DEZVOLTAREA ORIENTATĂ SPRE OBIECTE A PROGRAMELOR

dr. ing. Ştefan Trăuşan-Matu

Institutul de Cercetări în Informatică

Rezumat: În lucrare sunt prezentate conceptele de bază ale abordării orientate spre obiecte în dezvoltarea programelor. Sunt discutate avantajele acestei abordări din punctul de vedere al ingineriei programării şi al teoriei limbajelor de programare. Sunt analizate pe scurt cîteva din cele mai cunoscute metodologii de dezvoltare orientată spre obiecte a programelor.

Cuvinte cheie: Ingineria programării, limbaje de programare, programare orientată spre obiecte, metodologii de dezvoltare a programelor.

1. Introducere

În ultimii ani, în ingineria programării s-a impus o nouă paradigmă: abordarea orientată spre obiecte a programelor (OO), cu componentele sale: analiza orientată spre obiecte, proiectarea orientată spre obiecte, programarea orientată spre obiecte (OOP) etc. Această paradigmă are rădăcini ramificate către mai multe domenii cum ar fi: ingineria programării, modelarea, teoria limbajelor de programare, inteligenţa artificială şi teoria cunoaşterii. Abordarea OO este o speranţă de reducere şi, eventual, chiar de depăşire a problemelor dezvoltării programelor.

Pentru înţelegerea specificului şi a avantajelor abordării OO este necesar în primul rînd să se facă o analiză a problemelor care duc la dificultatea dezvoltării programelor şi la căile posibile de rezolvare, acesta fiind şi scopul secţiunii a 2-a din lucrare. În urma acestei analize au rezultat ca principale cauze complexitatea cu totul deosebită a dezvoltării programelor şi faptul că programele sunt în permanenţă supuse schimbării. În consecinţă, căile de urmat sunt direcţionate către stăpînirea acestor două aspecte. După cum se va vedea în secţiunea a 3-a, abordarea OO este un progres evident în aceste direcţii faţă de abordările tradiţionale.

Secţiunile următoare ale lucrării prezintă particularităţile OOP şi ale metodologiilor OO de dezvoltare a programelor. În acest sens, abordarea OO este o continuare firească a cercetărilor în ingineria programării şi în teoria limbajelor de programare, ea oferind soluţii noi care sporesc, atât potenţialul de creştere a productivităţii muncii de programare, cât şi puterea de expresie a limbajelor de programare.

În lucrare nu au fost incluse două domenii care sunt, de asemenea, în mari transformări în prezent ca urmare a impactului abordării OO: sistemele de gestiune a bazelor de date şi programarea concurentă. Există intense cercetări, în prezent, pentru elaborarea de sisteme de gestiune a bazelor de date, orientate spre obiecte. Aceasta abordare are avantaje evidente cum ar fi posibilitatea utilizării unor modele conceptuale evoluate, naturaleţea reprezentării structurării datelor etc. În domeniul programării concurente, obiectele sunt, de asemenea, o reprezentare naturală pentru o mulţime de agenţi care cooperează în vederea atingerii unui scop.

2. Probleme in dezvoltarea programelor. Cauze şi căi de rezolvare

După cum se ştie, progresul sub toate aspectele (viteză, memorie, cost etc) al echipamentelor de calcul este caracterizat de o rată cu totul deosebită. După cum remarca F. P. Brooks [4], nici o altă tehnologie n-a avut o creştere de şase ori în 30 de ani a raportului performanţă/preţ. Din păcate, progresul în productivitatea dezvoltării programelor a fost cu mult mai redus (aproximativ 5% pe an [6]). Programarea a rămas în mare măsură o activitate artizanală, care depinde foarte mult de talentul şi de experienţa specialiştilor implicaţi. Pe lângă productivitatea redusă, o altă implicaţie a caracterului programării o constituie problemele de fiabilitate a produselor - program rezultate.

Faptele de mai sus au dus la intense cercetări în vederea găsirii unor căi de creştere a productivităţii muncii de programare şi a calităţii programelor dezvoltate. Au fost astfel elaborate diverse metodologii de dezvoltare a programelor, au fost introduse tehnici de organizare şi conducere a proiectelor mari de programe, au fost implementate noi limbaje de programare, au fost dezvoltate instrumente de programare automată sau de asistare a personalului implicat etc. Cu toate acestea, nu a fost găsită o modalitate de a creşte substanţial productivitatea şi fiabilitatea produselor - program. Singurul salt considerabil în creşterea productivităţii programării a fost introducerea compilatoarelor, acum aproximativ 35 de ani. De atunci au fost făcute progrese, dar nu la fel de mari. În acest sens trebuie evidenţiate tehnicile de programare structurată şi tehnicile de structurare şi abstractizare a datelor.

