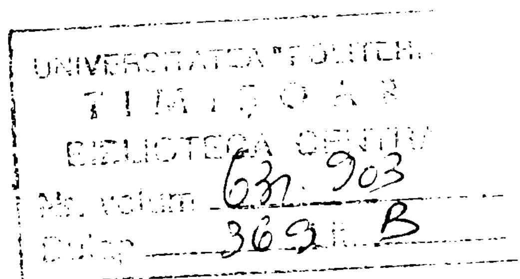
UNIVERSITATEA “POLITEHNICA” TIMIŞOARA

ing. Horaţiu Fara

CONTRIBUŢII LA O SOLUŢIE ORIENTATĂ SPRE OBIECTE PENTRU DEZVOLTAREA UNUI SISTEM INTEGRAT DE PROIECTARE

- TEZA DE DOCTORAT -

BIBLIOTECA CENTRALĂ
UNIVERSITATEA "POLITEHNICA"
TIMIŞOARA



Conducător ştiinţific:
Prof.Dr.Ing. George G. Savii

Timişoara
2001

BUPT

Mi-aş fi dorit ca Doru Mureşan să nu fi plecat dintre noi. Contribuţia sa la geneza actualei lucrări a fost esenţială. Mi-a fost mai mult decât mentor. Mi-a fost prieten.

Mulţumesc dlui. prof.dr.ing. George Savii pentru îndrumarea şi sprijinul pe care m i l-a acordat pe întreg parcursul elaborării lucrării, atât în privinţa conţinutului şi a ţinutei, cât şi a formei lucrării.

Mulţumesc dlui. prof.dr.ing. Toma-Leonida Dragomir pentru rigoarea, pertinenţa şi exactitatea observaţiilor pe care le-a făcut în urma lecturii primei versiuni a lucrării.

Mulţumesc colegilor mei din cadrul firmelor Computing Approach şi Dietrich's AG pentru atmosfera familiară şi pentru climatul de o ţinută profesională de excepţie care mi-au oferit şansa de a confrunta conceptele prezentate în cadrul tezei cu practica analizei, proiectării şi programării. Discuţiile legate de sistemul de ferestre şi de editorul 2D au contribuit în mod deosebit la conturarea metodei de analiză şi proiectare prezentată în cadrul tezei. Recunoştinţa mea se îndreaptă în mod special spre Voichiţa Mureşan, Uwe Emmer, Peter Philipps, Călin Stoicănescu, Dan Iosif, Doru Marchiş, Cristi Savii, Gerdi Ogârcin, Delia şi Dan Raţiu, Graţiana Riviş, Sorin Simu.

Nu în ultimul rând, recunoştinţa mea se îndreaptă spre membrii familiei. Mulţumesc soţiei mele pentru dragoste, sprijin şi înţelegere, timpul destinat elaborării lucrării afectând în mare măsură timpul pe care l-am dedicat familiei. Mulţumesc mamei mele şi tatălui meu pentru afecţiunea, armonia şi echilibrul care au marcat formarea mea. Le mulţumesc lor şi părinţilor soţiei mele pentru ajutorul pe care ni l-au oferit în această perioadă şi pentru faptul că au insistat ca această lucrare să fie dusă la bun sfârşit.

BUPT

CAPITOLUL 1 Obiectul şi conţinutul tezei de doctorat 6

CAPITOLUL 2 Elemente de tehnologie orientată spre obiecte. Stadiul actual 9
2.1 Introducere 9
2.2 Contextul şi importanţa programării orientate spre obiecte 9
2.3 Paradigme ale orientării spre obiecte 10
2.3.1 Clase şi obiecte 10
2.3.2 Încapsularea şi ascunderea informaţiilor 11
2.3.3 Abstractizarea datelor şi genericitatea 13
2.3.4 Responsabilităţi 17
2.3.5 Colaborări şi transmiterea de mesaje 18
2.3.6 Moştenirea 19
2.3.7 Polimorfismul 20
2.3.8 Legarea 22
2.3.9 Modularitatea 23
2.4 Clase şi obiecte 24
2.4.1 Definiţii 24
2.4.1.1 Obiectele 24
2.4.1.2 Clasele 25
2.4.1.3 Rolul claselor şi al obiectelor în analiza şi proiectarea orientată spre obiecte 25
2.4.2 Crearea claselor 26
2.4.3 Moştenirea şi ierarhiile de clase 27
2.4.4 Agregarea 28
2.4.5 Asocierea 29
2.4.6 Utilizarea 30
2.4.7 Clase „Mixin” 31
2.4.8 Clase container 31
2.4.9 Metaclase 32
2.5 Modele utilizate în analiza şi proiectarea orientată spre obiecte 32
2.5.1 Modele orientate spre obiecte 33
2.5.2 Modele statice 34
2.5.3 Modelele dinamice 37
2.5.4 Modelele de structură a sistemului 40
2.5.5 Modelele de concurenţă 41
2.5.6 Modelele funcţionale 42
2.6 Metode de analiză şi proiectare orientate spre obiecte 42
2.6.1 Tehnica modelării obiectelor (OMT) 42
2.6.1.1 Etapa OOA 44
2.6.1.2 Proiectarea sistemului 44
2.6.1.3 Etapa OOD 46
2.6.1.4 Aspecte legate de operarea în timp real şi de concurenţă 46
2.6.1.5 Instrumente CASE 46
2.6.2 Metoda Object-Oriented Software Engineering (OOSE) 47
2.6.2.1 Etapa OOA 47
2.6.2.2 Proiectarea sistemului 48
2.6.2.3 Etapa OOD 49
2.6.2.4 Aspecte legate de operarea în timp real şi de concurenţă 50
2.6.2.5 Instrumente CASE 50

BUPT

2.6.3 Metoda Booch'93 50
2.6.3.1 Etapa OOA/OOD 51
2.6.3.2 Proiectarea sistemului 54
2.6.3.3 Aspecte legate de operarea în timp real şi de concurenţă 54
2.6.3.4 Instrumente CASE 55
2.6.4 Metoda Shlaer-Mellor 55
2.6.4.1 Etapa OOA 56
2.6.4.2 Proiectarea sistemului 58
2.6.4.3 Etapa OOD 58
2.6.4.4 Proiectarea recursivă 59
2.6.4.5 Aspecte legate de operarea în timp real şi de concurenţă 60
2.6.4.6 Instrumente CASE 61
2.6.5 Metoda RDD (Responsibility-Driven Design) 61
2.6.5.1 Etapa OOA 61
2.6.5.2 Etapa OOD 62
2.6.5.3 Proiectarea sistemului 63
2.6.6 Metoda Coad-Yourdon 63
2.6.6.1 Etapa OOA 63
2.6.6.2 Proiectarea sistemului 66
2.6.6.3 Etapa OOD 66
2.6.6.4 Aspecte legate de operarea în timp real şi de concurenţă 67
2.6.6.5 Instrumente CASE 68
2.6.7 UML – un standard în formare în domeniul analizei şi proiectării orientate spre obiecte 68
2.7 Concluzii. Contribuţii 69

CAPITOLUL 3 Abordare bazată pe tehnologia orientată spre obiecte a procesului de dezvoltare software 72
3.1 Introducere 72
3.2 Comparaţie între metodele structurate şi metodele orientate spre obiecte 73
3.3 Procesul de dezvoltare a unei aplicaţii orientate spre obiecte 75
3.3.1 Analiza domeniului 77
3.3.1.1 Înţelegerea domeniului problemei 77
3.3.1.2 Crearea claselor şi obiectelor lumii reale 78
3.3.1.2.1 Clasificare 79
3.3.1.2.2 Identificarea claselor şi a obiectelor 79
3.3.1.3 Evaluarea obiectelor 79
3.3.1.4 Produse de lucru 80
3.3.1.5 Domenii de risc 81
3.3.2 Analiza cerinţelor sistemului 82
3.3.2.1 Utilizarea scenariilor 82
3.3.2.2 Definiţia scenariului 82
3.3.2.3 Proprietăţile scenariilor 83
3.3.2.4 Identificarea scenariilor 84
3.3.2.5 Documentarea scenariilor 86
3.3.2.6 Proiectarea bazată pe scenarii 87
3.3.2.7 Analiza orientată spre obiecte 88
3.3.2.8 Modelul de analiză 89
3.3.2.8.1 Obiecte interfaţă 90
3.3.2.8.2 Obiecte entitate 91

BUPT

3.3.2.8.3 Obiecte de control 92
3.3.2.8.4 Corelarea obiectelor de analiză 92
3.3.2.8.5 Criterii de încheiere a fazei de analiză 93
3.3.2.9 Euristica creării obiectelor de analiză 94
3.3.2.10 Selecția claselor 96
3.3.2.11 Evaluarea obiectelor de analiză 96
3.3.2.12 Domenii de risc 97
3.3.2.13 Produse de lucru 98
3.3.3 Proiectarea sistemului 99
3.3.3.1 Partiționarea 100
3.3.3.1.1 Perspective 101
3.3.3.1.2 Partiții 101
3.3.3.1.3 Principii de partiționare 102
3.3.3.1.4 Produse de lucru ale partiționării 105
3.3.3.2 Configurarea 105
3.3.3.2.1 Alocarea funcțională 106
3.3.3.2.2 Utilizarea metodelor structurate în cadrul configurării 107
3.3.3.2.3 Arhitectura hardware 107
3.3.3.2.4 Produsele de lucru ale configurării 108
3.3.3.2.5 Metode orientate spre obiecte pentru crearea subsistemelor 109
3.3.3.2.6 Abordare bazată pe scenarii pentru crearea subsistemelor 109
3.3.4 Analiza cerințelor software 110
3.3.4.1 Modele 111
3.3.4.2 Analiza cerințelor software orientată spre obiecte 112
3.3.4.3 Crearea scenariilor 113
3.3.4.4 Identificarea claselor şi a obiectelor 113
3.3.4.5 Identificarea atributelor şi a operațiilor 114
3.3.4.6 Reprezentarea obiectelor 115
3.3.4.7 Modelarea datelor 116
3.3.4.8 Evaluarea claselor şi a obiectelor 116
3.3.4.9 Produse de lucru 117
3.3.5 Proiectarea software 117
3.3.5.1 Tranziția de la etapa de analiză 118
3.3.5.2 Structurarea proceselor 121
3.3.5.3 Proiectarea orientată spre obiecte 122
3.3.5.4 Produse de lucru 123
3.3.5.5 Tratarea excepțiilor 123
3.3.5.5.1 Categorii de excepții 124
3.3.5.5.2 Strategia de proiectare a mecanismului de tratare a excepțiilor 124
3.3.5.6 Evaluarea proiectării 125
3.3.5.6.1 Structura proceselor 126
3.3.5.6.2 Structura de clase 126
3.3.5.6.3 Interfețele claselor 127
3.3.5.6.4 Utilizarea moştenirii 128
3.3.5.6.5 Utilizarea excepțiilor 129
3.3.5.6.6 Metrici software pentru evaluarea structurii de proiectare 129
3.3.6 Implementarea 132
3.3.6.1 Tranziția de la etapa de proiectare 132
3.3.6.2 Programarea 133
3.3.6.3 Tratarea excepțiilor 134
3.3.6.3.1 Utilizarea mecanismului intern C++ de tratare a excepțiilor 134

