungimea estimată** a programului :

$$\hat{L} = \eta_1 \bullet \log_2 \eta_1 + \eta_2 \bullet \log_2 \eta_2 = 4\log_2 4 + 3\log_2 3 = 12,76$$

d) **volumul** programului : $V = L \cdot \log_2 \eta = 12 \log_2 7 = 33,68$
e) **dificultatea** programului :

$$D = \frac{\eta_1 N_2}{2\eta_2} = \frac{4*6}{2*3} = 4$$

f) **nivelul** programului :

$$L_1 = \frac{1}{D} = \frac{1}{4} = 0,25$$

g) **efortul** :

$$E = \frac{V}{L1} = \frac{33,68}{0,25} = 134,72$$

h) **numărul de erori** :

$$B = \frac{V}{3000} = \frac{33,68}{3000} = 0,01123$$

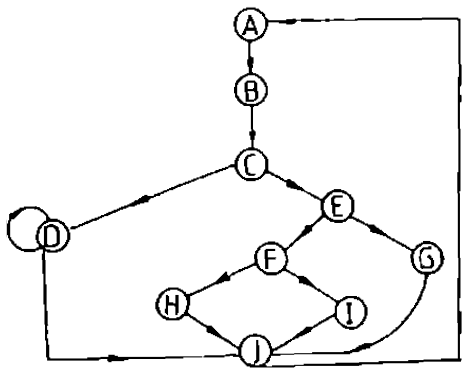
i) **timpul** :

$$T = \frac{E}{S} = \frac{134,72}{18} = 7,48$$

ANEXA 2.2

Exemple de calcul al complexității ciclomatice (§ 3.1)

Exemplul 1 : În [Mih95] este dat un exemplu simplu de calcul al compexității ciclomatice. Modulul de program pentru care se calculează această complexitate poate fi reprezentat cu ajutorul grafului strict conectat din figură, în care fiecare nod reprezintă un bloc de instrucțiuni executate secvențial :



Pentru situația din figură, datele deduse pentru modulul respectiv sunt :

Numărul de noduri N = 10
Noduri bifurcate : C, D, E, F deci
SN = 4
Numărul de arce E = 14

Regiunile sunt mărginite de traseele :

ABCEGJA și JA
ABCEFIJA și ABCEGJA
ABCEFHJ și ABCEFIJ
ABCDJ și ABCEFHJ
D-D deci RG = 5

Complexitatea ciclomatică este deci (în cele 3 exprimări din rel. 3.4) :

C = E - N + 1 = 14 - 10 + 1 = 5
C = RG = 5
C = SN + 1 = 4 + 1 = 5

Exemplul 2 : Calculul complexității ciclomatice pentru programul de mai jos, este prezentat în [Ram88] :

```
VAR
     a, b, c, d, m : integer;
BEGIN
     readln(a, b, c, d);
     IF a > b THEN
          IF b > c THEN
               m := a + b
          ELSE
               m := b + c
     ELSE
          m := b + c + d;
     writeln(m)
END.
```

Complexitatea ciclomatică, calculată conform relației (3.4,a), este, pentru programul de mai sus :

C = E - N + 1 = 13 - 11 + 3

ANEXA 2.3

Exemplu de calcul pentru măsurile ponderate (§ 3.1)

În lucrarea [Ram88] este dat un exemplu de calcul pentru măsurile ponderate. Exemplul se referă la programul din exemplul al doilea din Anexa 2.2.

În programul respectiv, structura externă **IF ... THEN ... ELSE** este considerată ca fiind de nivelul 1. Deci, complexitatea sa ciclomatică (v. § 3.1) este 2, iar ponderea sa asociată este 1. Structura internă **IF ... THEN ... ELSE** este considerată de nivelul doi, deci complexitatea sa ciclomatică este 3, iar ponderea este 2. Operatorii de la cele două nivele (deci cei doi operatori ">") au şi aceştia, ponderile respective. Deoarece toţi ceilalţi operatori din program au pondere nulă, rezultă (rel. 3.5,a) :

$$N_{w1} = N_1 + 6 = 22 + 6 = 28$$

Operanzii care au ponderi nenule în program, apar în clauzele **IF** ale celor două structuri **IF ... THEN ... ELSE** : "a" are ponderea 1, "b" are o dată ponderea 1, apoi, pe nivelul celălalt, ponderea 2, iar "c" are ponderea 2. Deci, conform rel. (3.5,b) :

$$N_{w2} = N_2 + 6 = 19 + 6 = 25$$

Măsurile **lungime**, **volum**, respectiv, **efort**, ponderate, vor avea deci valorile (conform relaţiilor 3.5,c-e) :

$$N_w = 28 + 25 = 53$$

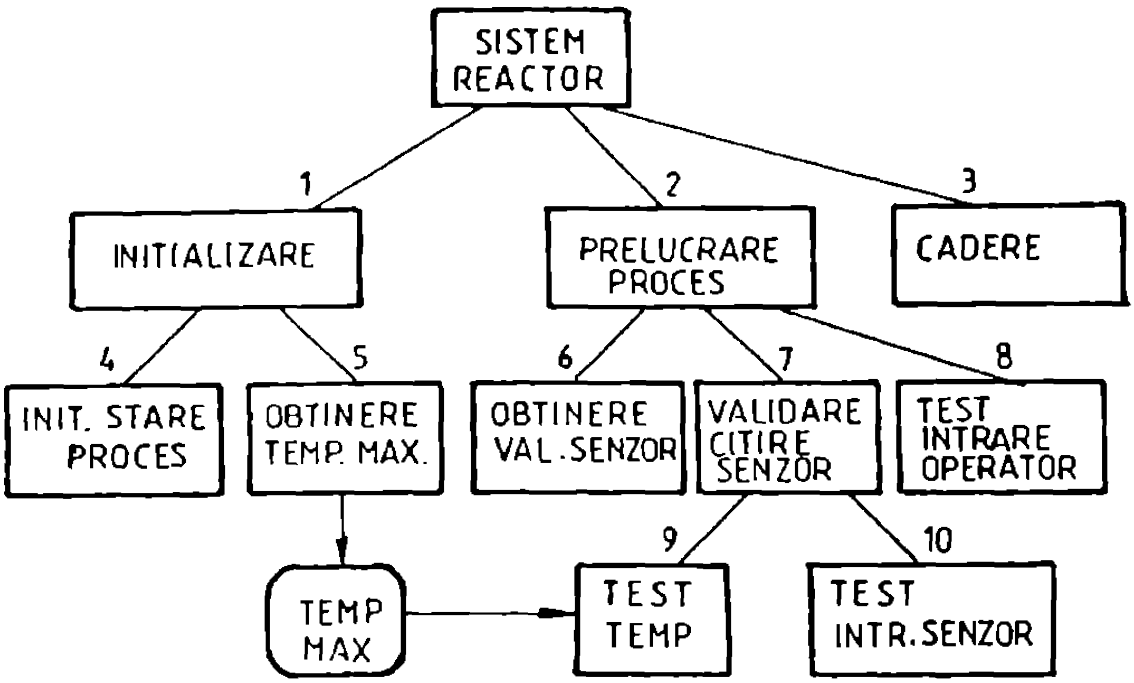
$$V_w = 53 * \log (11 + 5) = 212$$

$$\hat{E}_w = 212 * [(11 * 25) / (2 * 5)] = 5830$$

ANEXA 2.4

Exemplu de calcul pentru determinarea coeficientului IF4, prezentat în § 3.1, D

Prezentăm în cele ce urmează, un exemplu de aplicare a relației de definire (3.6), pentru un program de comandă a unui reactor [She95], reprezentat în figura de mai jos :



Arhitectura programului de comandă a unui reactor

În tabelul următor, au fost trecute conexiunile directe și inverse între module :

Interfața	Conexiune directă	Conexiune inversă
1	-	stare_reactor
2	stare_reactor	stare_reactor
3	stare_reactor	cod_eroare
4	-	stare_reactor
5	-	cod_eroare
6	-	nr_senzor, temperatura, stare_reactor
7	nr_senzor, temperatura	stare_reactor
8	-	stare_reactor
9	temperatura	stare_reactor
10	nr_senzor, numar_ultim_senzor	stare_reactor

Ținând cont de diagrama din figură și de tabelul conexiunilor de mai sus, calculele de determinare a coeficientului **IF4** sunt sintetizate în tabelul următor :

Modulul	FI	FO	FI*FO	$IF4_i$
sistem reactor	5	2	10	100
initializare	0	1	0	0
prelucrare-proces	6	3	18	324
cadere	1	1	1	1
initializare-stare-reactor	0	1	0	0
obtinere-temp.-maximă	0	1	0	0
obtinere-valoare-senzor	0	3	0	0
validare-citire-senzor	4	4	16	256
test-intrare-operator	0	1	0	0
test-temperatura	2	1	2	4
test-nr.-senzor	2	1	2	4
			Total :	689

Deci **IF4** = 689

ANEXA 3.1

Generarea de mesaje prezentată în § 6.1

Subrutina de generare de mesaje

```
;
;*********************************************************
;
;     Denumire: Subrutina construire mesaj
;     Cod: CONMES (CONstruire MESaj)
;     Descriere: se construieste un mesaj compus din mai
;                multe siruri de caractere, pe structura
;                unui text de baza ; un text dureaza pina
;                la b 7 = 1
;     Intrari: in bc - adr. bufferului de formare mesaj
;              in de - adr listei de parametri actuali
;              in hl - adr structurii de descriere mesaj
;     Iesiri : in bufferul indicat - mesajul format conform
;                                    valorii parametrilor
;                                    actuali
;              CARRY = 0 - formare OK
;                    = 1 - eroare (valoare parametru incorecta)
;     Subrutine apelate: MVSR1 (transfer text pina la b 7 = 1)
;     Data elaborarii : mai 1987
;
;*********************************************************
;

      cseg
      public    CONMES

CONMES:
      push      bc          ; salvare pointer buffer
      push      de          ; salvare adr. listei de param.
      ld        e,(hl)      ; extragere adresa biblioteca
      inc       hl
      ld        d,(hl)
      push      de
      exx                   ; salvare adrese curente
      pop       hl
      exx
      inc       hl          ;    adr. numarului mesajului de baza
      push      bc          ; aici, adresa buffer
      ld        c,(hl)      ;.  in c va fi nr. mesaj baza
      dec       c           ;  corectie
      sla       c           ;   inmultire cu 2 (calcul
```

```
                                ;         deplasament)
        ld      b,0             ;     in bc va fi deplasamentul
        push    de
        pop     ix              ;  in ix va fi adr. structura
        push    ix
        add     ix,bc           ;  in ix va fi adresa adresei
                                ;    mesajului    de baza
        ld      e,(ix)          ;  extragere adresa mesaj de baza
        inc     ix
        ld      d,(ix)
        ex      de,hl           ; schimbare convenabila
        push    de
        pop     ix              ;  rearanjari
        pop     iy
        pop     de
        push    iy
        push    ix
        call    mvsrl           ;  transferul mesajului de baza in
                                                          ;    buffer
        push    de
        exx
        pop     de
        exx
        pop     hl              ; aici a ajuns adr. numarului mes. de
                                ;     baza
        inc     hl              ; avans la  nr. parametri
        ld      c,(hl)          ;  extr. nr. param.
        ld      a,c
        or      a               ;  nr. param. este 0 ?
        jr      z,gata          ;  da, gata
reia:
        inc     hl              ;   avans la deplasament
        ld      b,(hl)          ; extr. deplasament
        inc     hl              ; avans la adresa structurii pentru
                                ;    un param.
        push    bc
        ld      e,(hl)          ; extr. adr. structura pt. un param.
        inc     hl
        ld      d,(hl)
        push    hl
        pop     bc              ;  rearanjare (subrutina trebuie
        pop     ix              ;  sa fie reentranta deci nu lucreaza
        pop     iy              ;   decit cu stiva si registrele !
        pop     hl
        push    iy
        push    ix
        push    bc
        ld      a,(de)          ;   val. max. parametru
        push    de
        ld      e,(hl)          ;   valoarea parametrului
        sub     e               ;  comparare
        jp      p,bine          ;    e in regula
        pop     bc
        pop     de
        pop     hl              ;  altfel, se termina cu eroare
        pop     ix
        pop     iy              ;  salvari
        jr      fin             ;    si  gata
```