Există în literatura de specialitate multe analize şi opinii asupra cauzelor care determină dificultăţile şi problemele care apar în dezvoltarea produselor-program. Vor fi considerate, în continuare, două dintre cele mai importante motive: complexitatea mare a programelor şi vocaţia lor de a fi permanent schimbate.

Complexitatea produselor - program depăşeşte, după cum remarca F.P. Brooks [4], orice altă creaţie umană. Multe probleme ale dezvoltării programelor derivă din această complexitate. De exemplu, datorită complexităţii programelor, apar problemele manageriale şi ale

comunicării între membrii echipelor de elaboratori de programe. Asigurarea fiabilității, în general, a calității produselor-program, este dificilă tot datorită complexității care face imposibilă supervizarea tuturor stărilor posibile ale unui program. Dificultatea extinderii și modificării programelor este datorată tot complexității acestora.

Pentru stăpânirea complexității programelor au fost folosite în principal, două căi care nu se exclud una pe alta. Prima cale a fost elaborarea de limbaje de programare evoluate, care oferă abstracții în direcția micșorării volumului și gradului de detaliere ale produselor - program. Astfel, o parte din complexitatea problemei pentru care se scrie un program este preluată de complexitatea limbajului în care se scrie programul. Aici apar însă câteva probleme. Ideal ar fi să se furnizeze un limbaj universal foarte puternic, care să preia cât mai mult din complexitatea problemei. Acest ideal nu poate fi atins deoarece, o dată cu creșterea complexității limbajului, complexitatea compilatoarelor care îl traduc crește atât de mult încât realizarea acestora devine problematică. Totodată, un limbaj foarte complex devine și foarte greu de învățat și stăpânit. În fine, o altă problemă este traducerea eficientă în cod a unui program scris în astfel de limbaje.

O a doua cale de stăpânire a complexității a fost încercarea de a sistematiza activitatea de dezvoltare a programelor. În acest sens, s-a conturat ideea că activitatea de construire a programelor trebuie să devină o disciplină inginerească, abordabilă în mod similar cu alte domenii inginerești. O consecință directă a acestei idei este necesitatea realizării unei standardizări care să permită reutilizarea sub cât mai multe aspecte. Standardizarea este o modalitate utilizată în toate domeniile inginerești pentru a reutiliza de un foarte mare număr de ori proiectarea elementelor standardizate. Metodologiile, îndrumarele, ghidurile sunt și ele tot modalități de reutilizare, în acest caz a experienței în dezvoltarea unor produse similare. În consecință, o posibilitate de stăpânire a complexității dezvoltării programelor este reutilizarea pe o scară cât mai largă și la mai multe niveluri a rezultatelor obținute.

Reutilizarea în dezvoltarea programelor este foarte importantă, putând spune, de fapt, că ea constituie esența programării automate [10]. Conceptul de reutilizare a evoluat în timp [7]. Una din primele forme de reutilizare au fost bibliotecile de subrutine. Compilatoarele sunt și ele, dintr-un punct de vedere, tot o modalitate de reutilizare. Alte abordări în direcția reutilizării sunt sistemele parametrizate și generatoarele de aplicații.

Un al doilea aspect care mărește dificultatea elaborării de produse-program este faptul că acestea sunt în permanență supuse schimbării. Spre deosebire de alte produse ale activității umane (de exemplu, clădiri, mașini etc.), datorită maleabilității programelor există permanent tentația de a le modifica, de a le îmbunătăți [4]. Această tendință este, bineînțeles, lăudabilă numai că, datorită complexității programelor, nu se pot detecta toate consecințele modificărilor făcute. De aceea, este necesar să se aibă în vedere vocația programelor de a fi schimbate și de a pleca în toate activitățile conexe dezvoltării programelor de la acest fapt.

3. Elemente de bază ale abordării orientate spre obiecte a programării.

3.1 Conceptul de programare orientată spre obiecte; conceptul de obiect.

În secțiunea anterioară s-a făcut o analiză a cauzelor care concură la gradul mare de dificultate în elaborarea programelor. Programarea orientată spre obiecte este o paradigmă de programare care își propune tocmai depășirea problemelor discutate. Conceptul de programare orientată spre obiecte a apărut în urma mai multor cercetări, atât din domeniul teoriei limbajelor de programare (tipuri de date abstracte), a inteligenței artificiale (reprezentarea cunoștințelor, modele de rezolvare distribuită a problemelor), a ingineriei programării (reutilizarea unor fragmente de program), cât și al modelării și simulării.