BUPT

3.3.6.3.2 Mecanism de aserţiuni bazat pe excepţii implementat pentru C++ 136
3.3.6.4 Testarea 142
3.3.6.5 Depanarea 142
3.4 Concluzii. Contribuţii 143

CAPITOLUL 4 Soluţie orientată spre obiecte aplicată în dezvoltarea componentelor unui sistem CAD/CAM 145
4.1 Introducere 145
4.2 Descrierea sistemului de proiectare CAD/CAM 146
4.3 Procesul de dezvoltare orientat spre obiecte 147
4.3.1 Locul şi rolul bibliotecii de ferestre şi a editorului 2D în cadrul aplicaţiei 148
4.3.2 Procesul de dezvoltare software pentru biblioteca de ferestre şi editorul 2D 148
4.3.3 Sistemul de ferestre 149
4.3.3.1 Analiza domeniului 150
4.3.3.2 Analiza cerinţelor sistemului 157
4.3.3.3 Proiectarea sistemului 158
4.3.3.4 Analiza cerinţelor software 158
4.3.3.5 Proiectarea software 170
4.3.3.6 Implementarea 176
4.3.4 Editorul 2D 179
4.3.4.1 Contextul aplicaţiei client 179
4.3.4.2 Definirea cerinţelor sistem 179
4.3.4.3 Analiza şi proiectarea orientată spre obiecte 181
4.3.4.3.1 Analiza domeniului 181
4.3.4.3.2 Analiza cerinţelor sistemului 183
4.3.4.3.2.1 Cazuri de uz 183
4.3.4.3.2.2 Analiza comportamentului dinamic şi diagrama de tranziţie a stărilor 191
4.3.4.3.3 Proiectarea sistemului 194
4.3.4.3.4 Analiza cerinţelor software 195
4.3.4.3.5 Proiectarea software 200
4.3.4.3.5.1 Modelul de moştenire şi agregare 200
4.3.4.3.5.2 Stabilirea interfeţelor claselor 202
4.3.4.3.5.3 Proiectarea claselor pe baza diagramei de tranziţie a stărilor 202
4.3.4.3.5.4 Tratarea excepţiilor 204
4.3.4.3.5.5 Evaluarea proiectării 204
4.3.4.4 Implementarea 205
4.4 Concluzii. Contribuţii 205

CAPITOLUL 5 Concluzii şi perspective 209
5.1 Concluzii 209
5.2 Contribuţii 210
5.3 Perspective 212

Anexa 214
A.1 Interfeţele claselor din biblioteca de ferestre 214
A.1.1 Spaţiul de lucru 214
A.1.2 Fereastra 216
A.1.3 Rastrul 220
A.1.4 Sistemul de coordonate liber 221
A.1.5 Fereastra 2D 222

BUPT

A.2 Interfețele claselor editorului 2D 224
A.2.1 Elementele modulului UserInput 224
A.2.2 Elementele modulului Editor2D 232
A.3 Mecanismul de aserțiuni bazat pe excepții 235
A.3.1 Codul sursă al mecanismului de aserțiuni bazat pe excepții 235
A.3.2 Exemplu de implementare la nivelul clasei `CDhp2DFCS` 237
A.4 Implementarea aplicației Dhp-Bauwerk 238
A.4.1 Sistemul de ferestre 238
A.4.2 Editorul 2D 240
A.4.2.1 Integrarea editorului 2D în modulul „Grundriss" 240
A.4.2.2 Integrarea editorului 2D în modulul „Wandkonstruktion" 241
A.4.2.3 Integrarea editorului 2D în modulul „Dachausmittlung" 243
A.5 Referiri în literatura de specialitate cu privire la aplicația DHP-Bauwerk .. 243
Abrevieri şi notații 250
Bibliografia 253

BUPT

Obiectul şi conţinutul tezei de doctorat

Complexitatea aplicaţiilor din sfera CAD/CAM, a proiectării şi a fabricaţiei asistate de calculator se află într-un proces de continuă creştere. Numărul mare de facilităţi integrate în cadrul acestui tip de aplicaţii se caracterizează prin funcţionalitate din ce în ce mai sofisticată. Adaptarea la nevoi individuale ale utilizatorilor impune un nivel înalt de flexibilitate. Fiabilitatea şi productivitate reprezintă cerinţe esenţiale atât în contextul utilizării cât şi în cel al dezvoltării produsului software. Aceste aspecte reprezintă o pledoarie în favoarea utilizării unui demers riguros în procesul de dezvoltare a aplicaţiilor software.

Actuala teză de doctorat prezintă o abordare a procesului de dezvoltare a unei aplicaţii software aparţinând domeniului CAD/CAM pe baza tehnologiei orientate spre obiecte, argumentând pe parcursul fiecărui pas al acestui proces justeţea abordării propuse.

Capitolul 2 prezintă cele mai importante concepte legate de tehnologia orientată spre obiecte, concepte pe care se va clădi întreg procesul de dezvoltare software. Sunt prezentate paradigmele orientării spre obiecte abordând definirea claselor şi a obiectelor ca entităţi fundamentale in contextul sistemelor orientate spre obiecte.

Procesul de generare a claselor şi obiectelor este introdus prin analiza modelelor ce se utilizează în analiza orientată spre obiecte, insistând asupra tipului de informaţii pe care le redau fiecare dintre acestea.

O trecere în revistă a câtorva din cele mai semnificative metode de dezvoltare orientate spre obiecte, dintre care amintim:

- Metoda OMT
- Metoda OOSE
- Metoda Booch'93
- Metoda Coad-Yourdon
- Metoda Shlaer-Mellor
- Metoda RDD

are scopul de a evidenţia caracteristicile, avantajele şi dezavantajele oferite de fiecare dintre acestea, construind o punte spre capitolul următor, abordarea procesului de dezvoltare software propus în continuare bazându-se pe avantajele oferite de câteva dintre aceste metode.

BUPT

Capitolul 3 prezintă o propunere de abordare bazată pe tehnologia orientată spre obiecte a procesului de dezvoltare software al unei aplicaţii. Sunt urmărite etapele procesului de dezvoltare a unei aplicaţii:

- OOA (*O*bject *O*riented *A*nalysis): etapa de analiză orientată spre obiecte
- OOD (*O*bject *O*riented *D*esign): etapa de proiectare orientată spre obiecte
- OOP (*O*bject *O*riented *P*rogramming): etapa de implementare pe baza unui limbaj de programare orientat spre obiecte.

Paradigmele orientării spre obiecte prezentate în cadrul capitolului anterior constituie elemente de referinţă pentru desfăşurarea etapelor procesului de dezvoltare enunţate mai sus. Delimitarea clară a celor trei etape are ca rezultat un proces de dezvoltare flexibil, capabil să se adapteze cu efort minim la modificări ale cerinţelor. Sunt enunţate reguli clare de identificare a entităţilor de analiză, proiectare şi implementare. Fiecare etapă a procesului de dezvoltare software este încheiată cu o evaluare a rezultatelor pentru a asigura consistenţa acestuia. O evaluare a riscului implicat de abordarea orientată spre obiecte este în măsură să orienteze atenţia spre eventualele zone critice. Implementarea este completată de un mecanism performant de tratare a excepţiilor menit să garanteze detecţia celei mai mari părţi a problemelor legate de implementare încă în faza de depanare.

Capitolul 4 exemplifică aplicarea în practică a procesului de dezvoltare propus în cadrul capitolului 3 pentru dezvoltarea unor componente în cadrul sistemului de proiectare CAD/CAM *DHP Bauwerk*. O descriere rezumativă a contextului şi a principalelor trăsături ale aplicaţiei este urmată de abordarea orientată spre obiecte a dezvoltării a două componente esenţiale ale acesteia:

- *biblioteca de ferestre* reprezentând un server ce pune la dispoziţie clienţilor săi servicii de reprezentare a entităţilor geometrice 2D şi 3D în ferestre şi de interacţiune între acestea şi utilizator
- *editorul 2D*, modul ce oferă mediului gazdă servicii de generare, prelucrare şi identificare a entităţilor geometrice 2D.

Fiecare dintre aceste două componente ale aplicaţiei *DHP Bauwerk* are propriile particularităţi, astfel că paşii urmaţi în cadrul procesului de dezvoltare software vor diferi pentru cele două componente.

BUPT

Capitolul 5 este dedicat concluziilor la care s-a ajuns în urma realizării actualei teze, subliniind elementele originale, precizând direcţiile de dezvoltare care pot fi abordate în continuare.

Anexa prezintă codul sursă ca rezultat final al procesului de dezvoltare precum şi ilustraţii legate de implementarea celor două componente în cadrul aplicaţiei *DHP Bauwerk*.

BUPT

Elemente de tehnologie orientată spre obiecte. Stadiul actual

2.1 Introducere

Capitolul actual prezintă paradigmele orientării spre obiecte aflate la baza procesului de dezvoltare a unei aplicaţii. Acest set de paradigme reprezintă elementul axiomatic atât pentru faza de analiză şi proiectare a aplicaţiei cât şi în faza de implementare utilizând un limbaj de programare orientat spre obiecte.