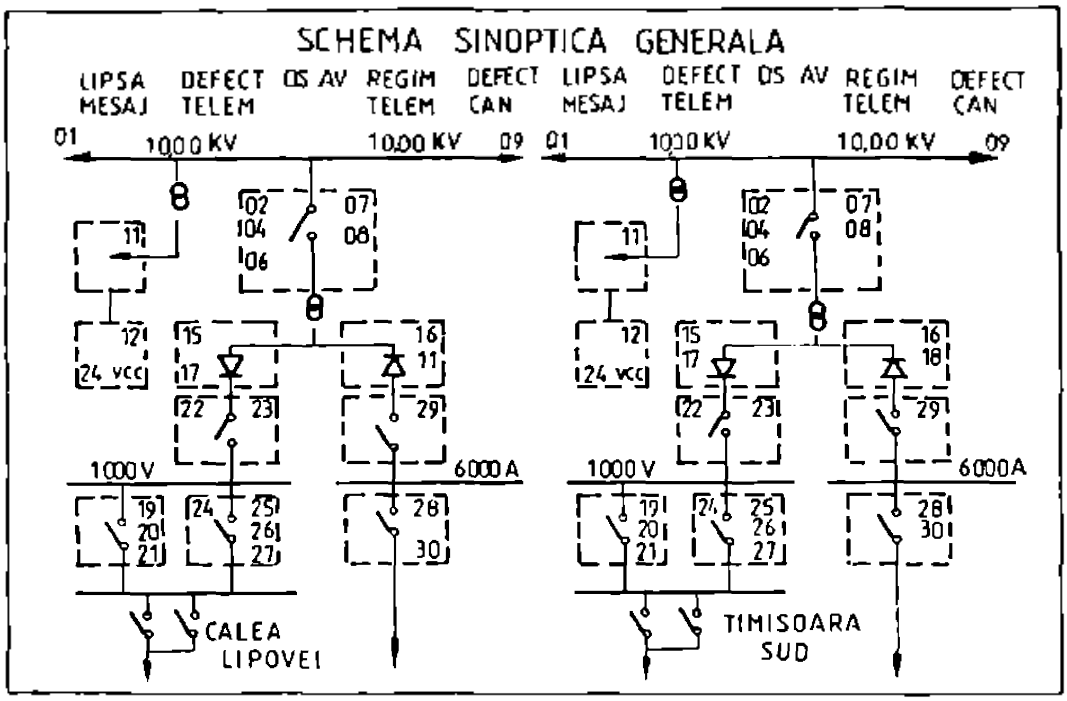
```
bine:
     ld        d,0
     pop       ix          ;  se calculeaza adr. nr. mesaj pt.
     add       ix,de       ;   valoarea actuala a parametrului
     ld        a,(ix)      ;  se extrage nr. mesaj respectiv
     dec       a           ;   corectie
     sla       a           ;    inmultire cu 2 (calcul
                           ;    deplasament)
     ld        e,a         ;  in de va fi deplasamentul
     push      de
     pop       ix
     pop       iy
     pop       bc ,
     pop       de          ;   aici, adr. tabel adrese mesaje
     add       ix,de       ;  calcul adresa mesaj curent
     pop       de          ;  readucere adr. buffer
     push      de
     push      iy
     push      de
     pop       iy
     ld        e,(ix)
     inc       ix          ;  extragere adr. mesaj
     ld        d,(ix)
     push      hl
     ex        de,hl
     push      bc
     ld        c,b         ;    in c va ajunge offset mesaj
     ld        b,0
     add       iy,bc       ;  calcul adr. dest. sir car.
     push      iy
     pop       de          ;    aducere adr. dest. la locul cerut
     call      mvsrl       ;transf.mesaj coresp. val.param.
     pop       bc          ;  in b este deplas. iar in c nr.
                           ;    param.
     ld        a,c
     pop       bc
     inc       bc          ;    avans la param. urmator
     pop       hl          ;    readucere pointer de structura
     exx
     push      hl
     pop       iy
     exx
     push      bc
     push      iy
     ld        c,a
     dec       c           ;  mai sint parametri ?
     jr        nz,reia     ;  da, se reia cu param. urmator
gata:
     pop       hl          ;    secventa
     pop       de          ;      de rearanjare
     pop       bc          ;  finala
fin: exx                   ;      a
     push      de          ;      parametrilor
     pop       iy
     exx
     ret                   ;     retur
```

O iniţializare a structurii de date pentru generare de mesaje
(aplicaţia DISPAT - [Tiv87-B])

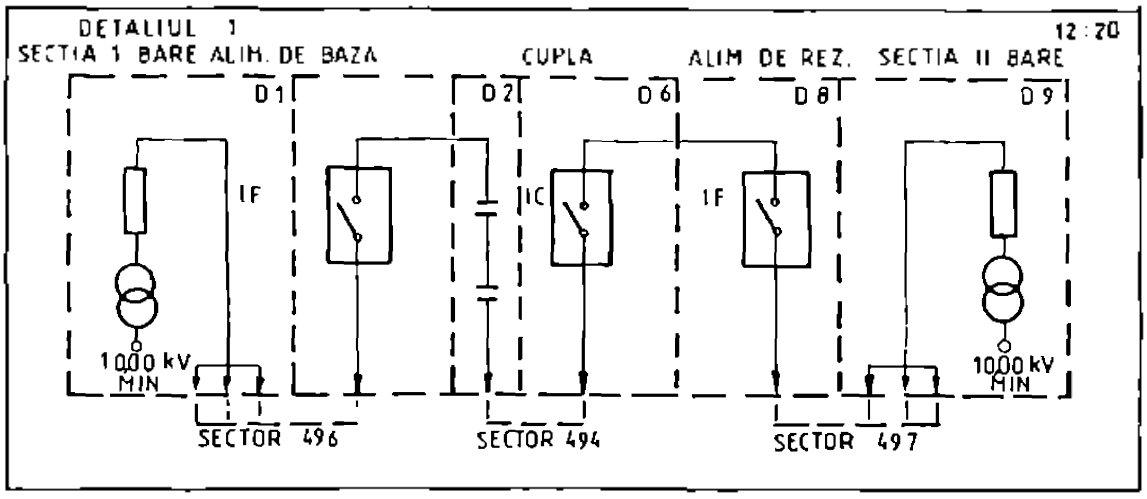
```
;
;     AVARII SISTEM
;
MSAS1:      DW    TABM              ;    adr. tabel mesaje
            DB    52                ;   text de baza
            DB    3                 ;    3 parametri de intrare
            DB    0                 ;      offset text 1
            DW    AS1               ;    adr. descriptor text 1
            DB    18                ;   offset text 2
            DW    AS2               ;   descr. text 2
            DB    23                ;   offset text 3
            DW    AS3               ;    descriptor text 3

AS1:        DB    2                 ;    val. max. prim parametru
            DB    3,4               ;      lista nr. logice texte
AS2:        DB    7                 ;   val. max. al doilea param.
            DB    5,6,7,8,9,10,11     ;  lista 7 nr. logice texte
AS3:        DB    7
            DB    53,54,55,56,57,58,59
...
...
...
;
;     texte propriu-zise
;
M1:         DC    '            LA     (  ) :                ,  '
M2:         DC    '    LA   (  ) : , '
M3:         DC    'APARE'
M4:         DC    'DISPARE'
M5:         DC    'C1'
M6:         DC    'C2'
M7:         DC    'C3'
...
...
...
M53:        DC    'LPS MES'
M54:        DC    'MES ER'
M55:        DC    'RAM DEF'
M56:        DC    'EPR DEF'
M57:        DC    'REL DEF'
M58:        DC    'CAN DEF'
M59:        DC    'MOD DEF'
M60:        DC    '        '

;
;     tabel adrese texte
;
TABM:       DW    M1,M2,M3,M4,M5,M6,M7,M8
            ....
            DW    M57,M58,M59,M60
```

ANEXA 3.2

Exemple de scheme sinoptice generate cu tehnicile prezentate în § 6.1



Schemă sinoptică generală



Detaliu de schemă sinoptică

ANEXA 3.3

Exemplu de inițializare a structurii de date pentru interfața inteligentă prezentată în § 6.2

Variantă pentru NUMEROM 633

```
; Panou NUMEROM 633 - STRUCTURI DE DATE

strdat    equ  0A000h

          aseg

          org  strdat                    ; adresa de inceput a
                                         ; struct. de date

TABAD:    DW   AD1,AD2,AD3,AD4
          DW   AD5,AD6,AD7,AD8           ; tabel   adrese
          DW   AD9,AD10,AD11,AD12        ;  de
          DW   AD13,AD14,AD15,AD16       ;   descriptori
          DW   AD17,AD18,AD19,AD20       ;          contexte

TABAM:    DW   MES1,MES2,MES3,MES4       ; tabel adrese mesaje
          DW   MES5,MES6,MES7,MES8       ;    de baza
          DW   MES9
;
;    Descrierea contextelor
;
;         la punere sub tensiune
;

AD1:      DB   1,30,81H,0                ; context
          DW   0                         ; "NUMEROM 633"
          DB   0,0,0
          DW   ADRL1                     ;   adr. tabel link
          DW   ADRUS1                    ;  adr. lista USER
          DW   0

ADRL1:    DB   2                         ;  link pe nr. logic 2
ADRUS1:   DB   2                         ;  USER 2 logic

;
;         regimul JOG
;

AD2:      DB   2,31,81H,4                ;   "A/R  JOG"
          DW   ADRD2
          DB   0
          DW   0,0,ADRUS2,0
```

```
ADRL2:      DB    2
ADRD2:      DB    TXP,AVXP,01H,2              ;  <+X>
            DB    TXM,AVXM,02H,2              ;  <-X>
            DB    TYP,AVYP,10H,2              ;  <+Y>
            DB    TYM,AVYM,40H,2              ;  <-Y>
            DB    TSTOP,CSTOP,4,3             ;  <STOP>
ADRUS2:     DB    3,4,9
;
;           regimul citire banda perforata CIT
;

AD3:        DB    3,32,81H,2                  ; "NP/PP    CIT"
            DW    ADRD3
            DB    0
            DW    0,ADRL3,ADRUS3,C

ADRL3:      DB    4
ADRD3:      DB    TSTCIC,CSTCIC,1,2           ;  <START CICLU>
            DB    TDEL,CDEL,0,1               ;  <DEL>
            DB    TNEXT,CNEXT,1,0             ;   pt. CTX.11
ADRUS3:     DB    5,6                         ;  pornire motor,

                                              ; resp. afisare PP sau
                                              ;   NP
....
....
....
;
;           regimul  IMD
;
...
...
...
AD13:       DB    7,37,41H,0                  ;  "M=       DM"
            DW    0
            DB    1
            DW    ADRF2,ADRL9,ADUS13,C

ADRF2:      DW    ZAFIS+1
            DB    1                           ;  ILS
            DB    0                           ; afisare fara

                                              ;  modificare
            DB    0                           ;  NLM
            DW    ADRR1                       ;   ARAM
            DW    1                           ;   DF
            DW    0,9                         ;  MIN, MAX
ADRL9:      DB    14,19
ADUS13:     DB    28
...
...
...
;
;           Lista de rutine ale utilizatorului
;

TABUS:                                        ;  rutine USER
```

```
                DW          STECR1                  ;  stergere afisaj
                DW          TESTIN                  ;  teste la power on
                DW          PAFISC                  ;  pozit. ind. AFISC=1
                DW          ASTST?                  ; astept. elib tasta

                                                    ;    AVANS pe axa
                DW          TEPPNP                  ;   afisare PP sau NP
                DW          PORLEC                  ;  pornire motor

                                                    ;   lector
    ...
    ...
    ...
    ;
    ;           texte de baza

MES1:       DC          'NUMEROM 633'
MES2:       DC          'X      A  Y'
MES3:       DC          ' '
MES4:       DC          'WAIT'
MES5:       DC          'END'
MES6:       DC          'COR=       '
MES7:       DC          'A'
MES8:       DC          'M'
MES9:       DC          'NP'
MES10:      DC          'B'
MES11:      DC          'C'
MES12:      DC          'D'

    ;
    ;           descriptori taste functionale
    ;

TABTF:      DB    TESC,CESC,1,C             ;  ESC
            DB    THLP,C,C,C                ;  HELP
            DB    TJOG,CJOG,2,C             ;  JOG
            DB    TREF,CREF,3,C             ;  REF
            DB    TCOR,CCOR,4,C             ;  FANTA
    ...
            DB    0FFH                      ;  sfirsit tabel
    ...
    ;           urmeaza zona echivalari pentru coduri taste si
    ;                adrese de porturi
    ...
    ...
    ...
            end
```