Spre deosebire de alte paradigme de programare (funcțională sau logică), OOP nu are o bază formală bine precizată. Obiectele sunt definite conform unei metafore antropomorfice: au o identitate, au o stare caracterizată de valorile unei mulțimi de atribute (componente) și o comportare determinată de o mulțime de proceduri atașate, denumite metode. Ele pot comunica prin transmiterea de mesaje, efectul recepționării unui mesaj de către un obiect constând în determinarea metodei atașate tipului respectiv de mesaj și în lansarea ei în execuție. Dintr-o altă perspectivă, un obiect poate fi comparat cu o structură de date activă, care ar putea fi văzută ca o înregistrare, având o parte din componentele ei procedurale.

OO constituie mai mult decât o simplă modalitate de a aborda faza de codificare din ciclul de viață al unui program. Ea face parte dintr-o concepție unitară de abordare a tuturor activităților din ciclul de viață având ca punct central conceptul de obiect. Această concepție are drept elemente de bază mai multe principii generale referitoare la modalități de stăpânire a complexității, care fuseseră deja formulate (eventual adaptate) și utilizate în ingineria programării: abstractizarea, modularizarea, ierarhizarea, asocierea, încapsularea prelucrărilor și partajarea.

3.2 Abstractizarea și încapsularea.

Abstractizarea (ca și modularizarea, ierarhizarea și asocierea) este un principiu folosit în aproape toate activitățile umane. Pentru a analiza, înțelege și dirija un sistem sau un fenomen complex, de obicei se face

abstracție de niște caracteristici considerate (eventual momentan) neesențiale ale acelui sistem sau fenomen pentru a rămâne doar elementele sale esențiale, definitorii. În programare, în funcție de aspectele avute în vedere, se folosesc abstractizări ale datelor, abstractizări procedurale, de control sau temporale.

Abordarea OO este (printre altele) o modalitate de abstractizare a datelor. După cum remarca B.Meyer, "obiectele trebuie descrise ca implementări de tipuri de date abstracte" [8]. Aceasta presupune identificarea și definirea de grupe de structuri de date similare. În plus, conceptul de tip de date abstract presupune un mecanism de separare a specificației de implementare. Acest lucru este deosebit de important și semnifică faptul ca un utilizator al unui grup trebuie să cunoască numai cum să utilizeze clasa respectivă. Detaliile asupra cum se implementează tipul de date sunt ascunse utilizatorului. În acest sens, un tip de date abstract poate fi imaginat ca o capsulă, ca o cutie neagră pe care programatorul poate să o utilizeze, să-i observe comportarea, dar pe care nu poate să o desfacă să vadă ce este înăuntru.

De exemplu, să considerăm o capsulă pentru tipul de date coadă. Utilizatorul poate adăuga un element la coadă, poate cere eliberarea următorului element disponibil din coadă, poate testa dacă coada este goală. El nu știe și nici nu trebuie să fie preocupat de cum este implementată coada (vector cu doi indici, lista circulară cu doi pointeri etc).

Avantajele acestei separări între specificare și implementare sunt următoarele:

- În acest mod se ușurează considerabil modificarea capsulelor. Deoarece toate utilizările capsulelor nu se referă la detaliile interne, de implementare, este sigur că la înlocuirea implementării unei capsule cu altă implementare care are aceeași comportare nu mai trebuie făcute modificări în afara capsulei. Dacă nu aveam încapsularea prelucrărilor, în momentul înlocuirii, de exemplu, a implementării unei cozi printr-un vector cu una prin lista circulară, trebuia să se verifice că în program nu se fac alte referiri la variabilele interne implementării cum ar fi, de exemplu, indicii vectorului care indică începutul și sfârșitul cozii. Această tendință de referire la variabilele interne ale unei implementări (în cazul când nu se face încapsularea prelucrărilor) apare de foarte multe ori în programare și, de fapt, duce la o răspândire a informațiilor în întreg codul unui program. Această distribuire a informațiilor este una din principalele cauze ale dificultăților modificării programelor [1].
- O dată făcute verificări de corectitudine ale tipurilor de date abstracte (ale capsulelor), cum în interiorul acestor capsule nu se mai fac modificări ulterioare se reduce volumul de verificări și testări ulterioare.
- Capsulele pot fi reutilizate de ori câte ori este necesar.

În încheierea considerațiilor asupra tipurilor de date abstracte trebuie precizat că nu toate limbajele OOP restricționează total accesul la implementarea unei capsule. De exemplu, în C++ [11], extensia limbajului C cu caracteristici OOP, se pot declara componente ale unui obiect ca fiind publice și pot fi astfel accesate sau modificate de oricine și componente private sau protejate care nu pot fi accesate decât de metodele obiectului respectiv (sau de cele agreate - așa numiții prieteni) sau, eventual, de moștenitorii acestuia (în cazul componentelor protejate).