Un spaţiu extins este alocat definiţiei şi analizei conceptului de clasă în contextul paradigmelor amintite anterior.

Procesul de dezvoltare bazat pe tehnologia orientată spre obiecte are la bază un set de modele statice, dinamice şi funcţionale ce se regăsesc în cadrul tuturor metodelor consacrate de dezvoltare orientate spre obiecte. Prezentarea acestor modele reprezintă un element introductiv pentru abordarea comparativă a câtorva din cele mai populare metode de analiză şi proiectare orientate spre obiecte.

2.2 Contextul şi importanţa programării orientate spre obiecte

Programarea orientată spre obiecte este considerată a fi cel mai important instrument de dezvoltare software apărut după programarea structurată [DEM79, PAG80, YOU79]. Programarea orientată spre obiecte a fost introdusă ca şi concept iniţial al tehnologiei orientate spre obiecte şi s-a dezvoltat în contextul unor limbaje de programare cum ar fi Simula [DAH70], Smalltalk-80 [GOL83], C++ [STR91, LIP91] şi Eiffel [MEY88].

Proiectarea orientată spre obiecte s-a dezvoltat pornind de la abordările sugerate de Abbott [ABB83], Booch [BOO86, BOO91, BOO93] şi Liskov [LIS86]. Avantajele majore ale proiectării orientate spre obiect sunt:

- *nivelul înalt de modularitate* facilitând cuplarea redusă între module,
- *mecanismul de încapsulare* ce facilitează ascunderea informaţiilor,
- *abstractizarea datelor* facilitând crearea de obiecte ce se referă la un singur tip de date şi la operaţiile asociate acestuia
- *mecanismul de clasificare şi moştenire* care facilitează reutilizabilitatea şi extensibilitatea.

BUPT

Conceptele orientării spre obiect au fost extinse de la faza de proiectare (OOD = *Object-Oriented Design*) la cea de analiză (OOA = *Object-Oriented Analysis*) [BOO93, JAC92, SHL92, COA91], astfel că tehnologia orientată spre obiecte poate fi utilizată în întreg procesul de dezvoltare al unui produs software.

Conceptele fundamentale ale tehnologiei orientate spre obiecte sunt prezentate în cele ce urmează. Unele dintre acestea nu sunt specifice numai programării orientate spre obiecte, ele pot fi aplicate în egală măsură analizei şi proiectării structurate.

2.3 Paradigme ale orientării spre obiecte

2.3.1 Clase şi obiecte

Termenii *clasă* şi *obiect* sunt utilizaţi pentru a desemna concepte ale tehnologiei orientate spre obiecte în cadrul analizei, proiectării şi programării orientate spre obiecte. Definiţiile acestor termeni variază în funcţie de nivelul de abstractizare, diverşi autori sugerând diverse definiţii:

- „Un lucru vizibil sau tangibil având o formă relativ stabilă; un lucru care poate fi perceput intelectual; un lucru spre care se îndreaptă gândul sau acţiunea." [RHCD75]
- „Un obiect are identitate, stare şi comportament" [BOO94]
- „Un obiect este o unitate de modularitate structurală şi comportamentală care are proprietăţi." [BUH88]
- Champeaux [CHA93] defineşte obiectul ca entitate conceptuală care:
  - este identificabilă
  - are trăsături care descriu un spaţiu local de stări
  - are operaţii care pot schimba local starea sistemului, incluzând şi operaţii din cadrul altor obiecte.

Definiţiile ce urmează reprezintă doar o abordare succintă destinată înţelegerii celorlalte concepte legate de tehnologia orientată spre obiecte.

Obiectul reprezintă un concept utilizat pentru a descrie entităţi specifice sistemului ce se dezvoltă. Fiecărui obiect îi este asociat un set de atribute şi de operaţii ce îi definesc comportamentul.

BUPT

Clasa reprezintă un prototip sau un şablon care descrie atributele şi operaţiile comune unui set de obiecte înrudite. Un obiect reprezintă o instanţă a unei clase şi este identificat printr-un nume sau printr-un identificator numeric.

Importanţa claselor şi obiectelor în dezvoltarea unui produs software este legată de descompunere, comunicare, documentare, reutilizare şi mentenabilitate:

- *Descompunerea.* Putem utiliza clasificarea ca mijloc de descompunere a unui sistem într-un număr de clase abstracte. Există relaţii directe între aceste clase şi obiectele asociate lor şi entităţi ale lumii reale. Clasele şi obiectele pot fi utilizate pentru a controla complexitatea sistemelor mari.
- *Comunicarea şi documentarea.* Fiind legate direct de entităţi ale lumii reale, clasele abstracte şi obiectele pot fi utilizate în cadrul discuţiilor cu beneficiarul sau cu utilizatorul final pentru a mări gradul de înţelegere a cerinţelor sistemului.
- *Reutilizabilitatea.* Strategia de clasificare implică o ierarhie şi un potenţial pentru moştenire, cum ar fi relaţia superclasă-subclasă. Subclasele pot moşteni atribute şi operaţii de la superclasă, subclaselor le pot fi adăugate atribute şi operaţii unice.
- *Mentenabilitatea.* Potenţialul de reutilizare al claselor şi obiectelor reprezintă un suport pentru extensibilitate, facilitând reconfigurarea pentru sisteme viitoare şi reducând efortul legat de mentenabilitate.

Această abordare informală a claselor şi a obiecte are rolul de introducere pentru descrierea în continuare a celorlalte concepte legate de orientarea spre obiecte ce se bazează şi sunt strâns legate de noţiunile de clasă şi de obiect.

2.3.2 Încapsularea şi ascunderea informaţiilor

Un element comun al analizei, proiectării şi programării orientate spre obiecte este reprezentat de utilizarea extensivă a abstractizărilor. O abstractizare reprezintă o descriere de nivel înalt sau un model al unui concept detaliat sau complex. Abstractizarea funcţională sau procedurală este utilizată în analiza, proiectarea şi programarea structurată.

O abstractizare ce reprezintă o entitate software trebuie să fie descrisă de către o notaţie uniformă care să poată fi recunoscută de către un dezvoltator software sau de un compilator sau alt instrument de procesare software. *Încapsularea* reprezintă o instanţiere a abstractizării, fiind singura cale de a accesa informaţia dorită pentru o abstractizare dată.

BUPT

Un obiect bine proiectat reprezintă încapsularea unei singure entităţi a lumii reale ce posedă un set de atribute şi de operaţii. O bună încapsulare a unui modul software are următoarele caracteristici la nivel de proiectare:

- *Implementarea detaliilor legate de structura de date şi de algoritmii utilizaţi nu este vizibilă utilizatorilor.* Rezultă un grad de conexiune redus între diversele module ale aplicaţiei care utilizează obiectul. Modificările ce vor fi făcute la nivelul implementării obiectului nu vor necesita modificări de proiectare la nivelul întregului sistem.
- *Partea vizibilă a modulului reprezintă o interfaţă unică şi neambiguă a obiectului încapsulat.* Această interfaţă reprezintă unica modalitate de a accesa structurile de date ce reprezintă abstractizarea. Acest aspect facilitează un înalt grad de reutilizabilitate, obiectele încapsulate pot fi utilizate în cadrul diferitor sisteme fără a modifica interfaţa acestora.

Încapsularea la nivel de programare este prezentată în continuare pe baza exemplului unei stive:

```
class Stiva {
  public:
    void initStiva();
    long varfStiva();
    long pop();
    void push(long);
  private:
    long elementeStiva[50];
    int indicatorStiva;
};
```

reprezentarea grafică a acestei încapsulări este ilustrată în figura următoare:

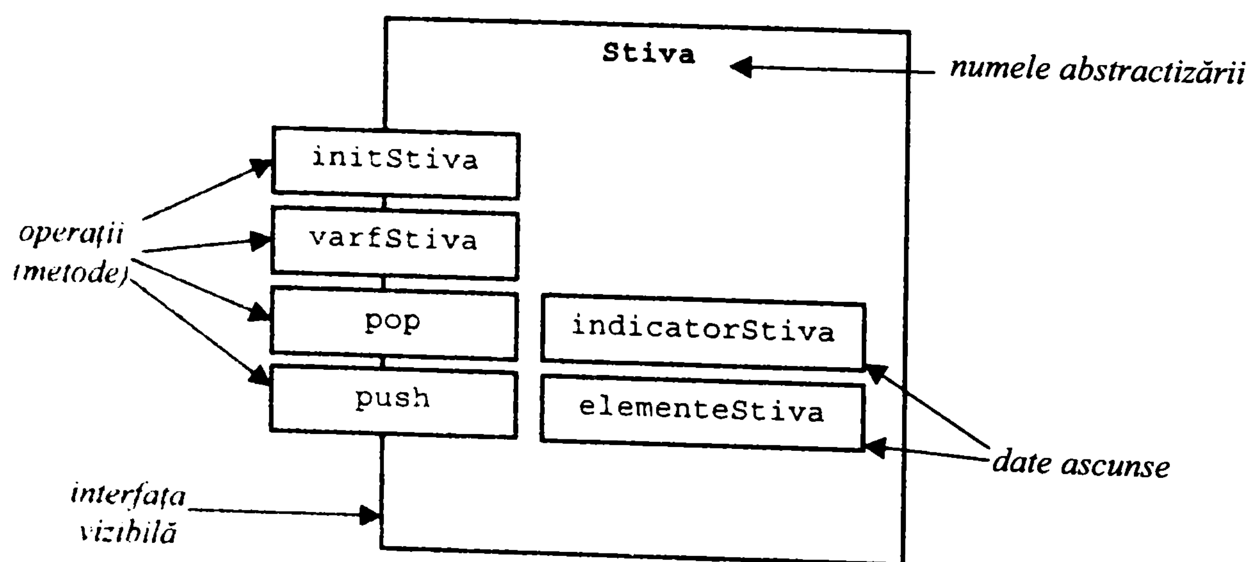


Figura 2.1 Obiect încapsulat

BUPT

Încapsularea constă din numele abstractizării, o interfață vizibilă care permite accesul la operații, o frontieră impenetrabilă şi datele ascunse. Toate detaliile implementării sunt conținute în cadrul încapsulării şi sunt ascunse clienților care utilizează această abstractizare.