ANEXA 3.4

Exemplu de modul *BIOS extins*, pentru microsisteme cu microcontroller PCB80C552, prezentat în § 6.4

```
;
;*************************************************************
;
;     Modulul BIOS cu extensie de masurari si servicii
;     pentru sistem cu PCB80C552
;
;     Data: 20.01.96
;
;     Autor: Stoicu-Tivadar Vasile
;
;*************************************************************
;
;     ***************************
;
;     Adrese de memorie
;
;     ***************************
;
adbaza          equ   0          ;   adresa de baza pag. 0
ofs_pb          equ   0          ;   offset inceput program
ofs_ex          equ   3h         ;    intrerupere externa
ofs_ct          equ   0bh        ;   ceas de timp real
ofs_us          equ   23h        ;    intrerupere unitate seriala
ofs_ad          equ   53h        ;   intrerupere CAN
ofs_bf          equ   60h        ;   tabel salturi pt. functii BIOS

;
;     Adrese RAM intern BIOS
;

;
;     Flag-uri interne utilizate
;
it_sta      data  20h            ;  ind. rutine utilizator
f_urtk      bit   it_sta.0       ;  bit user seq. RTclK.
b_urtk      bit   it_sta.1       ;   apel user rtk incep./sf. trat. it
f_uadc      bit   it_sta.2       ;  bit user ADConversion
b_uadc      bit   it_sta.3       ;   apel incep./sf. trat. it-DAC
f_uext      bit   it_sta.4       ;  bit user EXTernal int.
b_uext      bit   it_sta.5       ;   apel incep./sf. trat. it-ext
f_user      bit   it_sta.6       ;  bit user SErial Unit
b_user      bit   it_sta.7       ;   apel incep./sf. secv. trat. it-us

ap_sta      data  21h            ;  indicator aplicatie instalata
f_uapl      bit   ap_sta.0       ;  aplicatie utilizator instalata
s_uapl      bit   ap_sta.1       ;  aplicatie suspendata
i_uapi      bit   ap_sta.2       ;   aplicatia trebuie inst. la cold
                                 ;  boot

su_sta      data  22h            ; ind. starea transmisiei seriale
s_ssta      bit   su_sta.0       ;  caracter emis
s_rsta      bit   su_sta.1       ;  caracter receptionat
s_serr      bit   su_sta.2       ;  eroare la emisie
s_rerr      bit   su_sta.3       ;   eroare la receptie

tm_sta      data  23h            ; starea masurarilor de timp
c_mtim      bit   tm_sta.0       ;  cerere de masurare int. timp
```

```
c_mdel      bit   tm_sta.1          ;   cerere de secv. intirziata
c_pern      bit   tm_sta.2          ;   cerere lans. per. secv. neprior.
c_perp      bit   tm_sta.3          ;   cerere lans. per. secv. pric.


cm_sta      data  24h               ;  starea masurarilor folosind ADC
g_adco      bit   cm_sta.0          ;  conversie pornita
d_adco      bit   cm_sta.1          ;   valoare disponibila
m_adco      bit   cm_sta.2          ;  masurare pornita
f_adco      bit   cm_sta.3          ;   ciclu in curs

kt_sta      data  25h               ;  starea tastaturii
k_tsta      bit   kt_sta.0          ;   caracter disponibil
k_tflp      bit   kt_sta.1          ;   flip-flop

;
;     Notes propriu BIOS
;
;           adrese rutine user
;
a_uapl      data  28h               ;  adr. apl. pt. lansare la boot
a_urtk      data  2ah               ;  adresa rutinei user rtclk
a_uadc      data  2ch               ;  adresa rutinei user ADC
a_uext      data  2eh               ;  adresa rutinei user EXTernal int.
a_useu      data  30h               ;  adresa rutinei user SErial Unit
;
;           perioade
;
p_pern      data  32h               ; per. secv. periodica neprioritara
ct_pen      data  34h               ;   contor lans. per. neprioril.
p_perp      data  36h               ;  per. secv. per. prioritara
ct_pep      data  38h               ; contor lans. per. priorit.
p_dels      data  3ah               ;  durata secv. intirziata
;
;           adrese rutine instalate pe secvente temporizate
;
a_pern      data  3ch               ; adr.rut.user pe secv. per. neprior.
a_perp      data  3eh               ; adr. rut. user pe secv. per. pri.
a_dels      data  40h               ; adr. rut. user pe secv. intirziata
;
;           legate de masurari
;
s_adco      data  42h               ;  masca selectare canale
a_adco      data  43h               ;  canalul activ la ADC
p_adco      data  44h               ;  pointer rezultate
n_adco      data  46h               ; nr. valori de citit

;
;           Rezultate ale masuratorilor
;
r_mtim      data  48h               ;  rezultatul masurarii int. timp
r_madc      data  4ah               ;  rez. conversie

;
;
;     Zone private BIOS
;
;           unitatea seriala
;
c_rseu      data  4ch               ;  caracter rec. prin Serial Unit
c_sseu      data  4dh               ;  caracter emis prin Serial Unit
;
;           tastatura
;
k_tcod      data  4eh               ;   cod tasta apasata

;
;           stiva
;
```

```
sp_init     data  50h                ;  adresa stivei sistem
sp_user     data  60h                ; adresa stivei utilizator

;   *
;     *********************************
;     pagina 0 - vectori de intreruperi
;
;           -punere sub tensiune
;           -intrerupere de CTR
;           -intrerupere CAN
;           -intrerupere US
;     *********************************
;
      org   adbaza+ofs_pb
      ajmp  seq_pb             ;  secventa punere sub tensiune

      org   adbaza+ofs_ex
      ajmp  it_ext             ;   secventa de tratare a intr. externe

      org   adbaza+ofs_ct
      ajmp  it_rtk             ;  secventa intrerupere de ceas de timp real

      org   adbaza+ofs_us
      ajmp  it_sun             ;  secventa de intrerupere unitatea seriala

      org   adbaza+ofs_ad
      ajmp  it_adc             ; secventa de intrerupere CAN


      org   adbaza+ofs_bf

;     ************************************************
;
;     Tabel de salturi BIOS
;
;     regula : grupa - caracterizare  -  functie
;                 x            x         xxxx
;      sau     grupa - functia
;                 x     xxxxx
;
;     ***********************************************Car
sgetch:     ajmp  s_getc           ;   receptie caracter de la us - GETCHar
sgtche:     ajmp  s_gtch           ;   receptie cu ecou - GeTCHar with Echo
sptcnw:     ajmp  s_ptcn           ;  emisie fara asteptare PuTChar and
                                   ;      NoWait
sptcnv:     ajmp  s_ptcv           ; emisie fara asteptare cu verif.
                                   ;  PuTChar No wait with Verify
sgtcnw:     ajmp  s_gtcn           ;  rec. fara asteptare - GeTChar and
                                   ;     No Wait
supldh:     ajmp  s_upld           ;  scriere zona memorie in hex -
                                   ;      UPLoaD Hex
sdwldh:     ajmp  s_dwld           ;   citire hex - DoWnLoad Hex

;
;           instalari de rutine user pe intreruperi "r"
;
aplusr:     ajmp  a_user           ;  inst. aplicatie utilizator (la boot)
rrtclk:     ajmp  r_rtcl           ; inst. rut. user CTR  - Real-Time CLocK
radcon:     ajmp  r_adco           ; user CAN - Analog-Digital CONversion
rextit:     ajmp  r_exti           ; user intr. ext. - EXTernal InTerrupt
rseunt:     ajmp  r_seun           ; unit. seriala - SErial UNiT

;
;           time handling "t"
;
tstime:     ajmp  t_stim           ;  start masurare interval de timp cu
                                   ; CTR
                                   ;   Start TIme MEasurement
thtime:     ajmp  t_htim           ;  stop masurare interval de timp cu CTR
```

```
                                        ;   Halt TIme MEasurement
tidels:     ajmp  t_idel                ;  armare  secv. intirziata
                                        ; Initialize DELayed Sequence
tipern:     ajmp  t_ipen                ; inst. secv. util. cu lans. per.
                                        ;   neprioirit.
                                        ; Install PERiodical started
                                        ; Nonprioritaire   sequence
thpern:     ajmp  t_hpen                ;   dezinstal. secv. utilizator
                                        ;     periodica
                                        ;    Halt ...
tiperp:     ajmp  t_ipep                ; inst. s. util. cu lans. per. in
                                        ;  timp real
                                        ;  Install PERiodical started
                                        ; Prioritaire sequence ( real-time)
thperp:     ajmp  t_hpep                ;   dezinstall ...

;
;           masurari "m"
;
msadco:     ajmp  m_sadc                ;  selectare canal de citit si start
                                        ;   conv.
                                        ; Select Analog Digital COnversion
                                        ;    channel
mradco:     ajmp  m_radc                ;  citire valoare convertita - Read ADC
mpadco:     ajmp  m_padc                ;  calare corecta a valorii - Convert
                                        ;    ADC


;
;           tastatura  "k"
;
kbgetc:     ajmp  k_getc                ;  citire caracter tastat
kbthit:     ajmp  k_thit                ;  testare caracter tastat
kbflus:     ajmp  k_flus                ;  flush (golire) buffer

;
;           afisajul  "a"
;
afchar:     ajmp  a_putc                ;  afisare digit
afdelt:     ajmp  a_delt                ;  stergere afisaj
afpozi:     ajmp  a_pozi                ;  pozitionare cursor
afsett:     ajmp  a_sett                ;  setare atribute

;
;           EEPROM serial "e"
;
esinit:     ajmp  e_init                ;  init. canal EEPROM serial
eswrit:     ajmp  e_writ                ;  scriere EEPROM serial
esread:     ajmp  e_read                ;  citire EEPROM serial

;
;     ************************************
;     tratarile de intreruperi
;
;     ************************************

;
;     la punere sub tensiune
;
seq_pb:
      mov   A,sp_init                   ;     adresa stivei sistem
      acall icboot                      ;  initializare la rece
      jnb   i_uapl,ijmoni               ;  salt in monitor daca nu e inst.
                                        ;    aplicatie
      acall aplusr                      ;   instalare aplicatie utilizator

      mov   A,#HIGH(seq_pb)
      push  ACC                         ;   asa nu mai e nevoie de stiva
                                        ;  utilizator
```

```
        mov     A,#LOW(seq_pb)
        push    ACC
        mov     A,#0
        mov     DPH,a_uapl          ;  aici, apelul de la aplicatia user
        mov     DPL,a_uapl+1        ;
        jmp     @A+DPTR             ;  apelul propriu-zis

;
;       secventa de tratare a intreruperii de ceas de timp real
;       foloseste Register Bank 1
;
it_rtk:
        push    PSW
        clr     PSW.4
        setb    PSW.3               ; selectare Register Bank 1
        mov     R3,SP               ;  salvarea adr. stivei
        dec     R3                  ;  (cu corectia de rigoare datorita
        push    ACC                 ;      primului push)
        push    DPH                 ;   salvari in
        push    DPL                 ; continuare
;
;       user sequence la inceput
;
        jnb     f_urtk,it_rk1       ; nu se trateaza user seq.
        jnb     b_urtk,it_rk1       ;  nu la inceput e apelul
        mov     A,#HIGH(it_rk1)
        push    ACC
        mov     A,#LOW(it_rk1)
        push    ACC
        mov     A,#0
        mov     DPH,a_urtk          ;  aici, apelul de la a_urtk
        mov     DPL,a_urtk+1        ;    (user sequence)
        jmp     @A+DPTR             ;  apelul propriu-zis
it_rk1:
;
;       secventa de tratare diverse aspecte de timp
;
;             masurare interval
;
        jnb     c_mtim,it_rk2       ;   nu trebuie masurare
        clr     C
        mov     A,r_mtim
        add     A,#1                ;  incrementarea
        mov     r_mtim,A            ;   contorului
        mov     A,r_mtim+1          ;   cu 1
        addc    A,#0                ;      pe 2
        mov     r_mtim+1,A          ;       octeti
it_rk2:
;
;             secventa intirziata
;
        jnb     c_mdel,it_rk3       ;trebuie secv. intirziata ? nu, salt peste...
        clr     C
        mov     A,p_dels
        subb    A,#1                ;  decrementarea
        mov     p_dels,a
        mov     A,p_dels+1          ;   contorului pentru
        subb    A,#0                ;     secventa
        mov     p_dels+1,A          ;       intirziata
        jnc     it_rk3              ;   nu s-a scurs inca timpul
        mov     A,SP                ;  adr. actuala a stivei
        subb    A,R3                ;   determinarea lung. stivei ocupate
        mov     R4,a                ;   acesta va fi contorul de transfer
        mov     R1,SP               ;  sursa transfer
        inc     SP                  ;    adr. stivei
        inc     SP                  ;      va fi deplasata cu 2
        mov     R0,SP               ;   dest. transfer
l_t1:
        mov     A,@R1
```

```
        mov   @R0,A             ;  transfer
        dec   R1
        dec   R0                ;  si decrementare
        djnz  R4,l_t1           ;  reluare pina s-a realizat tot transferul
        mov   A,R3              ; adr. unde trebuie inserare
        mov   R0,A
        mov   A,a_dels
        mov   @R0,A             ;  inserarea
        inc   R0
        mov   A,a_dels+1        ;   adresei
        mov   @R0,A             ;       rutinei
        inc   R3                ; adr. de inceput a stivei creste
        inc   R3                ;  datorita incarcarii cu adr. subrut. user
        clr   c_mdel            ;  stergerea cererii de lans. secv. intirz.

it_rk3:
;
;           lansare periodica prioritara

        jnb   c_perp,it_rk4     ;trebuie secv. intirziata ? nu, salt peste...
        clr   C
        mov   A,ct_pep
        subb  A,#1              ;  decrementarea
        mov   ct_pep,A
        mov   A,ct_pep+1        ;   contorului pentru
        subb  A,#0              ;     secventa
        mov   ct_pep+1,A        ;      intirziata
        jnc   it_rk4            ;   nu s-a scurs inca timpul
        mov   ct_pep,p_perp     ;  refacere contor
        mov   ct_pep+1,p_perp+1
        mov   A,#HIGH(it_rk4)
        push  ACC
        mov   A,#LOW(it_rk4)    ;  se forteaza in stiva adresa de retur
        push  ACC
        mov   A,#0
        mov   DPH,a_perp        ;  aici, apelul de la a_perp
        mov   DPL,a_perp+1      ;    (user sequence)
        jmp   @A+DPTR           ;  apelul propriu-zis
it_rk4:
;
;           lansare periodica neprioritara
;
        jnb   c_pern,it_rk5     ;trebuie secv. intirziata ? nu, salt peste...
        clr   C
        mov   A,ct_pen
        subb  A,#1              ;  decrementarea
        mov   ct_pen,a
        mov   A,ct_pen+1        ;   contorului pentru
        subb  A,#0              ;     secventa
        mov   ct_pen+1,A        ;      intirziata
        jnc   it_rk5            ;   nu s-a scurs inca timpul
        mov   ct_pen,p_pern     ; refacere contor
        mov   ct_pen+1,p_pern+1
        mov   A,SP              ;  adr. actuala a stivei
        subb  A,R3              ;   determinarea lung. stivei ocupate
        mov   R4,A              ;   acesta va fi contorul de transfer
        mov   R1,SP             ;  sursa transfer
        inc   SP                ;    adr. stivei
        inc   SP                ;      va fi deplasata cu 2
        mov   R0,SP             ;   dest. transfer
l_t2:
        mov   A,@R1
        mov   @R0,A             ;  transfer
        dec   R1
        dec   R0                ;  si decrementare
        djnz  R4,l_t2           ;  reluare pina s-a realizat tot transferul
        mov   A,R3              ; adr. unde trebuie inserare
        mov   R0,A
        mov   A,a_pern
```

```
        mov   @R0,A             ;  inserarea
        inc   R0
        mov   A,a_perm+1
        mov   @R0,A             ;    adresei rutinei
it_rk5:
;
;           aici se insereaza rutina de tratare TASTATURA
;

;
;           aici se insereaza rutina de tratare AFISAJ
;

;
;     user sequence la sfirsit
;
        jnb   f_urtk,it_rk9     ; nu se trateaza user seq.
        jb    b_urtk,it_rk9     ; nu la sfirsit e apelul
        mov   A,#HIGH(it_rk9)
        push  ACC
        mov   A,#LOW(it_rk9)
        push  ACC
        mov   A,#0
        mov   DPH,a_urtk        ;  aici, apelul de la a_urtk
        mov   DPL,a_urtk+1      ;   (user sequence)
        jmp   @A+DPTR           ;  apelul propriu-zis

it_rk9:
                                ; refaceri
        pop   DPL
        pop   DPH
        pop   ACC
        pop   PSW               ;   se reface si Register Bank initial
        reti                    ;   retur

;
;     secventa de tratare a intreruperii de CAN
;
it_adc:
        push  ACC
        push  PSW
        push  DPH               ;   salvari
        push  DPL

        jnb   f_uadc,it_ad1     ; nu se trateaza user seq.
        jnb   b_uadc,it_ad1     ;  nu la inceput e apelul
        mov   A,#HIGH(it_ad1)
        push  ACC
        mov   A,#LOW(it_ad1)
        push  ACC
        mov   A,#0
        mov   DPH,a_uadc        ;  aici, apelul de la a_uadc
        mov   DPL,a_uadc+1      ;    (user sequence)
        jmp   @A+DPTR           ;  apelul propriu-zis
it_ad1:
;
;     aici este partea specifica convertorului