3.3 Moștenirea proprietăților

Modularizarea este o altă tehnică foarte răspândită în stăpânirea complexității. Prin descompunerea unui sistem complex în mai multe module (mai simple) care au un număr redus de interconexiuni se reduce bineînțeles complexitatea înțelegerii sau dezvoltării acelui sistem. Totodată, modularizarea duce și la flexibilitate și la posibilități de reutilizare. Abordarea OO este, prin însăși natura sa, o modalitate de a gândi modular o aplicație, fiecare obiect constituind un modul.

Ierarhizarea este și ea o tehnică folosită, nu numai în domeniul programării, ci și în organizarea a multora din sistemele complexe. În cadrul OO, în conexiune cu modularizarea și cu abstractizarea se folosesc curent ierarhii de obiecte la diverse niveluri de abstractizare. Aceste ierarhii sunt organizate conform relației așa-numită de "moștenire", prin care obiectele pot moșteni componente sau metode unele de la altele. De fapt, OOP poate fi caracterizată ca o combinație între tipuri de date abstracte și moștenire.

Moștenirea proprietăților are mai multe avantaje. În primul rând, asigură continuitate în dezvoltarea și în evoluția programelor. Continuitatea este un deziderat în ingineria programării în sensul că adăugări mici în specificație să ducă la adăugări mici în cod. Acest lucru este realizat natural prin moștenire. De exemplu, dacă se dorește introducerea unui nou concept, de pătrat, într-un program în care există deja conceptul de dreptunghi, acest lucru se rezolvă foarte ușor prin definirea clasei de obiecte pătrat care moștenește toate proprietățile clasei de obiecte dreptunghi. În noua clasă vor trebui definite numai particularitățile care o diferențiază de vechea clasă (de exemplu, faptul că pătratul are laturile egale).

Moștenirea, în conexiune și cu ideea de continuitate, discutată mai sus, sprijină și reutilizarea. Folosind moștenirea, obiectele se pot reutiliza pe scară mai largă decât, de exemplu, subrutinele. Astfel, în cazul în care se dorește reutilizarea cu modificări a unui obiect (situație care limita posibilitățile de reutilizare a subrutinelor) se poate crea un nou obiect care moștenește toate proprietățile obiectului reutilizat și care conține doar modificările necesare.

Moștenirea este și o modalitate de partajare a codului. De exemplu, se pot utiliza metodele de la obiectele de la care

se moşteneşte în metodele obiectelor care moştenesc. Această tehnică este cu atât mai puternică în cazul moştenirii multiple, adică în cazul în care un obiect poate moşteni de la mai multe obiecte. Aici trebuie făcută o remarcă importantă. Facilităţile oferite de combinarea metodelor de la mai multe obiecte în cazul moştenirii multiple nu pot fi oferite la întregul lor potenţial decât (deocamdată?) de limbajele OOP dezvoltate în Lisp. Acest lucru este datorat faptului că rezolvarea combinării metodelor implică, atât construirea, cât şi, apoi, execuţia dinamică a codului rezultat în urma combinării.

3.4 Asocieri de obiecte

În abordarea OO, obiectele nu sunt entităţi legate numai prin relaţii de moştenire. Între obiecte pot exista diverse relaţii prin care acestea se asociază. O primă relaţie posibilă între obiecte este relaţia "utilizează", prin care un obiect utilizează alt obiect. Altă relaţie foarte folosită în abordarea OO este relaţia determinată de faptul că un obiect poate avea drept componente alte obiecte.

4. Limbaje de programare OO

Istoric vorbind, primul limbaj de programare cu caracteristici rudimentare de programare orientată spre obiecte a fost SIMULA, un limbaj dedicat aplicaţiilor de simulare. Cercetările din inteligenţa artificială referitoare la reprezentarea cunoştinţelor au condus la conceptul de "frame" şi la evidenţierea ideii necesităţii reprezentării unor taxonomii de concepte care se află în relaţii de moştenire. Pe de altă parte, în anii '70 a apărut ideea tipurilor de date abstracte şi a primelor limbaje care ofereau astfel de construcţii (CLU, ALPHARD, EUCLID). Primul limbaj propriu-zis de programare orientată spre obiecte a fost SMALLTALK, în care absolut toate entităţile sunt obiecte şi toate prelucrările se fac prin transmitere de mesaje.