Încapsularea presupune ascunderea informației, aceşti doi termeni fiind utilizați de multe ori ca sinonime. Limbajele de programare precum C++ suportă ascunderea informației în mod nativ, permițând focalizarea unui părți semnificative a activității de proiectare spre asigurarea nivelului adecvat de ascundere a informației.

2.3.3 Abstractizarea datelor şi genericitatea

Abstractizarea datelor reprezintă modul de implementare a încapsulării şi a ascunderii informației la nivelul proiectării şi al programării. Ea reprezintă procesul de creare de tipuri de date definite de utilizator pentru implementarea claselor şi obiectelor lumii reale ce compun sistemul ce se dezvoltă. Aceste noi tipuri de date reprezintă extensii ale tipurilor de date standard ce fac parte dintr-un anumit limbaj de programare, cum ar fi tipurile de date întregi, în virgulă mobilă sau caracter.

Conceptul de abstractizare se bazează pe combinația între abstractizarea procedurală şi abstractizarea prin specificație [LIS86]. O structură de date declarată ca tip şi operațiile necesare pentru a manipula obiecte de acest tip sunt specificate împreună şi grupate într-un modul de program. Operațiile devin parte a noului tip. Modulul reprezintă o încapsulare a tipului şi a operațiilor iar operațiile sunt considerate o interfață prin intermediul căreia poate fi afectat comportamentul obiectelor de acel tip.

Tipul de date abstract (TDA) reprezintă un element de bază pentru definirea claselor şi a obiectelor. El constă din două părți:

- *specificarea sintactică*: conține informații referitoare la nume, domenii şi spații ale valorilor ataşate tipului
- *un set de axiome*: definesc sensul operațiilor prin enunțarea relațiilor dintre ele.

Pentru a descrie un TDA din punct de vedere matematic se vor defini mai întâi următoarele noțiuni:

- *Produsul cartezian* al mulțimilor A şi B este definit ca mulțimea tuturor perechilor de forma (a, b) unde $a \in A$ şi $b \in B$:

$$A \times B = \{(a,b) \mid a \in A,\ b \in B\}$$

BUPT

Produsul cartezian poate fi extins la orice număr de mulţimi

- *O funcţie sau aplicaţie* `f` reprezintă o relaţie între elementele a două mulţimi `A` şi `B`. Mulţimea `A` se numeşte domeniul de definiţie al lui `f` iar `B` se numeşte domeniul de valori al lui `f`. O aplicaţie este o submulţime a produsului `A x B` şi au loc următoarele egalităţi:

```
domeniulDeDefinitie(f) = {a | (∃)b∈B a.î. (a,b)∈f}
domeniulDeValori(f) = {b | (∃)a∈A a.î. (a,b)∈f}
```

O aplicaţie între mulţimile A şi B se notează uneori:

```
f : A → B
```

Aplicaţiile pot fi combinate: dacă `f` aplică mulţimea `A` în mulţimea `B` iar `g` aplică mulţimea `B` în mulţimea `C` atunci aplicaţia `fg` aplică mulţimea `A` în mulţimea `C`.

Fie exemplul unei mulţimi de tipuri abstracte de date. O astfel de mulţime, `Set`, este o colecţie neordonată de obiecte fără chei externe. Elementele unei mulţimi sunt unice, fiind exclusă prezenţa duplicatelor. Presupunem în plus că elementele unei mulţimi aparţin unui tip generic de date, `T`.

Specificarea sintactică a unui TDA este dată de următoarele operaţii:

```
   nou:              → Set
 adaug: Set x T      → Set
elimin: Set x T      → Set
  vida: Set          → Boolean
```

Axiomele mulţimii `Set` stabilesc relaţii între operaţii. De exemplu:

```
(a)  vida(nou) == TRUE
(b)  vida(adaug(s,t)) == FALSE
(c)  adaug(elimin(s,t),t) = elimin(adaug(s,t),t)
```

Axioma (a) afirmă că o mulţime nou creată este întotdeauna vidă. Axioma (b) afirmă că o mulţime nu este niciodată vidă dacă i se adaugă un element. În acest caz `s` este o instanţă a mulţimii `Set` iar `t` este un element de tip `T`. Axioma (c) afirmă că ordinea în care se adaugă şi se elimină un element nou mulţimii este irelevantă, ordinea celor două operaţii poate fi schimbată.

Obiectivul axiomelor este de a genera o definiţie suficient de consistentă şi de completă pentru tipul de date abstract. Tipul de date abstract va fi implementat mai târziu de una sau mai multe clase C++ iar axiomele pot contribui la creşterea robusteţii codului. Axiomele sunt general valabile, astfel că valabilitatea lor nu trebuie testată în cadrul codului.

BUPT

Fiecărei operații aparținând unui TDA i se pot asocia zero sau mai multe precondiții. *Precondiția* reprezintă o constrângere care trebuie specificată în vederea asigurării bunei funcționări a operației respective. Exemple de precondiții pentru mulțimea `Set` pot fi:

- adăugarea unui element: elementul trebuie să nu existe deja în mulțime
- ştergerea unui element: elementul trebuie să existe în mulțime.

Precondițiile sunt deosebit de importante pentru limbajele de programare care se bazează pe conceptul de *programare pe bază de contract*, cum ar fi Eiffel [MEY88]. Prezentarea acestui concept precum şi implementarea sa în cadrul limbajelor de programare ce nu suportă programarea pe bază de contract în mod nativ (cum ar fi C++) a fost abordată în cadrul lucrării „Exception-based Assertion Mechanism for Object-oriented Systems used in the Implementation of a 2d-Editor for a CAD/CAM System", prezentată în cadrul conferinței internaționale CONTI'2000 [FAR2000]. Contractul reprezintă o înțelegere între două părți, furnizorul şi consumatorul unui serviciu. Încălcarea termenilor contractului, de obicei de către consumator, reprezintă cauza apariției erorilor în timpul rulării aplicației.

Proprietățile globale ale instanțelor tipurilor de date abstracte necesare a fi respectate pe întreg parcursul existenței acestora se numesc *invarianți* ai TDA şi pot fi implementați în cadrul limbajelor orientate spre obiecte sub denumirea de *invarianți ai claselor*. Un exemplu de invariant al unui TDA este faptul că numărul de elemente al unei mulțimi trebuie să fie nenegativ.

Încapsularea şi ascunderea informațiilor poate fi exemplificată pe baza clasei `Stiva` prezentată anterior. Clasa Stiva poate fi considerată un tip de date abstract. Acesta reprezintă un tip de date definit de utilizator ce va fi recunoscut de un compilator C++. Cuvântul cheie `private` este utilizat pentru implementarea structurilor de date ascunse iar cuvântul cheie `public` este utilizat pentru a defini operațiile care permit acces la structurile de date. Această specificație a clasei `Stiva` defineşte structura tipului de date abstract precum şi interfața (contractul) acestuia în relația client-server.

Una dintre caracteristicile unui TDA este faptul că trebuie să încapsuleze o singură abstractizare. Interfața unui TDA care încapsulează mai multe abstractizări ar mări gradul de cuplare între client şi server mărind astfel cantitatea de cod ce trebuie rescris în cazul unor modificări ale cerințelor sistemului.

BUPT

Suportul direct al limbajului de programare pentru tipurile de date abstracte reprezintă o cerinţă foarte importantă pentru implementarea bazată pe tehnica orientării spre obiecte. Această trăsătură a limbajului poate reprezenta un factor semnificativ în crearea de module software de mari dimensiuni care să satisfacă cerinţele de cuplare strânsă în cadrul modulelor şi de cuplare redusă între module.

În practică se întâlneşte frecvent cazul aplicării unui TDA într-o implementare C++. TDA-ul depinde de un tip de dată generic, T, astfel că implementarea va fi sub forma unei clase *template* (şablon), tip de clasă relativ nou în C++. Trecerea de la TDA la clase (şi funcţii *template*) este exemplificată în cadrul următorilor paşi:

- TDA devine o clasă *template* cu acelaşi nume.
- Pentru implementarea altor tipuri care apar în cadrul TDA se va alege un tip de date C++.
- Operaţiile din cadrul TDA devin interfaţă în C++.
- Structura clasei *template* trebuie să corespundă tipurilor de date din TDA.
- Precondiţiile se implementează în corpul funcţiilor membre.
- Violările precondiţiilor şi a invarianţilor trebuie să se concretizeze în excepţii care vor fi rezolvate corespunzător.

Un exemplu simplu pentru un TDA îl reprezintă vectorii unidimensionali ai căror elemente sunt de tipul generic `T`. Acest TDA va fi denumit `Vector<T>`, `<T>` specificând faptul că elementele vectorului sunt de tip `T`. Fiecare funcţie sau operaţie asociată TDA-ului are un *nume*, un set de tipuri de intrare numit *domeniu de definiţie* şi eventual un tip de date returnat din *domeniul de valori*. Fie exemplul funcţiei de indexare:

```
index: Vector x Întreg → T
```

Implementarea acestei funcţii în C++ poate fi următoarea:

```
T index(const Vector<T>& vect, size_t size);
```

Un singur tip de date abstract poate conduce la obţinerea mai multor implementări. Relaţia dintre TDA şi implementarea sa în C++ este de tip „unu-la-mulţi" (*one-to-many*).

Setul de funcţii asociate clasei `Vector<T>` este:

```
nou      : Integer            → Vector
numără   : Vector             → Integer
index    : Vector x Integer   → T
copiază  : Vector             → Vector
```

BUPT

Precondiţia asociată funcţiei `index` afirmă că valoarea întreagă utilizată pentru accesarea unui element al masivului trebuie să se afle între limitele de indexare ale acestuia.