;     si achizitiei implementate
;
        jnb   f_uadc,it_ad9     ; nu se trateaza user seq.
        jb    b_uadc,it_ad9     ; nu la sfirsit e apelul
        mov   A,#HIGH(it_ad9)
        push  ACC
        mov   A,#LOW(it_ad9)
        push  ACC
        mov   A,#0
        mov   DPH,a_uadc        ;  aici, apelul de la a_uadc
        mov   DPL,a_uadc+1      ;   (user sequence)
        jmp   @A+DPTR           ;  apelul propriu-zis
```

```
it_ad9:
                                ; refaceri
        pop     DPL
        pop     DPH
        pop     PSW
        pop     ACC
        reti                    ;   retur

;
;       secventa de tratare a intreruperii de la unitatea seriala

it_sun:
        push    ACC
        push    PSW
        push    DPH             ;   salvari
        push    DPL

        jnb     f_useu,it_su1   ; nu se trateaza user seq.
        jnb     b_useu,it_su1   ;  nu la inceput e apelul
        mov     A,#HIGH(it_su1)
        push    ACC
        mov     A,#LOW(it_su1)
        push    ACC
        mov     A,#0
        mov     DPH,a_useu      ;  aici, apelul de la a_useu
        mov     DPL,a_useu+1    ;    (user sequence)
        jmp     @A+DPTR         ;  apelul propriu-zis
it_su1:
        jb      SCON.0,it_su2   ;  aici, test conditie receptie
        mov     A,c_sseu        ;  caracterul care trebuie emis
        mov     SBUF,a          ;  este scris in buffer disponibil
        setb    s_ssta          ;  si se indica emisia caracterului
        clr     SCON.1          ;   apoi se reseteaza bitul de emisie
        sjmp    it_su3          ;  continuare
it_su2:
        mov     a,SBUF          ;caracterul receptionat este citit din buffer
        mov     c_rseu,A        ;  si este retinut
        setb    s_rsta          ;  apoi se indica receptia caracterului
        clr     SCON.0          ;   si se reseteaza bitul de receptie
it_su3:
        jnb     f_useu,it_su9   ; nu se trateaza user seq.
        jb      b_useu,it_su9   ; nu la sfirsit e apelul
        mov     A,#HIGH(it_su9)
        push    ACC
        mov     A,#LOW(it_su9)
        push    ACC
        mov     A,#0
        mov     DPH,a_useu      ;  aici, apelul de la a_useu
        mov     DPL,a_useu+1    ;   (user sequence)
        jmp     @A+DPTR         ;  apelul propriu-zis

it_su9:
                                ; refaceri
        pop     DPL
        pop     DPH
        pop     PSW
        pop     ACC
        reti                    ;   retur

;
;       secventa de tratare intrerupere externa
;
it_ext:
        push    ACC
        push    PSW
        push    DPH             ;   salvari
```

```
        push    DPL
        jnb     f_uext,it_ex1       ; nu se trateaza user seq.
        jnb     b_uext,it_ex1       ;  nu la inceput e apelul
        mov     A,#HIGH(it_ex1)
        push    ACC
        mov     A,#LOW(it_ex1)
        push    ACC
        mov     A,#0
        mov     DPH,a_uext          ;  aici, apelul de la a_uext
        mov     DPL,a_uext+1        ;    (user sequence)
        jmp     @A+DPTR             ;  apelul propriu-zis
it_ex1:
;
;       aici se insereaza secventa specifica pentru it-ext
;
        jnb     f_uext,it_ex9       ; nu se trateaza user seq.
        jb      b_uext,it_ex9       ; nu la sfirsit e apelul
        mov     A,#HIGH(it_ex9)
        push    ACC
        mov     A,#LOW(it_ex9)
        push    ACC
        mov     A,#0
        mov     DPH,a_uext          ;  aici, apelul de la a_uext
        mov     DPL,a_uext+1        ;    (user sequence)
        jmp     @A+DPTR             ;  apelul propriu-zis

it_ex9:
                                    ; refaceri
        pop     DPL
        pop     DPH
        pop     PSW
        pop     ACC
        reti                        ;    retur

;
;       ********************************************
;       Functii BIOS
;
;
;       ********************************************
;
;
;
;       Functii de initializare
;
;
i_cboo:                             ;  initializare la rece - Cold BOOT


        acall   esinit              ;    initializare canal EEPROM serial
        mov     it_sta,#0
        mov     tm_sta,#0
        mov     cm_sta,#0           ; initializari pentru flaguri interne
        mov     kt_sta,#0
        setb    s_ssta
        setb    s_rsta              ;  setari pentru transmisia libera
        clr     s_serr
        clr     s_rerr              ;  si resetari conditii de eroare transmisie

                                    ; in continuare se executa functia warm boot
                                    ;  deci nu mai trebuie ret

i_wboo:                             ;  initializare la cald - Warm BOOT
;
;       aici se insereaza initializari in functie de secventele specifice
;       care nu sint inca prezente in sursa
;

        ret
```

```
i_jmon:                         ;  salt in monitorul serial - Jump to MONitor
      acall i_wboo              ; initializare la cald
      ajmp        monit         ; salt in monitor
;     ret

;
;     Functii ale unitatii seriale
;
s_putc:                         ;   emisie caracter prin us - PUTCHar - car.
                                ;    in ACC
      jnb   s_ssta,s_putc       ;  polling pe cond. buffer disponibil
      clr   s_ssta              ;  apoi se indica transmisie ocupata din nou,
      mov   c_sseu,A            ;  se "trimite" caracterul
      ret                       ;   si gata, retur

s_getc:                         ;   receptie caracter de la us - GETCHar
      jnb   s_rsta,s_getc       ;  polling pe car. disponibil
      mov   A,c_rseu            ;  retinerea caracterului receptionat
      clr   s_rsta              ;   se indica buffer eliberat
      ret

s_gtch:                         ;   receptie cu ecou - GeTCHar with Echo
      acall s_getc              ;   receptie
      ajmp  s_putc              ;     si trimitere in ecou

s_ptcn:                         ;  emisie fara polling
      mov   c_sseu,A            ;  inserare caracter in buffer intermediar
      ret                       ;    si asta e tot

s_ptcv:                         ;  emisie fara polling cu verificare
      jnb   s_ssta,s_ptcl       ;  daca nu e buffer disponibil, se marcheaza
      clr   s_ssta              ;  marcare transmisie ocupata
      mov   c_sseu,A            ;  inserare car. in buffer intermediar,
      clr   C
      ret                       ;    si retur
s_ptcl:                         ;  secv. de retur pentru transmisie ocupata
      setb  C                   ; marcare conditie de transmisie ocupata
      ret                       ;   si gata

s_gtcn:                         ;  rec. fara polling - GeTChar and No Wait
      jnb   s_rsta,s_gtcl       ;  daca nu e caracter disponibil, retur
      mov   A,c_rseu            ;  retinerea eventualului caracter
      clr   s_rsta              ;   apoi, se indica buffer eliberat
      clr   C
      ret                       ;    si gata
s_gtcl:                         ; aici, retur pe conditie de lipsa car. rec.
      setb  C                   ;      marcare conditie de lipsa car.
      ret                       ;   si retur

s_upld:                         ;  scriere zona memorie in hex - UPLoaD Hex
;
;     aici se insereaza rutina de scriere in HEX
;;

      ret

s_dwld:                         ;   citire hex - DoWnLoad Hex
;
;     aici se insereaza rutina de citire HEX
;

      ret


;
;     Instalari de rutine user pe intreruperi
;     intrari:
;           - in R0, R1 - adr rutinei
```

```
;           - in acc - <> 0 - lansare la inceput secv.
;                     -  = 0 - lansare la sfirsit secv.
;
;
;
a_user:                         ; (nu are param. in ACC. !)
      mov   a_uapl,R0
      mov   a_uapl+1,R1         ;  adr. aplicatie
      setb  f_uapl              ;  marcare apl. user
      setb  s_uapl              ;  marcare aplicatie activa
      ret                       ; si gata


r_rtcl:                         ; inst. rut. user CTR  - Real-Time CLocK
      setb  b_urtk
      cjne  A,#0,r_rtl          ;  init. bit lansare inceput/sf. secv. user
      clr   b_urtk
r_rtl:
      mov   a_urtk,R0
      mov   a_urtk+1,R1         ;  init. adresa subrutinei utilizator
      setb  f_urtk              ;  si in final cererea de lansare
      ret

r_adco:                         ; user CAN - Analog-Digital CONversion
      setb  b_uadc
      cjne  A,#0,r_adl          ;  init. bit lansare inceput/sf. secv. user
      clr   b_uadc
r_adl:
      mov   a_uadc,R0
      mov   a_uadc+1,R1         ;  init. adresa subrutinei utilizator
      setb  f_uadc              ;  si in final cererea de lansare

      ret

r_exti:                         ; user intr. ext. - EXTernal InTerrupt
      setb  b_uext
      cjne  A,#0,r_exl          ;  init. bit lansare inceput/sf. secv. user
      clr   b_uext
r_exl:
      mov   a_uext,R0
      mov   a_uext+1,R1         ;  init. adresa subrutinei utilizator
      setb  f_uext              ;  si in final cererea de lansare

      ret

r_seun:                         ; unit. seriala - SErial UNiT
      setb  b_useu
      cjne  A,#0,r_sel          ;  init. bit lansare inceput/sf. secv. user
      clr   b_useu
r_sel:
      mov   a_useu,R0
      mov   a_useu+1,R1         ;  init. adresa subrutinei utilizator
      setb  f_useu              ;  si in final cererea de lansare

      ret

;
;     Time handling
;
t_stim:                         ;  start masurare interval de timp cu CTR
                                ;   Start TIme MEasurement
                                ;  iesire cy=1 daca mas. e ocupata
      jb    c_mtim,t_st1        ; salt la masurare ocupata
      mov   r_mtim,#0
      mov   r_mtim+1,#0         ; anulare contor
      setb  c_mtim              ; start masurare
      clr   C                   ;  cy=0 (retur normal)
      ret
t_st1:                          ;  retur pe resursa ocupata
```

```
        setb  C                 ;  cy=1 adica masurare neterminata (eroare)
        ret                     ; retur

t_htim:                         ;  stop masurare interval de timp cu CTR
                                ;   Halt TIme MEasurement
                                ;iesiri : interval masurat in R0, R1
                                ;   cy=1 daca eroare(timerul nu a fost
                                ;    pornit)
        jnb   c_mtim,t_ht1      ; test eroare
        mov   R0,r_mtim
        mov   R1,r_mtim+1       ;  preluarea valorii de timp
        clr   c_mtim            ;   si oprire masurare
        clr   C                 ;  cod retur normal
        ret
t_ht1:                          ;  retur pe cod de eroare
        setb  C                 ; cy=1 adica masurare nepornita
        ret


t_idel:                         ;  armare  secv. intirziata
                                ; Initialize DELayed Sequence
                                ;  intrare in R0, R1 - adr. rutinei
                                ;             R2, R3 - valoarea temporizarii
                                ;  iesire pe cY=1 daca deja e instalata secv.
        jb    c_mdel,t_id1      ;  salt la tratare eroare
        mov   a_dels,R0
        mov   a_dels+1,R1       ;  init. adr. subrutina secv. intirziata
        mov   p_dels,R2
        mov   p_dels+1,R3       ;  init. perioada intirziere
        setb  c_mdel            ;  si lansare secventa intirziata
        clr   C                 ;  retur normal
        ret
t_id1:                          ;  tratare eroare
        setb  C                 ;  iesire cY=1 daca deja e inst. secv. si e
        ret                     ;     in derulare


t_ipen:                         ; inst. secv. util. cu lans. per. neprioirit.
                                ; Install PERiodinal started Nonprioritaire
                                ;   sequence
                                ;  intrari : in R0, R1 - adr. subrutina
                                ;            in R2, R3 - val. perioada
                                ;  iesire:  cY=1 - eroare, secv. deja inst.
        jb    c_pern,t_in1      ;  tratare eroare
        mov   a_pern,R0
        mov   a_pern+1,R1       ; adr. rutina user
        mov   p_pern,R2
        mov   ct_pen,R2
        mov   p_pern+1,R3       ;  val. perioada
        mov   ct_pen+1,R4
        setb  c_pern            ;  setare lans. periodica
        clr   C                 ;  retur normal
        ret                     ; (cy = 0)
t_in1:
        setb  C                 ;  indicare eroare
        ret                     ;  retur pe eroare (cY=1)

t_hpen:                         ;   dezinstal. secv. utilizator periodica
                                ;    Halt ...
                                ; iesire cy=1 pe eroare (serv. inca nesolic.)
        jnb   c_pern,t_hn1      ; eroare
        clr   c_pern            ; oprire lansare temporizata
        clr   C                 ;  retur normal
        ret
t_hn1:
        setb  C                 ; eroare cy=1 serviciu nepornit
        ret

t_ipep:                         ; inst. s. util. cu lans. per. in timp real
```

```
                                ;  Install PERiodical started Prioritaire
                                ; sequence ( real-time)
                                ;  intrari : in R0, R1 - adr. subrutina
                                ;            in R2, R3 - val. perioada
                                ;  iesire:  cY=1 - eroare, secv. deja inst.
        jb      c_perp,t_ipl    ;  tratare eroare
        mov     a_perp,R0
        mov     a_perp+1,R1     ; adr. rutina user
        mov     p_perp,R2
        mov     ct_pep,R2
        mov     p_perp+1,R3     ;  val. perioada
        mov     ct_pep+1,R4
        setb    c_perp          ;  setare lans. periodica
        clr     C               ;  retur normal
        ret                     ; (cy = 0)
t_ipl:
        setb    C               ;  indicare eroare
        ret


t_hpep:                         ;   dezinstall secv. utilizator per. priorit.
                                ; iesire:
                                ;   cy=1 pt.eroare pe solicitare
                                ;      in afara secventei
        jnb     c_perp,t_hpl    ; test eroare
        clr     c_perp          ; oprire secventa
        clr     C               ; retur normal
        ret
t_hpl:                          ; secv. de eroare
        setb    C               ; eroare out of sequence
        ret                     ;  retur


;
;       Masurari
;
m_sadc:                         ;  selectare canal de citit si start conv.
                                ; Select Analog Digital COnversion channel
                                ;   intrare - in ACC - canalul de selectat
;
;       urmeaza o secventa specifica hardware-lui disponibil
;

        ret

m_radc:                         ;  citire valoare convertita - Read ADCO
;
;       urmeaza o secventa specifica hardware-lui disponibil
;

        ret

m_padc:                         ;  calare corecta a valorii - Process ADCO
;
;       urmeaza o secventa specifica hardware-lui disponibil
;

        ret

;
;       Functii de intrare/iesire clasice
;
;             intrari (tastatura)
;
k_getc:                         ;  citire caracter tastat
        jnb     k_tsta,k_getc   ; polling pina la caracter tastat
        mov     A,k_tcod        ;  extragerea codului tastei
        clr     k_tsta          ;  si indicare cod extras
        ret
```

```
k_thit:                         ;  testare caracter tastat
      jnb   k_tsta,k_thi1       ;   nu e caracter tastat
      mov   A,k_tcod            ;  extragere cod,
      clr   k_tsta              ;    indicare cod extras si
      clr   C                   ;    indicare retur avand caracter
      ret                       ;      si retur
k_thi1:                         ; aici, nu e caracter,
      setb  C                   ;    ceea ce trebuie marcat
      ret                       ;    apoi, retur

k_flus:                         ;  flush (golire) buffer
      clr   k_tsta              ;  se indica buffer gol
      ret                       ;    si retur


;
;           iesiri (afisajul)
;
a_putc:                         ;  afisare digit
;
;     urmeaza o secventa specifica hardware-lui disponibil
;

      ret

a_delt:                         ;  stergere afisaj
;
;     urmeaza o secventa specifica hardware-lui disponibil
;

      ret

a_pozi:                         ;  pozitionare cursor
;
;     urmeaza o secventa specifica hardware-lui disponibil
;

      ret

a_sett:                         ;  setare atribute
;
;     urmeaza o secventa specifica hardware-lui disponibil
;

      ret

;
;     intrari/iesiri non-standard
;     EEPROM serial
;
e_init:                         ;  initializare
;
;     urmeaza o secventa specifica hardware-lui disponibil
;

      ret

e_writ:                         ;  scriere
;
;     urmeaza o secventa specifica hardware-lui disponibil
;

      ret

e_read:                         ; citire
;
;     urmeaza o secventa specifica hardware-lui disponibil
;
```

```
        ret

;
;       ***********************************************
;       Monitor serial
;
;       ***********************************************
;
monit:
;
;       aici se va insera un monitor serial
;

        end
```