În prezent, au fost implementate noi limbaje OO, cum ar fi Eiffel sau au fost extinse limbaje existente, cum ar fi C++ [11] pentru C, CLOS pentru Common Lisp [12] sau versiunile OO ale Pascal-ului [13].

În cadrul Institutului de Cercetări în Informatică a fost dezvoltat mediul de programare orientată spre obiecte structurate XRL dezvoltat în Lisp şi destinat în special aplicaţiilor bazate pe cunoştinţe [2, 14].

4.1 Clase şi instanţe

Limbajele OOP se clasifică în două mari categorii în funcţie de modalitatea de creare de noi obiecte. O primă categorie, cea mai răspândită, este cea a limbajelor OOP bazate pe clase. În această categorie se face o separare netă între obiectele generice, denumite clase, şi obiectele individuale, denumite instanţe. O clasă constituie o descriere a unui grup de obiecte similare. Instanţa este o particularizare a unei clase, având structura şi comportarea dată de acea clasă.

În lucrul cu clase şi instanţe se disting următoarele tipuri de activităţi:

1. Definirea unei clase, deci a unui obiect generic care va fi punctul de plecare în definirea de obiecte individuale (instanţe). În definirea unei clase sunt indicate clasele de la care ea moşteneşte proprietăţi, componentele pe care le vor avea instanţele sale, eventual (în unele limbaje OOP) componente ale clasei (cu valori comune tuturor instanţelor) şi perechi selector - metoda prin care este indicată comportarea instanţelor. Selectorul este numele prin care este recunoscută cererea unei anumite comportări, iar metoda este o procedură, specifică clasei respective, care conţine comportarea corespunzătoare selectorului în cazul clasei curente.

De exemplu, putem defini clasa figurilor de tip dreptunghi (cu laturile paralele cu axele de coordonate), cu componentele x, y, l şi h (coordonatele colţului din stânga jos, lungimea respectiv înălţimea) şi cu metode ataşate pentru desenare, ştergere şi deplasare (nu folosim pentru exemplificare un limbaj OOP anume ci un limbaj generic):

clasa dreptunghi

componente: x=0, y=0, l, h

metode: desenează: desen_dreptunghi

ştergе: ştergе_dreptunghi

deplasează: deplasează_dreptunghi(xnou,ynou)

2. Declararea unui obiect particular. Pentru a declara o instanţă a unei clase se indică numele clasei şi, eventual, valori pentru o submulţime de componente. De exemplu, pentru a declara dreptunghiuri particulare se creează instanţe ale clasei anterioare, prin specificarea numelui noii instanţe, a numelui clasei şi a valorilor componentelor:

instanţa d122, clasa dreptunghi, x=11, y=15, l=10, h=20

instanţa d125, clasa dreptunghi, l=5, h=35

După aceste declaraţii, d122 şi d125 sunt numele instanţelor respective, nume care pot fi utilizate pentru referiri la ele. Pentru instanţa d125 nu au fost precizate valorile lui x şi y. Acest lucru nu este o eroare, pentru aceste componente luându-se valorile implicite, indicate în definiţia clasei (x = 0 şi y = 0), practica uzuală în OOP.

3. Accesarea unor valori ale componentelor obiectelor, prin indicarea numelui obiectului şi a componentei respective. De exemplu, înălţimea lui d125 va fi d125.h, coordonatele lui d122 vor fi d122.x respectiv d122.y.

4. Actualizarea componentelor unor obiecte. De exemplu, schimbarea înălţimii lui d125 s-ar putea face prin atribuirea:

d125.h:=40

)bservație: În unele limbaje OOP, o parte din componente unt protejate la accesul sau actualizarea făcute din afara biectului căruia aparțin, din considerentele de încapsulare iscutate într-o secțiune anterioară.

. Trimiterea de mesaje unui obiect. Trimiterea unui mesaj este similară apelului unei proceduri dintr-un limbaj de programare clasică. Diferența constă în faptul că la apel (la trimiterea mesajului) este indicată prelucrarea dorită și obiectul țintă și nu procedura care va efectua prelucrarea. Procedura (sau procedurile) care efectuează prelucrarea sunt cunoscute doar de obiectul țintă.

)e exemplu, pentru deplasarea lui d122 din punctul (11,15) n punctul de coordonate (25,65) i se va trimite mesajul:

deplasează(d125, x=25, y=65)

\cest mesaj va fi prelucrat conform atașării procedurale, ndicată în clasa (va fi activată procedura leplasează_dreptunghi). După trimiterea acestui mesaj 'alorile lui x și y ale lui d125 vor fi, bineînțeles, noile valori ndicate (x=25 si y=65).