Aplicarea regulilor de mai sus pentru TDA-ul `Vector<T>` alegând pentru implementarea tipului generic `Întreg` tipul `size_t` rezultă în următoarea declaraţie *template* în C++:

```
template <class T> class Vector
  {
  private:
    T       *val;
    size_t  dim;
  public:
    Vector(size_t dim);
    size_t numara() const;
    T index(size_t i) const;
    Vector copiaza() const;
    size_t index_min() const;
    size_t index_max() const;
  };
```

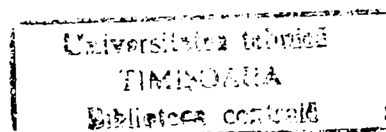
Conceptul de *genericitate* se referă la posibilitatea de a crea entităţi software cu tipuri parametrizate care pot fi instanţiate cu diferite tipuri în faza de compilare. Ca urmare rezultă clase parametrizate sau generice. Clasa generică este creată sub forma unui *template* având parametri formali iar instanţe ale clasei sunt create specificând parametrii actuali corespondenţi. Acest concept reprezintă cel mai înalt grad de reutilizabilitate, aceeaşi clasă poate fi folosită cu diferite tipuri fără a face uz de moştenire. Structura unei clase generice este mai simplă decât ierarhiile de clase utilizate în contextul conceptului de moştenire.

Pentru exemplul clasei generice `Vector` prezentată mai sus tipul parametrizat este `T`, fiind posibilă declararea de diverse tipuri de vectori:

```
Vector<int>      intVector;     //vector de elemente de tip întreg
Vector<float>    floatVector;   //vector de elemente de tip float
Vector<complex>  complexVector; //vector de numere complexe
Vector<char>     charStack;     //vector de elemente de tip char
```

2.3.4 Responsabilităţi

O clasă C++ poate fi privită ca şi o specificaţie a unei reprezentări client-server. Interfaţa vizibilă a clasei include un număr de operaţii care reprezintă un server pentru un număr de clienţi care vor utiliza operaţiile asociate diferitelor instanţe (obiecte) ale clasei. Unele dintre operaţiile clasei pot fi ascunse clienţilor şi vor fi utilizate doar de către obiectul însuşi. Operaţiile (funcţiile



631.903
369 B

BUPT

membre în C++) asociate unui obiect determină comportamentul obiectului, deci sunt referite ca *responsabilități* ale acestuia.

O abordare a perspectivei client-server este prezentată de către Booch în [BOO93 p.43]. Interfața vizibilă a clasei pune la dispoziție contractul pentru clienți. Acest contract conține responsabilitățile obiectului, adică comportamentul pentru care poate fi făcut responsabil. Conceptul de responsabilitate este legat în mod direct de una dintre metodele de proiectare orientată spre obiecte ce utilizează cartelele CRC (*Class Responsability Collaboration Cards*), metodă ce va fi prezentată mai târziu.

2.3.5 Colaborări și transmiterea de mesaje

O *colaborare* poate fi definită ca și un serviciu pus la dispoziție de un obiect pentru a fi utilizat de alte obiecte. Împreună cu colaborările, obiectele definesc comportamentul întregului sistem.

Colaborările sunt strâns legate de noțiunea de *transmitere de mesaje*, aceasta reprezentând o descriere a modului în care comunică între ele modulele software. Modul de comunicare între module bazat pe partajarea datelor nu va fi analizat în actualul context.

Conceptul de transmitere a unui mesaj de la un obiect la altul este implementat de obicei ca o combinație de transmitere de parametri și de returnare de valori ca rezultate. În C++ parametrii sunt transmiși prin valoare sau prin referință în cadrul unui apel de funcție. Valorile sunt returnate de către funcții prin intermediul instrucțiunii `return` sau prin modificarea variabilelor transmise ca parametri prin referință.

Transmiterea de mesaje în C++ se realizează prin efectuarea de apeluri asupra funcțiilor membre vizibile declarate în specificația clasei. Transmiterea unui mesaj pentru obținerea valorii unei rădăcini pătrate declarate în cadrul unei biblioteci de clase matematice poate fi realizată prin apelul de funcție:

```
Y = Math::Sqrt(X);
```

Direcția de transmitere a unui mesaj poate fi indicată în cadrul unui graf orientat, după cum rezultă din figura următoare:

BUPT

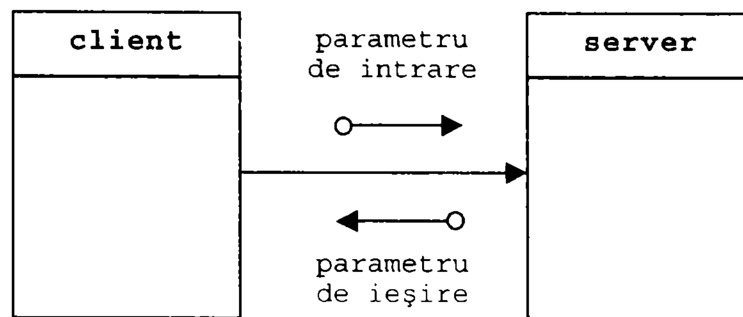


Figura 2.2 Direcția de transmitere a mesajelor

Datele transmise ca parametri pot circula în ambele direcții. Modul în care sunt returnate datele depinde de limbajul de programare utilizat pentru implementare.

Discuția de până în acest moment s-a concentrat asupra transmiterii de mesaje prin apel de metode ale obiectelor care sunt executate secvențial. Pentru sisteme în timp real trebuie avute în vedere colaborările şi transmiterea de mesaje în medii multiprocesor.

2.3.6 Moştenirea

Conceptul de moştenire este utilizat pentru a descrie crearea de subclase pornind de la una sau mai multe superclase. Subclasa moşteneşte structura şi comportamentul superclasei, inclusiv atributele şi operațiile acesteia. Moştenirea multiplă face posibil ca o subclasă să moştenească mai multe superclase.

O exemplificare a relațiilor de moştenire este reprezentată grafic în figura de mai jos, săgețile indicând spre părinte.

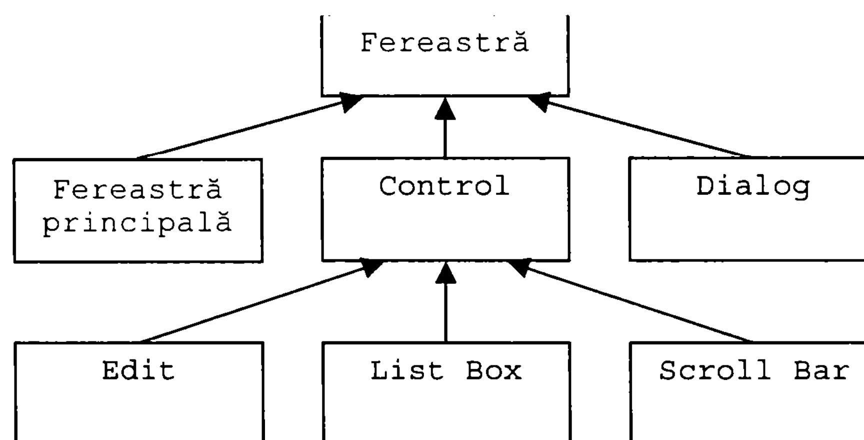


Figura 2.3 Relații de moştenire

BUPT

Atributele şi operaţiile superclasei sunt moştenite de fiecare subclasă ca şi o submulţime. Fiecărei subclase i se pot adăuga caracteristici (atribute şi comportament) ce sunt unice acesteia. Săgeţile din figura de mai sus denotă relaţii de tipul "is-a" sau "kind-of" [BOO93]. O subclasă poate fi la rândul ei superclasă pentru alte clase situate mai jos în arborele de moştenire.

Conceptul de moştenire, permiţând redefinirea de clase derivate pornind de la clase de bază, reprezintă un instrument pentru organizarea şi construirea de clase reutilizabile bazate pe module existente. Dacă operaţii din cadrul unor clase aflate în partea superioară a ierarhiei sunt modificate, aceste modificări sunt automat moştenite de clasele derivate. În cazul în care compilatorul utilizat nu suportă moştenirea, fiecare clasă va trebui dezvoltată conform unei metode „bottom-up" ca şi entitate de sine stătătoare, consistenţa fiind asigurată pe baza disciplinei in programare şi nu pe baza verificărilor de tip şi de interfaţă efectuată de către un compilator care suportă moştenirea.

Efectul net al utilizării moştenirii este de reducere prin reutilizare a cantităţii de cod ce trebuie dezvoltat. Faza de depanare poate deveni mai anevoioasă în cazul în care nu dispunem de un browser capabil de navigare prin nivelurile ierarhiilor de clase.

2.3.7 Polimorfismul

Polimorfismul se traduce prin faptul că un obiect poate să existe în timpul rulării ca şi instanţă a diferitor clase. În cadrul unui limbaj de programare care implementează polimorfismul operaţiile au asociate atât tipuri statice cât şi tipuri dinamice iar referinţa la un tip dinamic poate să se modifice în timpul execuţiei.

Figura următoare prezintă o ierarhie de clase pentru a ilustra conceptul de polimorfism.

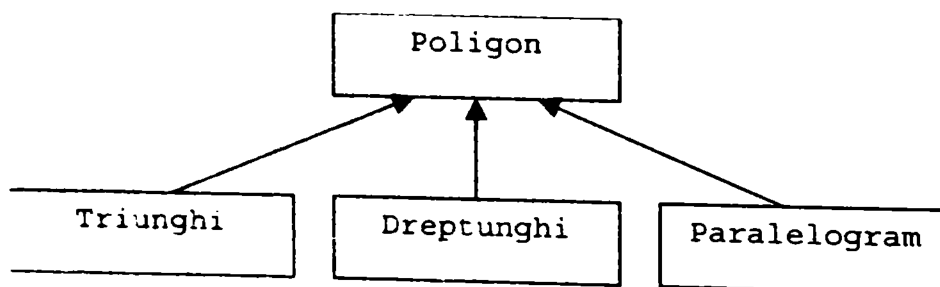


Figura 2.4 Ierarhie de clase

Poligonul reprezintă superclasa iar triunghiul, dreptunghiul şi paralelogramul reprezintă subclasele.