ANEXA 4
Programul *Instrument Designer*

Clase folosite

Prezentarea din această anexă provine din lucrarea [Rac96].

După cum s-a precizat la subcapitolul 8.2, programul **Instrument Designer** foloseşte clase derivate din clasele predefinite **Object Windows**. Ierarhia claselor folosite a fost prezentată în Figura 8.8.

Toate clasele **Object Windows** se derivă dintr-o clasă de bază numită *TModule*, care este o clasă generică şi realizează alocarea de resurse pentru clasele **Object Windows**.

Clasa *TApplication* derivă din *TModule*, şi este clasa care construieşte aplicaţia **Windows**. Iniţializează o instanţă a programului şi instalează modulul de gestiune a mesajelor.

↳ Din *TApplication* se derivă clasa principală a programului **Instrument Designer**, **TCASEApp**. Localizat în fişierul **case.cpp**, clasa *TCASEApp* are următoarea declaraţie:

```
class _CLASSTYPE TCASEApp : public TApplication
{
public:
   TCASEApp(LPSTR name, HINSTANCE hInstance, HINSTANCE hPrevInstance,
            LPSTR lpCmd, int nCmdShow) : TApplication(name, hInstance,
            hPrevInstance, lpCmd, nCmdShow) {};
   virtual void InitMainWindow();
   virtual void InitInstance();
   virtual void IdleAction();
};
```

Constructorul clasei apelează constructorul clasei de bază *TApplication* fără a efectua iniţializări proprii.

Funcţiile membre ale clasei sunt:

* ***InitMainWindow*** - creează fereastra principală a aplicaţiei prin crearea unui obiect *TCASEWindow*, care este clasa ferestrei principale. Pointerul la această clasă este stocat în variabila membru *MainWindow*.

* ***InitInstance*** - încarcă tabela de acceleratori folosită de program. Tabela de acceleratori defineşte un set de *hot-keys* cu ajutorul cărora se pot da comenzile din meniu prin nişte combinaţii de taste.

* ***IdleAction*** - este apelată cînd programul este în aşteptarea unui mesaj, şi nu execută nici o altă sarcină. Se foloseşte pentru a prelua mesajele din bufferul alocat portului serial şi a transmite la fereastra TTY când aceasta este activă.

Din *TObject* se derivă clasa *TWindowsObject* care defineşte un element *interface* vizibil (fereastră, meniu, buton, bară de rulare etc). Aceste elemente *interface* vizibile sunt derivate mai departe în clasele *TWindow* şi *TDialog*.

Clasa *TWindow* reprezintă o fereastră generică, cu cadru, bară de titlu, butoane de dimensionare şi meniu sistem, care poate fi redimensionată şi mutată. Această clasă este clasa de bază pentru clasa *TMDIFrame*, clasa ferestrei unei aplicaţii multidocument, care are grijă de crearea, iniţializarea şi gestiunea ferestrelor fiu (*child*) din cadrul unei aplicaţii multidocument.

↳ Fereastra principală a programului, **TCASEWindow** este un descendent al clasei *TMDIFrame*. Se prezintă sub următoarea formă:

```
class _CLASSTYPE TCASEWindow : public TMDIFrame
{
   PTToolBar    ToolBar;
   int             nIBarHeight;
   BOOL         fAssemble, fLink, fConvert, fBuild;

   void SetupWindow();
   virtual void Paint(HDC, PAINTSTRUCT _FAR &);
   BOOL CanClose();
   virtual void WMSize(RTMessage)               = [WM_FIRST + WM_SIZE];
   void HandleToolBarMsg(RTMessage)             = [WM_TOOLBAR];
   void HandleActivateAppMsg(RTMessage Msg)= [WM_FIRST + WM_ACTIVATEAPP];
   void HandleNewFileMsg(RTMessage)             = [CM_FIRST + CM_MDIFILENEW];
   void HandleOpenFileMsg(RTMessage)            = [CM_FIRST + CM_MDIFILEOPEN];
   void HandleDeleteFileMsg(RTMessage)   = [CM_FIRST + CM_DELETEFILE];
   void HandleAssembleMsg(RTMessage)            = [CM_FIRST + CM_ASSEMBLE];
   void HandleLinkMsg(RTMessage)                = [CM_FIRST + CM_LINK];
   void HandleConvertMsg(RTMessage)             = [CM_FIRST + CM_CONVERT];
   void HandleBuildMsg(RTMessage)               = [CM_FIRST + CM_BUILD];
   void HandleUserCommandMsg(RTMessage)  = [CM_FIRST + CM_USERCOMMAND];
   void HandleExpertSysMsg(RTMessage)    = [CM_FIRST + CM_EXPERTSYS];
   void HandleDictionaryMsg(RTMessage)   = [CM_FIRST + CM_DICTIONARY];
   void HandleOpenLinkMsg(RTMessage)            = [CM_FIRST + CM_OPENLINK];
   void HandleCloseLinkMsg(RTMessage)    = [CM_FIRST + CM_CLOSELINK];
   void HandleIdentifyMsg(RTMessage)            = [CM_FIRST + CM_IDENTIFY];
   void HandleOpenTTYMsg(RTMessage)             = [CM_FIRST + CM_OPENTTY];
   void HandleLoadMsg(RTMessage)                = [CM_FIRST + CM_LOAD];
   void HandleMasterMsg(RTMessage)              = [CM_FIRST + CM_MASTER];
   void HandleSetBuildMsg(RTMessage)            = [CM_FIRST + CM_SETBUILD];
   void HandleSetSerialLinkMsg(RTMessage)       = [CM_FIRST + CM_SETSERIALLINK];
   void HandleSetTTYMsg(RTMessage)              = [CM_FIRST + CM_SETTTY];
   void HandleOpenSettingsMsg(RTMessage) = [CM_FIRST + CM_OPENSETTINGS];
   void HandleAboutMsg(RTMessage)               = [CM_FIRST + CM_ABOUT];
public:
   WORD  ChildNum;
   TCASEWindow(LPSTR, LPSTR);
};
```

Declaraţia clasei *TCASEWindow* este destul de voluminoasă pentru că trebuie să răspundă la comenzile venite de la meniul programului şi de la *toolbar*. Funcţiile care încep cu cuvântul *"Handle"* răspund la câte un mesaj, care este determinat de *indexul de distribuire*. Un caz special este funcţia ***WMSize***, care răspunde la mesajul de redimen-sionare, trimis de **Windows.** Această funcţie există şi în clasa de bază *TMDIFrame*, aici este redefinită. Pentru că este vorba de obiecte dinamice (alocate în timpul execuţiei) şi funcţia *WMSize* este apelată din afara clasei (deci nu de o funcţie membru a clasei) pe baza unui pointer, nu se ştie la compilare care funcţie *WMSize* va fi apelată de Windows: cea a clasei de bază, sau a clasei *TCASEWindows*. De acea funcţia este declarată ca fiind o **funcţie virtuală.** Asta înseamnă că abia în timpul execuţiei se va decide care funcţie *WMSize* va fi apelată. **Borland** C++ rezolvă acest lucru cu **tabele de funcţii virtuale.**

Variabila membru ***nTBarHeight*** păstrează valoarea lățimii *toolbar*-ului, iar ***fAssemble, fLink, fConvert, fBuild*** sunt niște indicatori care sunt activi dacă este în curs un proces de asamblare, linkeditare, sau conversie.

Constructorul acestei clase inițializează *toolbar*-ul creând un obiect *interface* de clasă *TToolBar*, care va fi atribuită variabilei membre *ToolBar*, și o variabilă care arată numărul ferestrelor *child* actuale.

Următoarele trei funcții sunt prezente la aproape toate clasele din program:

* ***SetupWindow*** - inițializează fereastra creată. Înainte de a face inițializările ferestrei derivate, trebuie apelată funcția *SetupWindow* din clasa de bază. În clasa *TCASEWindow* funcția aceasta afișează *toolbar*-ul, și citește configurația programului apelând funcția publică ***ReadConfig***.

* ***Paint*** - este funcția responsabilă de desenarea conținutului ferestrei. Mai întâi trebuie apelată funcția *Paint* a clasei de bază. Desenarea se va face utilizând **contextul dispozitivului** primit ca parametru (*PaintDC*), care este un descriptor al ferestrei pe care se face desenarea. În clasa *TCASEWindow* această funcție doar apelează funcția *Paint* a clasei de bază *TMDIFrame*.

* ***CanClose*** - este funcția apelată la închiderea ferestrei. Fereastra poate fi închisă numai dacă *CanClose* ***returnează*** TRUE. În clasa *TCASEWindow* această funcție execută salvarea configurației.

Celelalte funcții din această clasă sunt :

* ***HandleToolbarMsg*** - răspunde la mesajele trimise de *toolbar*. Primește ca parametru numărul de ordine al butonului apăsat de pe *toolbar*.