) a doua categorie de limbaje OOP sunt cele în care nu xistă o distincție între clase și instanțe. În aceste limbaje, rearea unui obiect particular se face plecând de la un obiect xistent (denumit prototip), prin moștenire. Acesta, la ândul lui poate fi prototip al unui alt obiect și așa mai leparte. Eliminarea distincției clasă-instanță permite afinarea (instanțierea) dinamică de oricâte ori a unui)biect.

.2 Polimorfism și legare dinamică

\bordarea OO permite introducerea unui mecanism foarte)uternic, care nu este întâlnit decât foarte rudimentar în imbajele clasice de programare: polimorfismul. Acesta onstă în faptul că o anumită prelucrare poate avea mai nulte forme în funcție de argumentele sale. Un exemplu implu de polimorfism sunt operatorii care, în funcție de ontext, au semnificații multiple (overloading). De xemplu, în Pascal operatorul + poate avea semnificația de umă algebrică sau de reuniune de mulțimi, în funcție de ipul operanzilor

n limbajele OOP, polimorfismul, în conexiune cu legarea inamică, ajunge la adevăratul său potențial. De exemplu, ă considerăm că există mai multe obiecte O1, O2, ... On,)ate răspunzând la mesajul m (fiecare în parte cu altă 1etodă). Să considerăm, de asemenea, că o variabilă v este :gată dinamic la unul din obiectele de mai sus. Trimiterea 1esajului m la obiectul legat la variabila v va avea efecte iferite în funcție de obiectul care a fost legat (dinamic, deci ı momentul execuției) variabila v. În limbajele clasice cest lucru nu era posibil, el trebuind implementat printr-un case" în care, în funcție de valoarea variabilei se apelează rocedura corespunzătoare.

Polimorfismul are consecinte directe asupra creșterii calității programelor. Putem argumenta acest fapt prin reducerea dimensiunii programului (nu mai este necesar "case"-ul de mai sus) cu efecte, atât asupra simplificării dezvoltării sale cât și asupra lizibilității textului rezultat și prin posibilitatea de a extinde sau modifica programul prin faptul că introducerea de noi obiecte nu implică modificări în codul existent (spre deosebire de limbajele clasice în care ar fi trebuit să se introducă noi ramuri în "case").

În finalul considerațiilor asupra polimorfismului trebuie remarcat că acesta, în cazul limbajelor care permit moștenirea multiplă cu combinarea metodelor (deci cu partajare complexă a codului) devine un instrument deosebit de performant. De fapt, acest lucru este un argument pentru faptul că trăsăturile limbajelor OOP se integrează sinergetic.

În încheiere, deși nu are legătură cu polimorfismul ci cu proprietățile dinamice ale limbajelor OOP trebuie făcută o observație asupra unui element pe care ar trebui să îl aibă sistemele OOP [8]: gestiunea automată a memoriei, respectiv alocarea și dealocarea automată a spațiului de memorie pentru obiecte. Din păcate, multe din sistemele care se declară ca fiind de programare orientată spre obiecte nu îndeplinesc această cerință.

5. Metodologii de analiză și de proiectare orientată spre obiecte

Deși această abordare este de dată relativ recentă au fost deja elaborate mai multe metodologii de analiză și proiectare OO.

Pot fi enumerate astfel metodologiile: [3],[5],[9], Wirfs-Brock, Schlaer și Mellor, Colbert, IBM, Kurtz, Martin și Odell, Berard, și nu au fost enumerate chiar toate. Plecând de la aceste metodologii au fost implementate și sisteme automate de asistare a dezvoltării de programe (sisteme CASE). În cele ce urmează vor fi considerate pe scurt doar primele patru metodologii din cele enumerate mai sus.

O metodologie de dezvoltare a programelor este un ansamblu integrat de tehnici și de metode care are drept scop direcționarea și sprijinirea procesului de elaborare a programelor. În general aceste metodologii pleacă de la principiile de reducere a complexității discutate anterior în această lucrare și folosesc intensiv metode grafice de reprezentare. Până la apariția metodologiilor OO au existat două clase de abordări în acest sens: abordarea bazată pe descompunerea funcțională și cea bazată pe fluxul datelor. Deși au dus la sporiri sensibile a productivității dezvoltării programelor, și a calității programelor rezultate, nici una din metodologiile propuse nu au avut drept consecință un salt remarcabil în acest sens. Cauzele sunt multiple, în esență, problemele care limitau creșterea productivității fiind legate de modelul irevocabil, în cascadă, al ciclului

de viață, de dificultatea evoluției și reutilizării la toate etapele ciclului.