BUPT

Superclasa Poligon poate fi implementată în C++ după cum urmează:

```
class Poligon    //superclasa
  {
  public:
    float Perimetru();
    points Translatie();
    virtual float Aria();
    virtual boolean punctInInterior();
  protected:
    void Display();
  };
```

Clasa dreptunghi este derivată public din clasa poligon, implementare ce corespunde relației de tipul "kind-of" menționată anterior:

```
class Dreptunghi: public Poligon //clasa derivată
  {
  public:
    virtual float Aria();
    virtual boolean punctInInterior();
    void setDreptunghi();
  private:
    float Top,
          Bottom,
          Left,
          Right;
  };
```

Implementarea unei funcții membre pentru calculul ariei dreptunghiului poate fi următoarea:

```
float Dreptunghi::Aria()
{
  return((Bottom - Top) * (Right - Left));
};
```

Pentru paralelogram şi pentru triunghi se vor specifica implementări diferite. Trei instanțe diferite ale aceleiaşi funcții membre pot deci să se refere la clase diferite.

Polimorfismul poate fi reprezentat şi sub forma referinței polimorfice, deci a unui tablou de elemente ce referă obiecte de tipuri diferite [MEY88] derivate din aceeaşi superclasă, după cum se arată în figura următoare.

BUPT

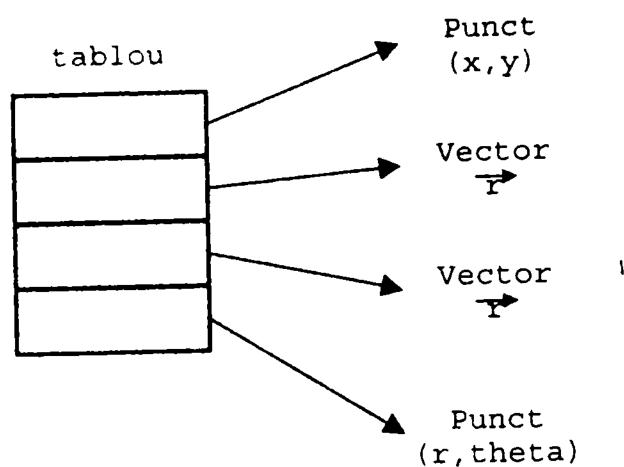


Figura 2.5 Referință polimorfică

Tabloul conține pointeri spre diferite tipuri de puncte care sunt toate derivate din aceeaşi superclasă. În cazul unui limbaj de programare care nu suportă polimorfismul structura de mai sus va trebui implementată utilizând o structură ale cărei elemente va referi diverse tipuri de puncte. În acest caz referințele sunt statice, pe când în cazul polimorfismului referințele pot fi dinamice.

2.3.8 Legarea

Conceptul de legare (binding) utilizat în contextul unui limbaj de programare se referă la legarea unui mesaj (de exemplu apelul unei operații asociate unui obiect) de codul ce se execută în legătură cu acest mesaj. Legarea poate fi:

- *statică*: dacă toate referințele sunt determinate în momentul compilării
- *dinamică*: dacă codul executat în legătură cu un mesaj se cunoaşte doar în momentul rulării.

În cadrul limbajului C++, cuvântul cheie `virtual` se referă la legarea dinamică.

Moştenirea, polimorfismul şi legarea sunt strâns legate una de cealaltă iar execuția unui apel al unei operații este asociată unei referințe polimorfice care depinde de tipul referinței. În timpul execuției programului C++ prezentat mai sus, referința adecvată la o funcție membră `Aria()` va fi efectuată în momentul rulării. Asocierea clasei poate fi efectuată în faza de compilare sau în momentul rulării. Utilizarea cuvântului cheie `virtual` informează compilatorul că asocierea va fi efectuată doar în momentul rulării. Atunci când cuvântul cheie `virtual` este omis, se presupune că referințele sunt rezolvate în faza de compilare.

BUPT

2.3.9 Modularitatea

Modularitatea reprezintă o măsură a "împachetării" (agregării) unui sistem în entități software. Un grad ridicat de modularitate are ca efect ameliorarea gestionabilității și reducerea complexității, aspecte dezirabile mai ales în contextul sistemelor mari. Entități reprezentate sub formă de module pot consta din:

- tipuri de date abstracte
- operații individuale
- biblioteci complete
- programe necesare creării unei imagini executabile

Entitățile software ce se împachetează se regăsesc sub denumirea de *module* și datorită faptului că ele reprezintă entități compilabile.

Conceptul de modul implică posibilitatea de a identifica o frontieră în jurul acestuia. Un deziderat de ordin general este de a crea module care să implementeze încapsularea și ascunderea informației. Cele două principii urmărite în cadrul dezvoltării modulelor sunt coeziunea puternică în cadrul modulului și grad redus de cuplare între module.

Diferența majoră între conceptul de împachetare utilizat în tehnologia structurată și cel utilizat în cadrul tehnologiei orientate spre obiecte este legată de faptul că în cazul din urmă se lucrează cu tipuri de date abstracte. Modulele bazate pe tehnologia structurată conțin de obicei un set de operații și structurile de date asociate. Aceste structuri de date nu pot fi ascunse, fiind deci vizibile în exterior. Rezultă astfel o cuplare strânsă între acea structură de date și codul care interacționează cu ea.

În cadrul limbajului C++ definiția unui tip de date abstract se plasează într-un fișier header având extensia „`.h`". Acest fișier este importat prin intermediul directivei de preprocesor `#include` și reprezintă specificația (contractul) pentru programul client care utilizează tipul de date abstract. Implementarea ascunsă a tipului de date abstract este conținută într-un fișier omonim cu extensia „`.cpp`".

Importanța majoră a modularității este legată de atingerea unui nivel înalt de reutilizare a codului în cadrul aplicațiilor bazate pe tehnologia orientată spre obiecte.

BUPT

2.4 Clase şi obiecte

Clasele şi obiectele reprezintă conceptele cele mai importante ale tehnologiei orientate spre obiecte. Aceste entităţi reprezintă baza pentru partiţionarea sistemelor mari precum şi o măsură a modului de realizare a componentelor software reutilizabile prin tranziţionarea entităţilor create in etapa de proiectare în entităţi de programare.

În continuare se va prezenta o definiţie formală a claselor şi a obiectelor descriind modul de descompunere a sistemelor mari utilizând paradigmele de moştenire, agregare şi asociere.

2.4.1 Definiţii

Definiţiile prezentate în cele ce urmează se referă la două din cele mai importante elemente ale orientării spre obiecte, clasele şi obiectele. Definirea obiectelor o va preceda pe cea a claselor pentru a facilita definirea acestora în încercarea de a evita definiţiile ciclice.

2.4.1.1 Obiectele

O definiţie uzuală a obiectului îl asociază unei entităţi tangibile care poate fi percepută vizual sau conceptual. Există obiecte cu frontieră distinctă şi obiecte cu frontieră vagă. Unui obiect îi pot fi asociate următoarelor caracteristici:

- Un obiect este identificat de un *nume unic*.
- Un obiect are asociate *stări*. Aceste stări definesc în totalitate proprietăţile unui obiect şi valorile acestor proprietăţi pe durata de existenţă a obiectului în interiorul frontierelor sistemului.
- Unui obiect îi este asociat un *set de operaţii*. Comportamentul obiectului în cadrul sistemului este descris de aceste operaţii. Aceste operaţii sunt utilizate pentru a afla sau pentru a schimba starea obiectului sau pentru a facilita comunicarea între obiecte. Operaţiile se regăsesc şi sub denumirea de *metode*. În context independent de limbaj ele vor fi denumite *operaţii* iar în C++ ele vor fi referite ca *funcţii membre.*
- Un obiect are asociat un *set de atribute*. Acestea conţin valori de stare pe parcursul existenţei obiectului în cadrul sistemului. Ele sunt implementate uzual sub formă de variabile ascunse ale abstractizării pe care o reprezintă obiectul.

BUPT

Setul de caracteristici prezentat mai sus nu este suficient pentru a identifica un obiect în procesul de dezvoltare. Un obiect ia diverse forme pe parcursul ciclului de dezvoltare şi este referit ca "obiect" la diferite nivele şi în diverse tehnologii de dezvoltare. În faza de analiză a domeniului o abstractizare de nivel înalt poate fi referită ca obiect al lumii reale iar o clasă C++ reprezentând o entitate OOD/OOP poate fi referită de asemenea ca obiect.

Un concept important în ceea ce priveşte obiectele este legat de relaţiile care există între acestea. Un obiect interacţionează cu alt obiect prin operaţiile asociate acestuia. Un obiect transmite un *mesaj* unui alt obiect apelând o operaţie vizibilă a acestuia. Direcţia de transfer a datelor poate fi unidirecţională sau bidirecţională. Acest tip de relaţie este definit ca *legătură* (*link*) între două obiecte [RUM91]. Un obiect poate fi deci privit fie ca şi client ce utilizează operaţiile altor obiecte fie ca server care pune la dispoziţie operaţii pentru alte obiecte.

2.4.1.2 Clasele

Există o strânsa legătură între clase şi obiecte, astfel că definiţia unei clase poate fi efectuată în termeni ai obiectului. Clasele reprezintă abstractizări care nu există în formă concretă, pe când obiectele reprezintă entităţi concrete. Ele au o perioadă finită de existenţă în cadrul sistemului care le-a creat şi le utilizează. Definirea unei clase se poate face pe baza următorului set de caracteristici:

- O clasă reprezintă un *nivel înalt de abstractizare*, fiind utilă pentru descrierea unui set de caracteristici comune unor entităţi de analiză şi proiectare.
- O clasă reprezintă o *colecţie de obiecte.* Obiectele unei clase partajează un set comun de atribute şi de operaţii.
- *Un obiect reprezintă o instanţă specifică a unei clase.* Un obiect este un membru al unei anumite clase şi are un set de atribute şi de operaţii asociate acelei clase.
- *O clasă poate face parte dintr-o ierarhie.* Fiecare clasă poate face parte dintr-o relaţie părinte-copil, ierarhia fiind reprezentată de conceptul de moştenire. O clasă (subclasă) poate fi derivată dintr-o clasă părinte (superclasă) şi moşteneşte atributele şi operaţiile acesteia.

2.4.1.3 Rolul claselor şi al obiectelor în analiza şi proiectarea orientată spre obiecte

Clasele şi obiectele au rolul de a descompune un sistem software în abstractizări ale lumii reale care pot forma baza analizei şi a proiectării sistemului. În faza de analiză a domeniului şi în

BUPT

faza de analiză a cerinţelor sistemului se determină obiecte de nivel înalt şi clasele asociate acestora.