* ***HandleActivateAppMsg*** - deoarece acest program implementează un mediu integrat de dezvoltare cu scule de dezvoltare de programe, separate de mediu (asambloare, linkeditoare, convertoare, aplicații utilizator), este necesară apelarea acestor programe externe. Deoarece **Windows** este un sistem de operare multitasking, scula încărcată nu se execută până când programul nu intră în bucla de așteptare de mesaje, și dat fiind faptul că programul trebuie să execute încă niște operații după terminarea rulării sculei respective (de exemplu încărcare fișierului .LST după asamblare), trebuie așteptat mesajul care semnalizează reactivarea programului, cea ce înseamnă terminarea rulării sculei. Acest mesaj (*WM_ACTIVATEAPP*) este receptat de această funcție care execută funcțiile cerute după rularea sculei.

* ***HandleNewFileMsg*** - deschide o nouă fereastră de editare, creând un obiect de clasei *TCASEFileWindow*.

* ***HandleOpenFileMsg*** - afișează un dialog de selecție a numelui fișierului, și deschide o nouă fereastră de editare, încărcând fișierul selectat.

* ***HandleDeleteFileMsg*** - șterge un fișier care este selectat dintr-un dialog de selecție fișier.

* ***HandleAssembleMsg*** - lansează asamblorul, transmițându-i ca parametru numele fișierului curent editat.

* ***HandleLinkMsg*** - lansează linkeditorul, transmițându-i ca parametru numele fișierului curent editat.

* ***HandleConvertMsg*** - lansează convertorul de cod, transmițându-i ca parametru numele fișierului curent editat.

* ***HandleBuildMsg*** - lansează succesiv cele trei scule amintite mai sus, realizând astfel obținerea codului executabil convertit într-un singur pas din punctul de vedere al utilizatorului.

* ***HandleUserCommandMsg*** - lansează un program definit de utilizator, transmițându-i ca parametru numele fișierului curent editat.

* ***HandleExpertSysMsg*** - deschide fereastra de navigare *hypertext* prin crearea unui obiect de clasă *TExpSysWin*.

* ***HandleDictionaryMsg*** - deschide dialogul de gestiune a bazei software prin crearea unui obiect de clasă *TDictionay*.

* ***HandleOpenLinkMsg*** - deschide legătura serială între calculator și microsistem alocând memorie pentru bufferele de intrare-ieșire și setează portul serial la parametrii specificați în fișierul de configurare. Dacă este nevoie, derulează dialogul de identificare.

* ***HandleCloseLinkMsg*** - închide legătura serială prin eliberarea memoriei pentru bufferele de comunicație.

* ***HandleIdentifyMsg*** - execută dialogul de identificare a plăcii prin transmisia și recepția mesajelor de identificare specificate în fișierul de configurație.

* ***HandleOpenTTYMsg*** - deschide fereastra **TTY** prin crearea unui obiect de clasă *TTTYWindow*.

* ***HandleLoadMsg*** - transferă un fișier **HEX** selectat dintr-un dialog de selecție fișiere la microsistemul - țintă.

* ***HandleMasterMsg*** - lansează o aplicație specificată și activează monitorizarea mesajelor prin portului serial.

* ***HandleSetBuildMsg*** - deschide un dialog de setare a sculelor și a parametrilor procesului de asamblare - linkeditare - conversie prin crearea unui obiect de clasă *TSetTools*.

* ***HandleSetSerialLinkMsg*** - deschide un dialog de setare a parametrilor comunicației seriale precum și a specificațiilor de identificare a microsistemului prin crearea unui obiect de clasă *TSetSerialLink*.

* ***HandleSetTTYMsg*** - deschide un dialog de setare a parametrilor de funcționare a ferestrei **TTY** prin crearea unui obiect de clasă *TSetTTY*.

* ***HandleOpenSettingsMsg*** - permite alegerea și deschiderea unui fișier de configurare, afișând un dialog de selecție fișiere

* ***HandleAboutMsg*** - afișează fereastra de informații despre program.

A doua clasă ce se derivă din *TWindowsObject* este *TDialog*. Reprezintă obiectul *interface* pentru ferestre de dialog. Fereastra de dialog este o fereastră cu controale (butoane, căsuțe de marcare, linii de editare, etc) în care utilizatorul setează diferite opțiuni, și apoi fereastra se închide. Aceste ferestre de dialog se numesc *modale*, și nu permit altă activitate înainte de închidere. Există și ferestre de dialog *nemodale* care pot să rămână active și în timpul altor activități.

↳ Clasa **TToolBar** reprezintă *toolbar*-ul ferestrei principale. Este o fereastră de dialog nemodală, derivată din *TDialog*, care nu are bară de titlu, nu poate fi mutată și redimensionată.

```
class _CLASSTYPE TToolBar : public TDialog
{
   void HandleFileNewMsg(RTMessage)  = [ID_FIRST + TB_FILENEW];
   void HandleFileOpenMsg(RTMessage) = [ID_FIRST + TB_FILEOPEN];
   void HandleFileSaveMsg(RTMessage) = [ID_FIRST + TB_FILESAVE];
```

```
    void HandleCutMsg(RTMessage)      = [ID_FIRST + TB_CUT];
  public:
    TToolBar(PTWindowsObject AParent, LPSTR AName)
                : TDialog(AParent, AName) {};
};
```

Declarația clasei conține constructorul ***TToolBar***, care apelează doar constructorul clasei *TDialog*, și o serie de funcții pentru fiecare buton al *toolbar*-ului. De exemplu ***HandleFileOpenMsg*** răspunde la mesajul venit de la butonul de deschidere fișier, cu identificatorul *TB_FILEOPEN*. Fiecare funcție trimite un mesaj *WM_TOOLBAR* ferestrei principale cu un parametru ce conține identificatorul butonului respectiv.

Din *TWindow* se derivă clasa *TEditWindow*, care implementează o fereastră de editare, cu facilități de editare text și operații cu *Clipboard*.

Clasa *TFileWindow* este o fereastră pentru editare de fișiere, și are ca și clasă de bază *TEditWindow*. Pe lîngă proprietățile de editare moștenite de la clasa de bază, clasa *TFileWindow* mai posedă funcții de creare, deschidere-încărcare și salvare de fișier.

↳ Clasa **TCASEFileWindow** este un descendent al clasei *TFileWindow*. Este obiectul *interface* pentru ferestrele de editare fișier.

```
class _CLASSTYPE TCASEFileWindow : public TFileWindow
{
  virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);
 public:
  TCASEFileWindow(PTWindowsObject AParent, LPSTR ATitle, LPSTR AFileName)
                        : TFileWindow(AParent, ATitle, AFileName) {};
  virtual LPSTR GetClassName();
};
```

Cele două funcții membre ale acestei clase vor mai apărea și în altele, tot cu aceleași funcții:

* ***GetWindowClass*** - setează informațiile legate de *clasa ferestrei*, cum ar fi **stilul ferestrei** (aliniamentul, dacă trebuie redesenat după redimensionare, dacă este permisă închiderea ferestrei, etc.), **icoana**, **meniul**, **cursorul ferestrei, culoarea fundalului**. Cuvântul *"clasă"* aici nu semnifică o clasă C++, ci un grup de atribute care descriu comportamentul ferestrelor care au aceeași clasă.

* ***GetClassName*** - returnează numele clasei ferestrei. Clasa ferestrei are un nume, prin care **Windows** deosebește și ține în evidență clasele.

Prezența acestor două funcții într-o clasă (C++) nu este obligatorie. În clasa *TCASEFileWindow* se folosesc pentru a defini un icon pentru ferestrele de editare.

↳ Clasa **TExpSysWindow** - este derivată din *TWindow* și reprezintă fereastra de navigare *hypertext*. Navigarea se realizează cu ajutorul *hyperlink*-urilor, care determină încărcarea paginii specificate de *hyperlink*. Fereastra de navigare poate avea cîmpuri de editare, căsuțe de marcare, precum și imagini, care sunt încărcate dintr-un fișier **DLL** (*Dynamic Link Library*) asociat fișierului *hypertext*. Generarea programului pe baza navigării se face apelând o funcție din fișierul **DLL**.

```
class _CLASSTYPE TExpSysWindow : public TWindow
{
  typedef struct tagLink
  {
```

```
    RECT rect;
    char Topic[50];
  } Link;
  typedef struct tagVariable
  {
    char Name[50];
    char Value[50];
  } Variable;
  typedef struct tagInput
  {
    int    VarIndex;
    PTEdit Editor;
  } Input;
  typedef struct tagCheck
  {
    int         VarIndex;
    PTCheckBox  CheckBox;
  } Check;
  PTExpSysButtons ToolBar;
  HINSTANCE    hDLL;
  HFONT        hNormalText, hBoldText, hItalicText, hLinkText;
  HCURSOR      hHandCursor;
  HFILE        hHDXFile;
  Variable     Variables[100];
  Link            Links[50];
  Input        Inputs[50];
  Check        Checks[50];
  char            History[100][50];
 char          *pBuff, *pBeginTopic, *pCurrPos;
  char            szCurrentTopic[50];
  int          cVariable, cLink, cInput, cCheck, cHistory;
  int          nLink, nYChar;
  BOOL            fMouseOnLink;

  void SetupWindow();
  BOOL CanClose();
  virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);
  virtual LPSTR GetClassName();
  virtual void Paint(HDC, PAINTSTRUCT _FAR &);
  void WMMouseMove(RTMessage)                              = [WM_FIRST + WM_MOUSEMOVE];
  void WMSetCursor(RTMessage)                              = [WM_FIRST + WM_SETCURSOR];
  virtual void WMLButtonDown(RTMessage)      = [WM_FIRST + WM_LBUTTONDOWN];
  void HandleButtonsMsg(RTMessage)           = [WM_EXPSYSBUTTONS];
  BOOL GoToTopic(char *szTopic);
  int  GetText(char *pText, BOOL *fCommand);
  int  GetVarIndex(char *szVarName);
  HBITMAP (FAR *GetBitmap)(LPSTR szName);
public:
  TExpSysWindow(PTWindowsObject, LPSTR);
};
```

Variabilele membre din această clasă au următoarea semnificație:

* structura ***Link*** - definește un *hyperlink*. Poziția ocupată în fereastră este păstrat de structura ***rect***, care conține coordonatele stânga-sus și dreapta-jos. Numele paginii la care conduce *hyperlink*-ul este stocat în câmpul ***Topic***.

* structura ***Variable*** - definește o variabilă în *hypertext*, care este folosită de câmpurile de introducere date și căsuțele de marcare. Câmpul ***Name*** conține numele, iar câmpul ***Value*** valoarea.

* structura ***Input*** - definește câmpul de introducere date. Conține un obiect de clasă *TEdit* în câmpul ***Editor***, care este linia de editare propriu-zisă, și un index pentru tabloul variabilelor (***Variables***) pentru a arăta care este variabila asociată liniei de editare, în câmpul ***VarIndex***.

* structura ***Check*** - defineşte o căsuţă de marcare. Obiectul de clasă *TCheckBox*, ce reprezintă căsuţa de marcare propriu zisă este conţinut de câmpul ***CheckBox***, iar indexul de variabilă în ***VarIndex***.

* ***ToolBar*** - este obiectul de clasă *TExpSysButtons*, ce reprezintă butoanele (*Back, History, Generate, Cancel*) ataşate ferestrei de navigare.

* ***hDLL*** - este instanţa modulului **DLL** încărcat.

* ***hNormalText***, ***hBoldText***, ***hItalicText***, ***hLinkText*** - sunt descriptorii fonturilor folosite în fereastra de navigare.

* ***hHandCursor*** - descriptorul cursorului "arătător" folosit când cursorul se află deasupra unui *hyperlink*.

* ***hHTXFile*** - descriptorul fişierului *hypertext* .**HTX**.

* ***Variables***, ***Links***, ***Input***, ***Checks*** - sunt tablourile ce conţin variabilele, *hyperlink*-urile, câmpurile de introducere date şi căsuţele de marcare.

* ***History*** - este tabloul care conţine drumul parcurs la navigare, adică numele paginilor parcurse

* ***pBuff***, ***pBeginTopic***, ***pCurrPos*** - sunt pointeri folosiţi la manipularea bufferului de bază software.

* ***szCurrentTopic*** - numele paginii curente.

* ***cVariable***, ***cLink***, ***cInput***, ***cCheck***, ***cHistory*** - arată numărul de elemente în tablourile corespunzătoare.

* ***nLink*** - este numărul *hyperlink*-ului de pe o pagină care a fost selectat în timpul navigării.

* ***nYChar*** - este înălţimea caracterelor.

* ***fMouseOnLink*** - este **TRUE** dacă cursorul *mouse*-ului se află deasupra unui *hyperlink*.

Prezentăm în continuare funcţiile membre ale clasei *TExpSysWindow*:

Constructorul clasei crează fonturile folosite şi butoanele (*toolbar*-ul) ferestrei de navigare.

* ***SetupWindow*** - încarcă fişierul **HTX** într-un buffer pointat de **pBuff*, şi fişierul **DLL** asociat. Dezactivează meniul *"Expert System"* pentru a nu permite deschiderea unei noi ferestre de navigare.

* ***CanClose*** - eliberează bufferul şi activează meniul.

* ***WMMouseMove*** - este funcţia care interceptează mişcarea *mouse*-ului. Dacă cursorul ajunge deasupra unui *hyperlink*, setează indicatorul *fMouseOnLink*.

* ***WMSetCursor*** - setează cursorul *mouse*-ului la forma de "mână" dacă *fMouseOnLink* = **TRUE**.

* ***WMLButtonDown*** - această funcţie este apelată de Windows când a fost apăsat butonul din stânga al mouse-ului. Dacă este setat *fMouseOnLink* face salt la pagina care este conţinut în elementul din tabloul *Links* cu indexul egal cu numărul *hyperlink*-ului din pagină

* ***HandleButtonsMsg*** - răspunde la mesajele butoanelor ferestrei de navigare.