Metodologiile OO pleacă de la conceptul de obiect. Acest fapt are niște consecințe remarcabile. În primul rând, acest concept este prezent în toate fazele ciclului de viață (analiză, proiectare, codificare, întreținere, evoluție) fiind un element de legătura între ele. În al doilea rând, abordarea OO, după cum am mai accentuat în lucrare, sprijină reutilizarea și evoluția. În acest sens, se pot dezvolta programele după un model evolutiv, mult mai flexibil decât modelul în cascadă. În al treilea, și nu în cel din urmă rând, el este mult mai apropiat de modul natural de percepere umană a realității, ceea ce ușurează munca analistului și permite și accesul unei game mai largi de utilizatori nespecialiști în ingineria programării.

Toate metodologiile OO au drept obiectiv esențial realizarea unei modelări bazate pe obiecte a aplicației considerate. Pentru aceasta, ele oferă diverse formalisme grafice, care ușurează evidențierea obiectelor (clase și instanțe) care sunt identificate, atributele și comportarea lor, relațiile între ele (relații de moștenire, relații "utilizează" etc), gruparea lor. Nu există încă un standard asupra modalității de reprezentare grafică dar, în esență, toate metodologiile folosesc, în diverse variante, cam aceleași elemente.

Între cele patru metodologii analizate se distinge cea a grupului condus de Rumbaugh [9] prin faptul că permite reprezentarea mult mai nuanțată a asocierilor între obiecte și dă posibilitatea exprimării grafice a restricțiilor și a eventualelor homomorfisme între elementele diagramelor. Pe de altă parte, această bogăție de posibilități de reprezentare grafică poate avea drept consecințe dificultatea însușirii de către utilizatori a tuturor posibilităților.

Un al doilea element esențial în dezvoltarea unei aplicații OO este faptul că, în afara modelului structural, static al obiectelor, mai trebuie considerate și alte aspecte ortogonale. De exemplu, trebuie exprimate și aspectele dinamice, ale comportării și interacțiunii obiectelor. Aceasta constituie un alt punct comun pentru majoritatea metodologiilor. Astfel, în metodologiile Rumbaugh și Booch sunt luate în considerare și diagrame de tranziție a stărilor și diagrame de timp pentru relevarea aspectelor dinamicii sistemului considerat. În metodologia Rumbaugh există posibilitatea de a utiliza și diagrame ale fluxului datelor, în care pot apare obiecte pe post de agenți de la care pornesc și ajung fluxuri de date.

În metodologia Wirfs-Brock se pleacă de la un punct de vedere interesant asupra unei mulțimi de obiecte: responsabilitatea. În acest sens, se consideră că între obiectele care colaborează apare o relație de tip contractual.

Considerațiilor asupra metodologiilor OO vor fi încheiate cu câteva remarci critice. În primul rând, este normal ca în aplicații mari să apară un număr considerabil de obiecte. În consecință, în majoritatea metodologiilor există o modalitate de grupare a obiectelor. Totuși, în acest sens soluțiile puse la dispoziție sunt încă doar un prim pas spr găsirea unor modalități mai nuanțate de grupare.

Un alt aspect în care probabil că vor fi dezvoltat metodologiile OO este exprimarea mai multor puncte d vedere asupra unor grupuri de obiecte. În fine, o dată c extinderea moștenirii multiple cu combinări va trebu inclusă și această facilitate în formalismele grafice.

6. Concluzii

Sperăm că prezentarea făcută în lucrare este relevant asupra avantajelor evidente ale abordării OO. Deși aceast abordare este de dată recentă ea a pătruns în toate domeniil informaticii. Există un corp de concepte comune, dar înc există și concepții diferite, reflectate și în absența unu consens în privința chiar a unor elemente de bază al abordării.

Bibliografie

1. **BALZER, R.**: A 15 Year Perspective on Automati Programming. În: IEEE Transactions on Softwar Engineering, vol. SE-11, nr. 11, noiembrie 1985, pp 1257-1268.
2. **BĂRBUCEANU, M., TRAUȘAN-MATU, ST.** Integrating Declarative Knowledge Programmin Styles and Tools in a Structured Object Environment In: J. Mc.Dermott (ed.) Proceedings of 10-t International Joint Conference on Artificia Intelligence IJCAI'87, Milano, Italia, Morgan Kaufmann Publishers, San Francisco, CA., 1987.
3. **BOOCH, G.**: Object-Oriented Design wit Applications, The Benjamin-Cummings Co, Inc. 1991
4. **BROOKS, F.P.**: No Silver Bullet. Essence an Accidents of Software Engineering. In: Computer aprilie 1987, pp. 10-19.
5. **COAD, P., YOURDON, E.**: Object-Oriente Analysis, Yourdon Press, 1991.
6. **FRENKEL, K.A.**: Toward Automating the Software Development Cycle. In: Communications of the ACM vol. 28, nr. 6, iunie 1985, pp. 578-589.
7. **HOROWITZ, E., MUNSON, J.B.**: An Expansiv View of Reusable Software. In: IEEE Transactions o Software Engineering, vol. SE-10, no. 5, septembri 1984, pag. 477-487;
8. **MEYER, B.**: Object-Oriented Software Construction Prentice Hall, 1988.
9. **RUMBAUGH, J., BLAHA, M., PREMERLANI W., EDDY, F., LORENSEN, W.**,: Object-Oriente Modeling and Design, Prentice-Hall, Englewoo Cliffs, 1991.