În faza de proiectare obiectele rezultate în faza de analiză sunt rafinate. Se adaugă detalii prin identificarea atributelor şi a operaţiilor reutilizabile şi se construiesc interfeţe adecvate. În faza de proiectare a aplicaţiei sunt identificate de obicei noi clase şi obiecte. Arhitectura rezultată facilitează construcţia scheletului aplicaţiei ce va fi implementat utilizând limbajul de programare ales.

Un rol important al claselor şi al obiectelor este de a pune la dispoziţie interfeţele necesare accesului la atributele asociate instanţelor clasei. Accesul se realizează prin intermediul operaţiilor vizibile declarate pentru fiecare clasă. Este important ca atributele şi operaţiile să fie asociate instanţelor (obiectelor) unei anumite clase şi nu clasei înseşi. Metodele declarate în interfaţa clasei efectuează anumite acţiuni asupra fiecărei instanţe ale acesteia şi nu asupra clasei în sine. Atributele declarate în specificaţia clasei se numesc *variabile ale instanţei* şi memorează stări ale unei anumite instanţe. Pentru atribute care sunt comune tuturor instanţelor unei anumite clase vom defini *variabile ale clasei*. Anumite limbaje de programare suportă în mod direct acest tip de variabile. În cadrul limbajului C++ acest tip de variabile este implementat cu ajutorul cuvântului cheie `static`.

2.4.2 Crearea claselor

Punctul de plecare în procesul de creare a claselor îl reprezintă modul în care pot fi grupate cerinţele sistemului într-un număr de categorii de clase. O clasă reprezintă un server pentru clienţii săi, fapt care face necesară definirea responsabilităţilor clasei.

Procesul de creare a claselor poate avea ca bază un set de cerinţe ale sistemului ţinând cont de existenţa următoarelor categorii de clase:

- *Clase interfaţă dispozitiv*: reprezintă categoria de clase ce interfaţează direct dispozitivele hardware
- *Clase sistem extern*: reprezintă categorii de clase specifice sistemelor care sunt interfaţate cu alte sisteme similare.
- *Clase interfaţă utilizator*: reprezintă categorii de clase utilizate în interfaţa sistemului cu utilizatorul. Ele pot fi divizate în categorii speciale pentru fiecare tip de dispozitiv de interfaţă.

BUPT

- *Clase computaționale*: reprezintă categorii de clase specifice sistemelor pentru care îndeplinirea cerințelor sistem presupune utilizarea unei componente computaționale semnificative.
- *Clase rol*: pot fi utilizate pentru modelarea situațiilor în care o persoană (sau un rol) reprezintă o componentă importantă a unui sistem.
- *Clase abstractizări de date*: sunt clase care nu apar ca atare în definirea cerințelor sistem, dar apar în utilizarea modelării datelor în strategia de dezvoltare. Diagramele de relații între entități utilizate în timpul analizei domeniului conțin obiecte ale lumii reale care pot fi uneori grupate în clase abstractizări de date.

Un alt element fundamental în crearea de clase este reprezentat de stabilirea corectă a interfeței sale ca server care pune la dispoziție servicii pentru clase client. Următoarea clasificare a operațiilor necesare manipulării instanțelor unei clase [LIP91] poate constitui un punct de plecare pentru definirea corectă a interfeței unei clase:

- *Operații de gestiune*: sunt operații care controlează crearea, inițializarea, distrugerea instanțelor unei clase, asignarea unui obiect altui obiect, gestiunea memoriei, conversii de tip. În majoritatea situațiilor aceste operații sunt apelate automat şi fac parte din interfața vizibilă a unei clase.
- *Operații de implementare*: colecția acestor operații implementează cerințele funcționale ale abstractizării reprezentate de clasă.
- *Operații auxiliare*: aceste operații sunt utilizate la implementarea primelor două categorii de operații şi de obicei nu sunt vizibile obiectelor client.
- *Operații de acces*: facilitează obiectelor client obținerea de informații referitoare la atributele ascunse, încapsulate în structura unei clase. Operațiile de acces sunt utilizate şi pentru a modifica valorile de stare ale atributelor unei clase.

2.4.3 Moştenirea şi ierarhiile de clase

Moştenirea reprezintă o formă de generalizare-specializare ce are ca rezultat crearea unei ierarhii de clase conținând entități de nivel scăzut şi specializare superioară ca modalitate de descompunere a unui sistem mare în părți mai mici, gestionabile.

Fiind dată o clasă A se poate construi specializarea B a acesteia care va conține toate proprietățile şi metodele clasei A la care se adaugă un set de proprietăți şi metode proprii. Se

BUPT

spune că există o relaţie de moştenire între clasa A şi clasa B, relaţie care poartă denumirea de relaţie "is-a" ("ESTE"). Se spune că B este o specializare a lui A.

Figura 2.3 ilustrează această idee pe baza descompunerii ierarhice a clasei `Fereastră`. Nivelul cel mai de jos conţine versiuni specializate ale ferestrei iar următorul nivel conţine versiuni specializate ale controlului.

Ierarhia rezultată din fazele OOA şi OOD poate fi implementată direct în faza OOP utilizând un limbaj de programare ce suportă moştenirea, cum ar fi C++.

Crearea unei ierarhii de clase împreună cu paradigma de moştenire asociată reprezintă doar una dintre metodele de relaţionare a claselor în cadrul sistemelor complexe de dimensiuni mari. Celelalte două metode, agregarea şi asocierea vor fi descrise în cele ce urmează.

2.4.4 Agregarea

Agregarea stabileşte relaţia dintre un obiect şi componentele sale în cadrul unui sistem compus din clase care nu sunt relaţionate în cadrul unei ierarhii de generalizare-specializare ci prin intermediul unor legături de tipul "părţi componente". Agregarea reprezintă de asemenea o formă de ierarhie pe baza căreia pot fi construite clase, relaţia generatoare fiind de tipul "has-a" ("are ca parte").

Relaţia de agregare este ilustrată în figura următoare pentru câteva din componentele unui avion.

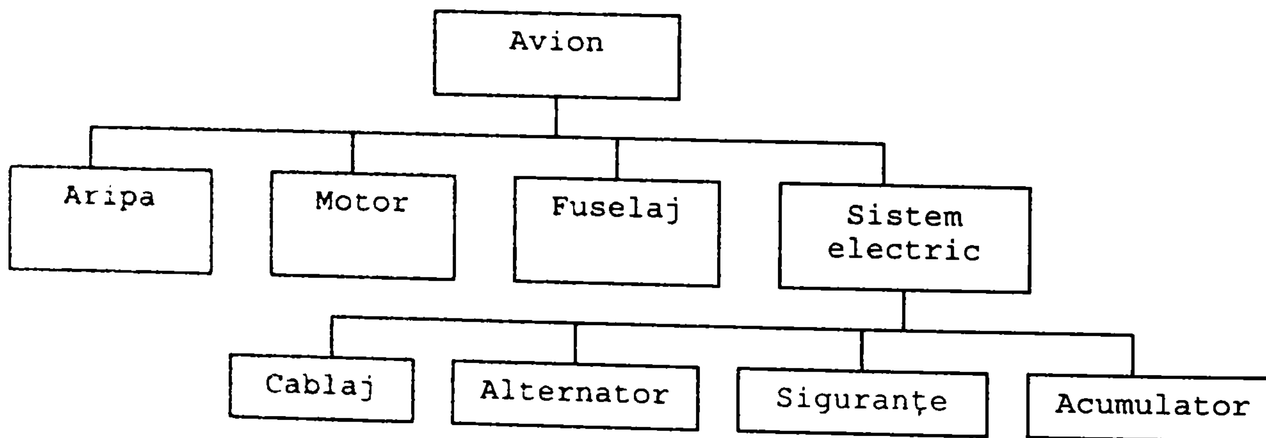


Figura 2.6 Relaţii de agregare

Noţiunea de agregare se mai regăseşte şi sub denumirea de compoziţie.

Următoarea porţiune de cod prezintă implementarea relaţiilor de agregare ilustrate în cadrul figurii de mai sus utilizând limbajul C++.

BUPT

```
class Aripa
{
   ...
};

class Motor
{
   ...
};

class Avion
{
   Aripa aripaStanga,
         aripaDreapta;
   Motor motorReactie;
   ...
};
```

2.4.5 Asocierea

Asocierea se referă la comunicaţia sau dependenţa între obiecte ale diferitor clase care nu sunt legate prin relaţii de tipul moştenirii sau agregării. Asocierea defineşte rolurile sau dependenţele între obiecte aparţinând la două clase şi cardinalitatea acestora, deci numărul de instanţe ale fiecărei clase care pot fi implicate în asociere. Fiecare asociere formează o dependenţă bidirecţională, atâta timp cât nu se specifică restricţionarea la unidirecţionalitate.

Există trei tipuri de cardinalitate a unei asocieri, ilustrate şi în cadrul figurilor următoare:

- unu la unu
- unu la mulţi (mulţi la unu)
- mulţi la mulţi

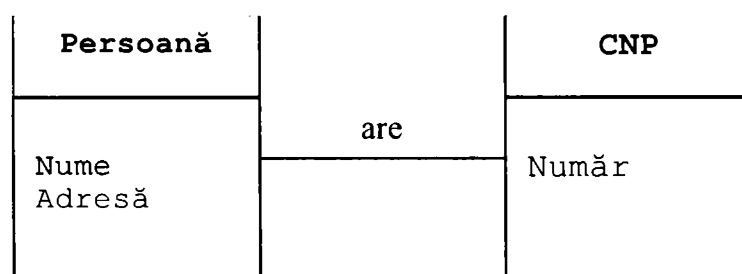


Figura 2.7 Asociere unu la unu

Figura 2.7 ilustrează asocierea „unu la unu" pentru obiecte ale claselor Persoană şi CNP. Fiecare persoană are un cod numeric personal unic iar un cod numeric personal se referă la un individ unic. Relaţia dintre cele două obiecte este bidirecţională. Relaţia directă se citeşte de la

BUPT

stânga la dreapta (o Persoană "are" un CNP) iar relația inversă de la dreapta la stânga (un CNP "identifică" o persoană).