* ***GoToTopic*** - se poziţionează la o pagină anume.

* ***GetText*** - returnează următoarea porțiune de text, care poate să fie o comandă de formatare sau un text care trebuie afișat.

* ***GetVarIndex*** - returnează indexul variabilei al cărei nume este furnizat ca parametru. Dacă variabila nu există, creează una nouă.

* ***GetBitmap*** - este pointerul la funcția din **DLL** care afișează un *bitmap*.

↳ Clasa **TExpSysButtons** este *toolbar*-ul ferestrei de navigare. Este derivat din *TDialog*, fiind o fereastră de dialog nemodală. Conține butoanele *"Back"*, *"History"*, *"Generate"* și *"Cancel"*.

```
class _CLASSTYPE TExpSysButtons : public TDialog
{
   void HandleBackMsg(RTMessage)     = [ID_FIRST + ES_BACK];
   void HandleHistoryMsg(RTMessage)  = [ID_FIRST + ES_HISTORY];
   void HandleGenerateMsg(RTMessage) = [ID_FIRST + ES_GENERATE];
   void HandleCancelMsg(RTMessage)   = [ID_FIRST + ES_CANCEL];
 public:
   TExpSysButtons(PTWindowsObject AParent, LPSTR AName)
                        : TDialog(AParent, AName) {};
};
```

Are ca funcții membre funcții care răspund la apăsarea butoanelor și trimit mesajul *WM_EXPSYSBUTTONS* ferestrei de navigare, având ca parametru identificatorul butonului respectiv.

↳ Clasa **TDictionary** este clasa pentru fereastra de dialog a sistemului de gestiune a bazei software. Este un descendent a clasei *TDialog* și conține o fereastră de listare pentru denumirile modulelor din baza software, un câmp pentru descrierea modulelor și butoane pentru gestiune. Este declarată în felul următor:

```
class _CLASSTYPE TDictionary : public TDialog
{
   PTListBox    Titles;
   PTStatic     Description;
   int          hDBFFile;
   char            *pItems;
   WORD            fLength;

   void SetupWindow();
   BOOL CanClose();
   char *GetTitle(int pos);
   char *GetDescription(int pos);
   char *GetRoutine(int pos);
   void ClearBufferEnd();
   void HandleListMsg(RTMessage)   = [ID_FIRST + DC_TITLES];
   void HandleCopyMsg(RTMessage)   = [ID_FIRST + DC_COPY];
   void HandleEditMsg(RTMessage)   = [ID_FIRST + DC_EDIT];
   void HandleInsertMsg(RTMessage) = [ID_FIRST + DC_INSERT];
   void HandleDeleteMsg(RTMessage) = [ID_FIRST + DC_DELETE];
 public:
   TDictionary(PTWindowsObject, LPSTR);
   char *GetCurrentTitle();
   char *GetCurrentDescription();
   char *GetCurrentRoutine();
   void ChangeCurrentTitle(char *newtitle);
   void ChangeCurrentDescription(char *newdescr);
   void ChangeCurrentRoutine(char *newroutine);
};
```

Clasa are următoarele variabile membre:

* ***Titles*** - este obiectul de clasă *TListBox* care reprezintă fereastra de listare a denumirilor modulelor din baza software.

* ***Description*** - este un obiect de clasă *TStatic* şi este câmpul de afişare a descrierii modulului tocmai selectat din baza software.

* ***hDBFFile*** - este descriptorul fişierului **.DBF** a bazei software.

* ***pItems*** - pointează la bufferul unde este încărcată baza software în memorie.

* ***fLength*** - lungimea fişierului **.DBF**.

Funcţiile membre sunt următoarele:

* ***Constructorul*** clasei creează fereastra de listare şi câmpul de descriere.

* ***SetupWindow*** - încarcă fişierul **.DBF** într-un buffer care va fi pointat de *pItems* şi introduce titlurile modulelor în fereastra de listare

* ***CanClose*** - actualizează fişierul **.DBF.**

* ***GetTitle*** - returnează denumirea modulului al cărui index este furnizat ca parametru.

* ***GetDescription*** - returnează descrierea modulului al cărui index este furnizat ca parametru.

* ***GetRoutine*** - returnează modulul al cărui index este furnizat ca parametru.

* ***ClearBufferEnd*** - umple bufferul cu "00h" unde nu este ocupat de module.

* ***HandleListMsg*** - tratează mesajul de selecţie a denumirii de la fereastra de listare

* ***HandleCopyMsg*** - se apelează la apăsarea butonului *"Copy"* şi copiază modulul curent în *Clipboard*.

* ***HandleEditMsg*** - se apelează la apăsarea butonului *"Edit"* şi lansează dialogul de editare a denumirii şi descrierii modulului curent, care este un obiect de clasă *TDictEdit*.

* ***HandleInsertMsg*** - se apelează la apăsarea butonului *"Insert"* şi inserează un nou modul în baza software. Modulul inserat va avea denumirea *"NEW"*, descrierea *"No description"*, iar modulul respectiv va conţine textul *"No routine"*.

* ***HandleDeleteMsg*** - se apelează la apăsarea butonului *"Delete"* şi şterge modulul curent din baza software.

* ***GetCurrentTitle*** - returnează denumirea modulului curent.

* ***GetCurrentDescription*** - returnează descrierea modulului curent.

* ***GetCurrentRoutine*** - returnează modulul curent.

* ***ChangeCurrentTitle*** - schimbă denumirea modulului curent cu denumirea furnizată ca parametru.

* ***ChangeCurrentDescription*** - schimbă descrierea modulului curent cu descrierea furnizată ca parametru.

* ***ChangeCurrentRoutine*** - schimbă modulul curent cu modulul furnizat ca parametru.

↳ Clasa **TDictEdit** derivată din *TDialog* este obiectul *interface* pentru fereastra de dialog pentru editarea a denumirii şi descrierii modulului. Este creat şi activat de funcţia *HandleEditMsg* din clasa *TDictionary*, şi conţine câmpuri de editare pentru denumire şi descriere de module precum şi butoanele *"OK"*, *"Cancel"* şi *"Edit routine"*.

Declararea clasei este .

```
class _CLASSTYPE TDictEdit : public TDialog
{
   PTEdit Title;
   PTEdit Description;

   void SetupWindow();
   BOOL CanClose();
   void HandleEditRoutineMsg(RTMessage) = [ID_FIRST + DC_EDITROUTINE];
 public:
   TDictEdit(PTWindowsObject, LPSTR);
};
```

Variabilele membre ale clase sunt:

* ***Title*** - este obiectul de clasă *TEdit* pentru linia de editare a denumirii modulului.
* ***Description*** - este obiectul *TEdit* pentru câmpul de editare a descrierii modulului.

Funcţiile membre:

* ***Constructorul*** creează editoarele pentru denumire şi descriere.
* ***SetupWindow*** - iniţializează editoarele cu valorile din buffer.
* ***CanClose*** - actualizează bufferul cu noile date introduse.
* ***HandleEditRoutine*** - răspunde la butonul *"Edit routine"* şi lansează dialogul de editare a modulului, care este un obiect de clasă *TDictEditRoutine*.

↳ Clasa **TDictEditRoutine** reprezintă fereastra de dialog pentru editarea modulului. Are ca clasă de bază clasa *TDialog* şi conţine un câmp de editare şi butoanele *"OK"* şi *"Cancel"*.

Clasa este declarată sub următoarea formă:

```
class _CLASSTYPE TDictEditRoutine : public TDialog
{
   PTEdit Routine;

   void SetupWindow();
   BOOL CanClose();
public:
   TDictEditRoutine(PTWindowsObject, LPSTR);
};
```

Clasa are o singură variabilă membru, ***Routine***, care un obiect de clasă *TEdit* şi reprezintă câmpul de editare pentru editarea modulului.

Funcţiile membre au aceeaşi sarcină ca la clasa *TDictEdit*.

↳ Clasa **TTTYWindow** este clasa pentru fereastra **TTY**. Este o fereastră care afişează tot ce vine de la microsistem, şi trimite la microsistemul-ţintă tot ce se tastează.

```
class _CLASSTYPE TTYWindow : public TWindow
{
   int  nXChar, nYChar, nXPos, nYPos, nRows, nCols;
   char *pScrBuff, *pScrPos;
   RECT rectClear;

   void SetupWindow();
   BOOL CanClose();
   virtual void GetWindowClass(WNDCLASS _FAR & AWndClass);
   virtual LPSTR GetClassName();
   virtual void WMSize(RTMessage) = [WM_FIRST + WM_SIZE];
   void WMSetFocus(RTMessage)     = [WM_FIRST + WM_SETFOCUS];
   void WMKillFocus(RTMessage)    = [WM_FIRST + WM_KILLFOCUS];
   virtual void Paint(HDC, PAINTSTRUCT _FAR &);
   void DisplayStr(char *pLine, BOOL fLocalEcho);
   void HandleKeypressMsg(RTMessage) = [WM_FIRST + WM_CHAR];
   void HandleCommReceiveMsg(RTMessage)     = [WM_COMMRECEIVE];
 public:
   TTYWindow(PTWindowsObject, LPSTR)
};
```

Variabilele membre ale clasei sunt:

* ***nXChar, nYChar*** - lățimea respectiv lungimea unui caret.
* ***nXPos, nYPos*** - poziția curentă a caracterului în fereastră (x,y).
* ***nRows, nCols*** - numărul de rânduri respectiv coloane din fereastra TTY.
* ***pScrBuff*** - pointer la bufferul în care sunt stocate caracterele afișate în scopul redesenării ferestrei la redimensionare.
* ***pScrPos*** - pointer la poziția curentă în buffer.
* ***rectClear*** - dreptunghiul care trebuie șters după defilarea în sus a conținutului ferestrei.

Funcțiile membre sunt:

* ***Constructorul*** - setează fontul pentru fereastra **TTY** și inițializează variabilele *nXChar, nYChar, nXPos, nYPos*.
* ***SetupWindow*** - crează bufferul pentru caractere și pointerii pentru manipularea acestui buffer și dezactivează meniul *"Open TTY"* pentru a nu permite deschiderea unei noi ferestre **TTY**.
* ***CanClose*** - eliberează bufferul și activează meniul.
* ***WMSize*** - răspunde la mesajul de redimensionare a ferestrei. Redesenează fereastra pe baza bufferului de caractere și actualizează variabilele *nRows, nCols*.
* ***WMSetFocus*** - este apelată când fereastra este activată. Se folosește pentru afișarea caretului.
* ***WMKillFocus*** - este apelată când fereastra este dezactivată. Se folosește pentru ștergerea caretului.
* ***DisplayStr*** - afișează un șir de caractere la poziția curentă.
* ***HandleKeypressMsg*** - răspunde la apăsarea unei taste, prin trimiterea caracterului corespunzător tastei apăsate la portul serial.
* ***HandleCommReceiveMsg*** - răspunde la mesajul *WM_COMMRECEIVE* primit de la funcția *IdleAction* al clasei *TCASEApp* care citește portul serial și trimite la fereastra **TTY** acest

mesaj, dacă au sosit caractere de la microsistem. Apelează funcţia *DisplayStr* pentru a afişa caracterele.

Următoarele clase sunt obiectele *interface* pentru ferestrele de dialog pentru diverse setări: a sculelor de dezvoltare program, a dicţionarului, a sistemului expert, a legăturii seriale, a ferestrei **TTY**, etc. În aceste clase, funcţia membru ***SetupWindow*** creează obiectele pentru controalele din fereastra de dialog (butoane, linii de editare, căsuţe de marcare, ferestre de listare, etc) şi iniţializează valorile şi stările controalelor conform variabilelor de configurare. Funcţia ***CanClose*** salvează în variabilele de configurare valorile şi stările controalelor.

↳ Clasa **TSetTools** este clasa ferestrei de dialog pentru setarea parametrilor sculelor de dezvoltare. Este declarată în felul următor:

```
class _CLASSTYPE TSetTools : public TDialog
{
   PTEdit          Assembler, Linker, Converter, UserCommand;
   PTCheckBox      DisplayCmdLine, ShowLst;

   void SetupWindow();
   BOOL CanClose();
   void HandleBrowseAssemblerMsg(RTMessage) = [ID_FIRST + TL_BROWSSEMBLER];
   void HandleBrowseLinkerMsg(RTMessage)    = [ID_FIRST + TL_BROWLINKER];
   void HandleBrowseConverterMsg(RTMessage) = [ID_FIRST + TL_BROWCONVERTER];
   void HandleBrowseUserCommandMsg(RTMessage)= [ID_FIRST + TL_BROWUSERCMD];
   void HandleSpecificationsMsg(RTMessage)  = [ID_FIRST + TL_SPECIFICATIONS];
 public:
   TSetTools(PTWindowsObject, LPSTR);
};
```

Variabilele membre sunt:

* ***Assembler***, ***Linker***, ***Converter***, ***UserCommand*** - sunt obiecte de clasă *TEdit* şi servesc la introducerea denumirii şi căii sculelor de dezvoltare.

* ***DisplayCmdLine***, ***ShowLst*** - sunt obiecte de clasă *TBCheckBox* şi reprezintă căsuţele de marcare pentru selecţia posibilităţii de afişare a liniei de comandă înainte de rularea sculei, respectiv a încărcării fişierului **.LST** după asamblare.

Funcţiile membre sunt următoarele :

* Primele patru funcţii membre care încep cu *"Handle"* răspund la apăsarea butoanelor *"Browse"*, care sunt puse lângă fiecare linie de editare, şi lansează câte o fereastră de selecţie fişier. Rezultatul selecţiei este introdus în linia de editare corespunzătoare.