0. **RICH, C., WATERS, R.C.**: Automatic Programming: Myths and Prospects. In: Computer, august 1988.
1. **STROUSTRUP, B.**: The C++ Programming Language, Addison-Wesley, Reading , MA, 1987.
2. **TELLO, E.R.**: Object-Oriented Programming for Artificial Intelligence. A Guide to Tools and System Design, Addison-Wesley, Reading, MA, 1989.
13. * * * TurboPascal, Version 6.0, User's Guide, Borland International, 1990.
14. **TRAUSAN-MATU, ST.**: Micro-XRL: An Object-Oriented Programming Language for Microcomputers, Raport de cercetare, Technical Cybernetics Institute, Bratislava, 1989.

INSTITUTUL DE CERCETĂRI ÎN INFORMATICĂ
B-dul Mareşal Averescu, Sect. 1, cod 71316, Bucureşti-Romania

RESEARCH INSTITUTE FOR INFORMATICS
8-10 Averescu Avenue, 71316, Bucharest 1, ROMANIA

ANALISIS v.1

Descriere produs:

Instrument software de asistare a analizei de sistem şi a elaborării de proiecte directoare de informatizare. Permite modelarea, evaluarea şi documentarea diferitelor aspecte ale sistemului: STRUCTURA ORGANIZAŢIEI, STRUCTURA SISTEMULUI, FLUXURILE DE DATE, STRUCTURA DATELOR cât şi DESCRIEREA PROIECTELOR INFORMATICE propuse.

Domenii de aplicabilitate:

Elaborarea de proiecte directoare de informatizare pentru orice tip de organizaţie (administraţie publică, agenţi economici etc.).

Configuraţie hardware-software:

- calculator compatibil IBM-PC-AT
- sistemul de operare MS-DOS v.3.0. şi superioare.

Punct de contact:

lab. Metodologii şi Instrumente asociate, Ileana Trandafir,
tel. 665.60.60 şi 665.70.15/212

Phone: +40-1-665.31.90
Telex: 11891 icpci r
Fax: +40-1-212.08.85

INSTITUTUL DE CERCETĂRI ÎN INFORMATICĂ
B-dul Mareşal Averescu, Sect. 1, cod 71316, Bucureşti-Romania

RESEARCH INSTITUTE FOR INFORMATICS
8-10 Averescu Avenue, 71316, Bucharest 1, ROMANIA

"Efectele negative ale nerezolvării unei probleme sunt direct proporţionale cu durata amânării deciziei"
(propunere pentru un dicţionar de locuţiuni pentru Managementul Postmodern)

Este adevărat că Dumneavoastră:

1. Aveţi gestionat o gamă de produse ofertate în continuă dinamică?
2. Prospectele tipărite vă costă scump şi devin repede depăşite?
3. Doriţi să informaţi potenţialii dumneavoastră clienţi într-o manieră modernă?
4. Doriţi să vă utilizaţi propria tehnică de calcul pentru prezentarea produselor sau serviciilor dumneavoastră la diferite târguri şi expoziţii?

Este adevărat că NOI ŞTIM să:

1. Preluam informaţiile complete despre oferta dumneavoastră şi le valorificăm în folosul dumneavoastră!
2. Vă pregătim cataloage la zi, consultabile pe calculator!
3. Promovăm informaţiile dumneavoastră în diferite reţele de informare existente şi în curs de constituire!
4. Vă asistăm în proiectarea şi execuţia materialului informativ computerizat cu care vă prezentaţi la expoziţii şi târguri!

Acordaţi câteva secunde din timpul dumneavoastră pentru a vedea la lucru asemenea cataloage

AICI şi ACUM

Punct de contact:

Lab. 2.22, 665.60.60 int. 175 sau 182, Iuliu Bara, Mircea Martiş

Phone: +40-1-665.31.90
Telex: 11891 icpcir
Fax: +40-1-212.08.85