* ***HandleSpecificationsMsg*** - lansează fereastra de dialog de clasă *TSetSLSpecs* pentru setarea parametrilor liniilor de comandă ale sculelor.

↳ Clasa **TSetTLSpecs** este clasa ferestrei de dialog pentru setările parametrilor liniilor de comandă.

```
class _CLASSTYPE TSetTLSpecs : public TDialog
{
   PTEdit AsmCmd, LinkCmd, ConvCmd, UserCmd;

   void SetupWindow();
   BOOL CanClose();
```

```
  public:
    TSetILSpecs(PTWindowsObject, LPSTR);
};
```

Variabilele membre sunt :

* ***AsmCmd***, ***LinkCmd***, ***ConvCmd***, ***UserCmd*** - care sunt obiectele liniilor de editare pentru introducerea parametrilor liniilor de comandă ale asamblorului, linkeditorului, convertorului şi a comenzii (programului) utilizator.

↳ Clasa **TSetSerialLink** este clasa ferestrei de dialog pentru setările parametrilor comunicaţiei seriale.

```
class _CLASSTYPE TSetSerialLink : public TDialog
{
   PTComboBox Port, BaudRate, Parity, FlowCtrl, DataBits, StopBits;
   PTCheckBox AskForId;

   void SetupWindow();
   BOOL CanClose();
   void HandleSpecificationsMsg(RTMessage) = [ID_FIRST + SL_SPECIFICATIONS];
 public:
   TSetSerialLink(PTWindowsObject, LPSTR);
};
```

Variabile membre sunt :

* ***Port***, ***BaudRate***, ***Parity***, ***FlowCtrl***, ***DataBits***, ***StopBits*** - sunt obiecte de tip *TComboBox*, şi reprezint ferestrele de listare de tip *drop-down* pentru setarea portului de comunicaţie, a ratei de transfer, a paritãţii, a controlului de transfer, numărului de biţi de dată şi a biţi de stop.

* ***AskForId*** - reprezintă o căsuţă de marcare de tip *TBCheckBox* prin care se specifică dacă se doreşte identificarea microsistemului la deschiderea legăturii seriale.

Funcţiile membre sunt :

* ***HandleSpecificationsMsg*** - răspunde la mesajul de apăsare a butonului *"Identification"* şi deschide o fereastră de dialog de clasă *TSetSLSpecs* pentru specificarea dialogului de identificare a microsistemului.

↳ Clasa **TSetSLSpecs** este clasa ferestrei de dialog pentru specificarea dialogului de identificare a microsistemului.

```
class _CLASSTYPE TSetSLSpecs : public TDialog
{
  PTComboBox PassNr, CurrPass;
  PTEdit Send, Receive;
  PTRadioButton SendStr, SendNr, SendNothing, RecStr, RecNr, RecIgnore;
  int nCurrIndex;

  void SetupWindow();
  BOOL CanClose();
  void HandlePassNrMsg(RTMessage)   = [ID_FIRST + SL_PASSNR];
```

```
    void HandleCurrPassMsg(RTMessage) = [ID_FIRST + SL_CURRPASS];
    void SetValues(int);
    void GetValues(int);
  public:
    TSetSLSpecs(PTWindowsObject, LPSTR);
};
```

Variabile membre sunt :

* ***PassNr***, ***CurrPass*** - sunt obiecte de clasă *TComboBox* şi servesc pentru selectarea numărului de etape de identificare, respectiv a etapei curente pentru care se specifică parametri.

* ***Send***, ***Receive*** - sunt liniile de editare de clasă *TEdit* în care se introduc valorile care trebuiesc transmise respectiv receptionate pe parcursul unei etape de identificare.

* ***SendStr***, ***SendNr***, ***SendNothing*** - sunt obiecte de clasă *TBRadioButton* şi reprezintă "butoanele radio" pentru selectarea tipului de dată transmisă pe parcursul unei etape de identificare: şir de caractere, numere separate prin virgulă sau nu se transmite nimic.

* ***RecStr***, ***RecNr***, ***RecIgnore*** - sunt obiecte de clasă *TBRadioButton* şi reprezintă "butoanele radio" pentru selectarea tipului de dată receptionată pe parcursul unei etape de identificare: şir de caractere, numere separate prin virgulă sau caracterele transmise sunt ignorate.

* ***nCurrIndex*** - etapa curentă.

Funcţii membre sunt :

* ***HandlePassNrMsg*** - răspunde la mesajul de selectare a numărului de etape de la prima fereastră de listare *drop-down*.

* ***HandleCurrPassMsg*** - răspunde la mesajul de selectare a etapei curente de la a doua fereastră de listare *drop-down*.

* ***SetValues*** - salvează parametrii etapei curente de identificare în variabilele de configurare.

* ***GetValues*** - setează parametrii etapei curente de identificare din variabilele de configurare.

Clasa **TSetTTY** este obiectul pentru fereastra de dialog de setare a parametrilor ferestrei **TTY:**

```
class _CLASSTYPE TSetTTY : public TDialog
{
  PTBCheckBox LocalEcho, CRLFIn, CRLFOut, ToUpper;

  void SetupWindow();
  BOOL CanClose();
public:
  TSetTTY(PTWindowsObject, LPSTR);
};
```

Variabile membre sunt :

* ***LocalEcho***, ***CRLFIn***, ***CRLFOut***, ***ToUpper*** - sunt obiecte de clasă *TBCheckBox* şi reprezintă căsuţele de marcare pentru selecţia posibilităţii de a avea ecou local în fereastra **TTY**,

de a transforma caracterele **CR** recepționate în secvență **CR-LF**, de a transforma caracterele **CR** tastate în secvență **CR-LF** și de a transforma caracterele mici în caractere mari.

Variabilele publice sunt declarate în fișierul **classdef.cpp** cu cuvântul cheie **extern** pentru ca fiecare fișier sursă să aibă acces la ele și sunt definite în fișierul **case.cpp** . Variabilele care încep cu ***"cfg"*** sunt variabile de configurare. Acestea conțin toate setările din fișierul de configurare al programului și sunt inițializate de funcția *ReadConfig*.
Variabile și funcții publice sunt următoarele :

* ***HomeDir*** - directorul în care se află programul.
* ***ConfigFile*** - numele fișierului de configurare.
* ***idComDev*** - identificatorul portului serial. Este folosit de funcțiile de comunicație.
* ***pReceiveBuff*** - bufferul de recepție de la portul serial. În acest buffer scrie funcția *IdleAction* a clasei *TCASEApp* caracterele sosite prin portul serial și din acest buffer citește funcția *HandleComReceiveMsg* a clasei *TTYWindow*.
* ***fTTYOpen*** - este **TRUE** dacă este deschisă fereastra **TTY.** Se folosește pentru a închide fereastra **TTY** dacă se închide legătura serială.
* ***HTTYWindow*** - este descriptorul ferestrei **TTY.** Este folosit pentru a trimite mesaje la fereastra **TTY.**

Funcțiile publice sunt următoarele:

* ***ReadConfig*** - citește fișierul de configurare și încarcă setările în variabilele de configurare.
* ***WriteConfig*** - scrie valorile variabilelor de configurare în fișierul de configurare.
* ***HandleComError*** - dacă s-a ivit o eroare de comunicație afișează un mesaj de eroare.
* ***WaitForTransmit*** - așteaptă până când bufferul de emisie se golește.
* ***WaitForReceive*** - așteaptă recepția unui număr de caractere precizat ca parametru.

Resurse folosite

Resursele unui program **Windows** sunt șabloanele (*templates*) pentru meniuri, ferestre de dialog, bitmap-uri, icoane, etc. Aceste resurse sunt prezente într-un fișier separat **.RC** care este inclus în *project*, compilat și adăugat programului executabil la linkeditarea acestuia din urmă.

Instrument Designer are ca fișier de resurse fișierul **case.rc**. Resursele programului sunt următoarele:

* Meniul **MAIN** - este meniul principal al programului.
* Acceleratorii **COMMANDS** - sunt o colecție de "scurtături" (*shortcuts*) adică o combinație de taste prin care se poate ajunge mai repede la o comandă.
* Icoana **CASE** - este icoana programului.
* Icoana **FILEWIN** - este icoana ferestrei de editare.

* Icoana **TTY** - este icoana ferestrei **TTY**.

* Cursorul **HAND** - este un cursor de *mouse* sub formă de mână care apare când mouse-ul ajunge deasupra unui *hyperlink* în fereastra "*Expert System*".

* Bitmapurile **1101 - 1114** sunt imaginile butoanelor de pe *toolbar* în stare normală.

* Bitmapurile **3101 - 3114** sunt imaginile butoanelor de pe *toolbar* în stare apăsată.

* Dialogul **ABOUT** - este şablonul ferestrei de dialog care se afişează la comanda *Help About*, şi conţine informaţii despre program.

* Dialogul **DICTIONARY** - este şablonul ferestrei de dialog a bazei software.

* Dialogul **EDIT_DICT** - este şablonul ferestrei de dialog pentru editarea denumirii şi descrierii unui modul din baza software.

* Dialogul **EDIT_ROUTINE** - este şablonul ferestrei de dialog pentru editarea unui modul din baza software.

* Dialogul **EXPSYS_BUTTONS** - este şablonul ferestrei de dialog pentru butoanele ferestrei de "*Expert System*".

* Dialogul **SET_BUILD** - este şablonul ferestrei de dialog pentru setările sculelor de dezvoltare.

* Dialogul **SET_SERIALLINK** - este şablonul ferestrei de dialog pentru setările parametrilor comunicaţiei seriale.

* Dialogul **SET_SLSPECS** - este şablonul ferestrei de dialog pentru setările dialogului de identificare a microsistemului.

* Dialogul **SET_TLSPECS** - este şablonul ferestrei de dialog pentru setarea parametrilor liniilor de comandă a sculelor de dezvoltare.

* Dialogul **SET_TTY** - este şablonul ferestrei de dialog pentru setările parametrilor ferestrei TTY.

* Dialogul **TOOLBAR** - este şablonul ferestrei de dialog care reprezintă *toolbar*-ul programului.

* Dialogul **SD_FILEOPEN** - este şablonul ferestrei de dialog pentru deschidere fişier.

* Dialogul **SD_FILESAVE** - este şablonul ferestrei de dialog pentru salvare fişier.

* Dialogul **SD_INPUTDIALOG** - este şablonul ferestrei de dialog pentru o linie de introducere date.

* Dialogul **SD_SEARCH** - este şablonul ferestrei de dialog pentru căutare cuvinte în fereastra de editare.

* Dialogul **SD_REPLACE** - este şablonul ferestrei de dialog pentru căutare şi înlocuire cuvinte în fereastra de editare.

Aceste resurse au fost create cu **Resource Workshop** din pachetul **Borland C++ 3.1**.

CURRICULUM VITAE

Numele şi prenumele : Stoicu-Tivadar Vasile

Data şi locul naşterii: 29 septembrie 1957, Săcuieni, jud. Bihor

Studii : Facultatea de Electrotehnică Timişoara, secţia Automatizări şi Calculatoare, specializarea *"Conducerea proceselor cu calculatorul"*, promoţia 1983, media de absolvire **9,96** (*şef de promoţie, diplomă de merit*).

Locuri de muncă : **inginer proiectant** la într. "*Electromotor*" Timişoara, atelierul de "Roboţi industriali" (1983-1985), **cercetător ştiinţific** la *Institutul de Proiectări pentru Automatizări* Filiala Timişoara (1985-1989), **asistent** la *Catedra de Electrotehnică* a Facultăţii de Electrotehnică Timişoara (1989-1991), **şef de lucrări** la *Catedra (din 1996 Departamentul) de Automatică şi Informatică Industrială* a Facultăţii de Automatică şi Calculatoare a Universităţii Tehnice din Timişoara (1991-).

Stare civilă : *căsătorit* cu *Lăcrămioara Stoicu-Tivadar* (şef de lucrări la Departamentul A.I.I. al Facultăţii de Automatică şi Calculatoare din Timişoara), fără copii

Limbi străine cunoscute : *franceza, engleza, maghiara*

Activitatea ştiinţifică :

În perioada activităţii de la într. "Electromotor" a contribuit la elaborarea de *software pentru roboţi industriali*, partea de *programe de mişcare*. A realizat de asemenea programe de *proiectare a algoritmilor de reglare de poziţie*.

În cadrul activităţii de la IPA Filiala Timişoara a lucrat în cadrul unor colective de proiectare pentru *aplicaţii de telemecanică şi comandă numerică*, la unele proiecte fiind coordonator.

Ca şi cadru didactic, abordează **cercetări în domeniul** *reutilizării software, generatoarelor de aplicaţii, instrumentelor CASE, programelor de conducere de proces şi instrumentaţie*

A elaborat (singur sau în colaborare) un număr de **15** lucrări publicate în *periodice ştiinţifice sau în volumele unor sesiuni internaţionale de comunicări ştiinţifice*, a participat cu **18** lucrări la *sesiuni de comunicări ştiinţifice naţionale*, publicate în volumele acestora, a prezentat la diferite sesiuni de comunicări un număr de **21** lucrări ştiinţifice *nepublicate* şi a colaborat la **9** *contracte de cercetare ştiinţifică* (la unele dintre acestea, în calitate de responsabil).

Activitatea didactică : titular al cursurilor de *"Ingineria programării aplicaţiilor de conducere de proces"*, *"Sisteme informatice de măsură"* şi *"Periferice şi achiziţia inteligentă a datelor"*, conducător de proiecte de diplomă din domeniul său de cercetare, un îndrumător sub tipar, două cursuri şi un îndrumător în curs de elaborare .