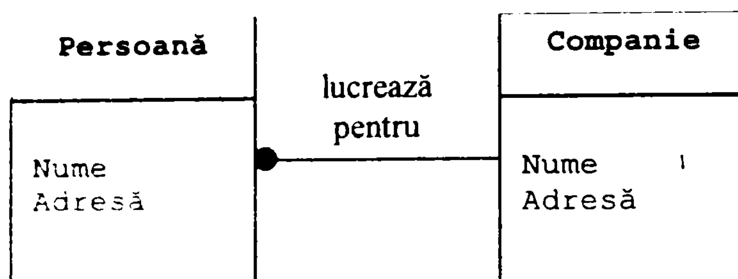


Figura 2.8 Asociere unu la mulți

Asocierea „unu la mulți" este ilustrată în cadrul figurii de mai sus pentru o `Persoană` și `Compania` pentru care lucrează aceasta. Asocierea prezentată mai sus presupune că o persoană lucrează pentru o singură companie.

Pentru ilustrarea asocierii de tipul „mulți la mulți" se va considera cazul obiectului `Punct` ce rezultă la intersecția a cel puțin două obiecta `Dreaptă`. Dreapta poate la rândul ei să treacă prin mai multe astfel de puncte. Notația utilizată în cadrul figurii este specifică metodei OMT descrise în [RUM91]. Specificația 2+ atașată capătului stâng al conexiunii se referă la faptul că există cel puțin două obiecte, iar capătul nemarcat al conexiunii semnifică oricâte obiecte. Notațiile referitoare la cardinalitatea unei asocieri sunt specifice fiecărei metode OOA/OOD.

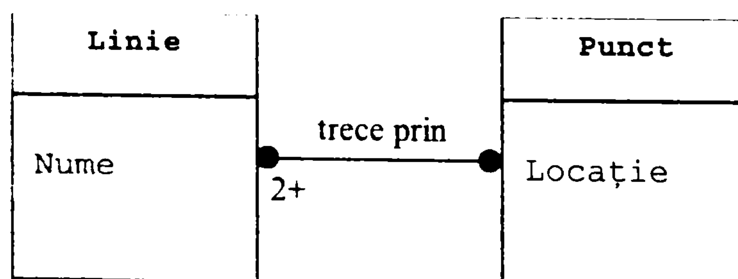


Figura 2.9 Asociere mulți la mulți

2.4.6 *Utilizarea*

Relația de "utilizare" între clase și obiecte semnifică faptul că se face o referință spre o clasă sau un obiect situat în afara abstractizării curente. Un exemplu al acestei relații se regăsește în clasa ATM care utilizează o clasă externă Cont:

```
extern class Cont;
```

BUPT

```
Cont contUtilizator(...);

class ATM
{
   ...
   float Balanta(const char *nrCont);
   ...
};
...
float ATM::Balanta(const char *nrCont)
{
   ...
   return contUtilizator.detBalanta(nrCont);      //utilizare
}
```

Relația de utilizare poate fi abordată și prin perspectiva client-server [BOO93]. În exemplul de mai sus clasa `ATM` reprezintă clientul iar clasa `Cont` reprezintă serverul. Relația de utilizare reprezintă un element important în cadrul tehnologiei orientate spre obiecte, unele dintre metodele OOA/OOD utilizând notații speciale pentru această relație.

2.4.7 Clase „Mixin"

O clasă „Mixin" [COP92, BOO93] reprezintă o clasă independentă ce nu face parte dintr-o ierarhie, deci nu are o superclasă. O utilizare tipică a claselor „Mixin" este legată de moştenirea multiplă în cadrul ierarhiilor de clase în care există căi convergente [RAD98]. Acest caz se întâlneşte când o clasă moşteneşte două sau mai multe clase care la rândul lor moştenesc o clasă ascendent comun. În acest caz, clasa „Mixin", intercalată în această ierarhie, contribuie la simplificarea structurii de clase.

Una dintre problemele legate de utilizarea moştenirii multiple este percepută în faza de depanare, când identitatea unui obiect este dificil de determinat. Utilizarea claselor „Mixin" creează un singur nivel în ierarhia clasei derivate şi poate reduce considerabil eforturile legate de depanare.

2.4.8 Clase container

Clasele container [BOO93] sunt utilizate pentru a grupa colecții de obiecte, cum ar fi stive, liste, mulțimi, cozi, etc. O clasă container omogenă [MUR93] conține obiecte de acelaşi tip iar o clasă container eterogenă conține obiecte de tipuri diferite. În C++ clasele eterogene sunt derivate dintr-o clasă de bază comună, accesul la obiecte fiind simplificat prin intermediul funcțiilor

BUPT

membre ale clasei de bază. Clasele container omogene sunt implementate în C++ de obicei prin intermediul template-urilor.

2.4.9 *Metaclase*

O metaclasă reprezintă o clasă de cel mai înalt nivel, din care pot fi create clase noi ca instanțe ale ei. O clasă derivată dintr-o metaclasă este un obiect. Importanța metaclaselor este legată de tratarea claselor de nivel înalt în cadrul unei ierarhii și de crearea de variabile clasă. Clasele derivate de nivel inalt pot fi manipulate la fel ca și alte obiecte [BOO93].

Conceptul de metaclasă nu este suportat în mod direct în C++, dar variabile ale clasei pentru o metaclasă pot fi simulate prin intermediul cuvântului cheie `static`. Membrii statici sunt partajați de către toate instanțele unei "metaclase", deci există o modalitate de a crea variabile comune care pot fi utilizate sau modificate de către toate obiectele aparținând aceleiași clase. Un exemplu de utilizare a unei variabile a clasei este de a păstra un contor al instanțelor create dintr-o clasă dată, procedeu denumit în unele cazuri numărare a referințelor[COP92,p.58].

2.5 Modele utilizate în analiza și proiectarea orientată spre obiecte

Generalizarea elementelor de tehnologie obiectuală specifice programării orientate spre obiecte spre celelalte faze ale ciclului de dezvoltare a unui produs software are ca efect crearea unei mulțimi de obiecte. Unele dintre acestea reprezintă pure abstractizări care vor exista doar în faza de analiză și de proiectare pe când altele vor fi implementate efectiv ca instrucțiuni în cadrul limbajului de programare utilizat. Este imperativ necesar ca diversele faze ale procesului de dezvoltare să fie caracterizate de o abordare coerentă și unitară, care să cuprindă atât nomenclatura utilizată cât și instrumentele utilizate în fazele de dezvoltare. Instrumentele trebuie să pună la dispoziție facilități de ilustrare grafică a deciziilor luate și elemente de euristică necesare tranziției de la o etapă a dezvoltării la următoarea.

Un dezavantaj al metodele de analiză și proiectare obiectuală consacrate constă în lipsa de unitate in ceea ce privește notațiile utilizate pentru a desemna obiectele etapelor de analiză și proiectare și relațiile existente între acestea în diversele faze ale dezvoltării produsului software. O incercare de unificare a fost prezentată [KOR90], dar nu a fost adoptată de diversele tehnici și instrumente OOA/OOD existente pe piață. Un demers mai recent în ceea ce privește unificarea acestor notații este legat de dezvoltarea limbajului UML de către Object Management Group [OMG99]. Versiunile actuale ale instrumentelor software dedicate analizei și proiectării orientate

BUPT

spre obiecte, cum ar fi Rational Rose [QUA98] [KRU98], OTW [BUR99], prezintă ca avantaj suportul pentru limbajul UML.

2.5.1 Modele orientate spre obiecte

Mecanismul de bază al tuturor metodelor de analiză şi proiectare orientate spre obiecte utilizează un set de modele de proiectare. Modelul de proiectare este definit în [LIA98] ca fiind o reprezentare a artefactului ce urmează a fi proiectat:

$$S = [o_i, .]$$

unde S reprezintă modelul de proiectare constând din unul sau mai multe obiecte de proiectare o_i,., notaţia $o_{i,j}$ desemnând obiectul de proiectare i cu atributul j.

Modelele utilizate în cadrul metodelor obiectuale descriu aspecte statice, dinamice şi funcţionale ale entităţilor lumii reale ce au rezultat din definirea cerinţelor sistemului, relaţiile dintre ele şi structurile arhitecturale corespunzătoare acestora.

Există patru tipuri de modele ce pot fi utilizate în cadrul metodelor orientate spre obiecte:

- modele de analiză (modele OOA)
- modele de proiectare (modele OOD)
- modele de clase
- modele de obiecte

Aceste modele pot fi utilizate atât în cadrul etapei de analiză cât şi a etapei de proiectare. Avantajul acestui fapt constă în faptul că pot fi utilizate aceleaşi notaţii şi instrumente grafice pentru ambele etape. Dezavantajul constă în faptul că delimitarea între cele două faze pierde din consistenţă. În esenţă, delimitarea primară între modelele de analiză şi modelele de proiectare constă în faptul că obiectele de analiză şi relaţiile dintre acestea se caracterizează printr-un nivel mai ridicat de abstractizare, obiectele de proiectare conţinând mai multe informaţii legate de interfaţă şi de implementare.

Două dintre cele mai importante categorii de modele ce apar sub o formă sau alta în cadrul tuturor metodelor de analiză şi proiectare orientate spre obiecte sunt modelele statice şi modele dinamice. Trecerea în revistă a modelelor statice şi dinamice ce urmează va utiliza notaţiile specifice metodei de analiză şi proiectare OMT (Object Modelling Technique) [TEX97], [RUM91].

BUPT

2.5.2 Modele statice

Modelele statice reprezintă relaţiile între obiectele lumii reale la nivel de analiză şi la nivel de proiectare. Ele descriu atributele şi operaţiile obiectelor. Modelele statice nu iau în considerare evenimentele şi secvenţierea instrucţiunilor în timpul execuţiei modelului implementat. Reprezentarea grafică conţine pictograme ale obiectelor ce descriu relaţiile acestora cu alte obiecte.

Prezentarea tipurilor de modele statice se va face pe exemplul unui automat ATM (Automated Teller Machine) simplificat al cărui diagramă sistem este prezentată în figura următoare.

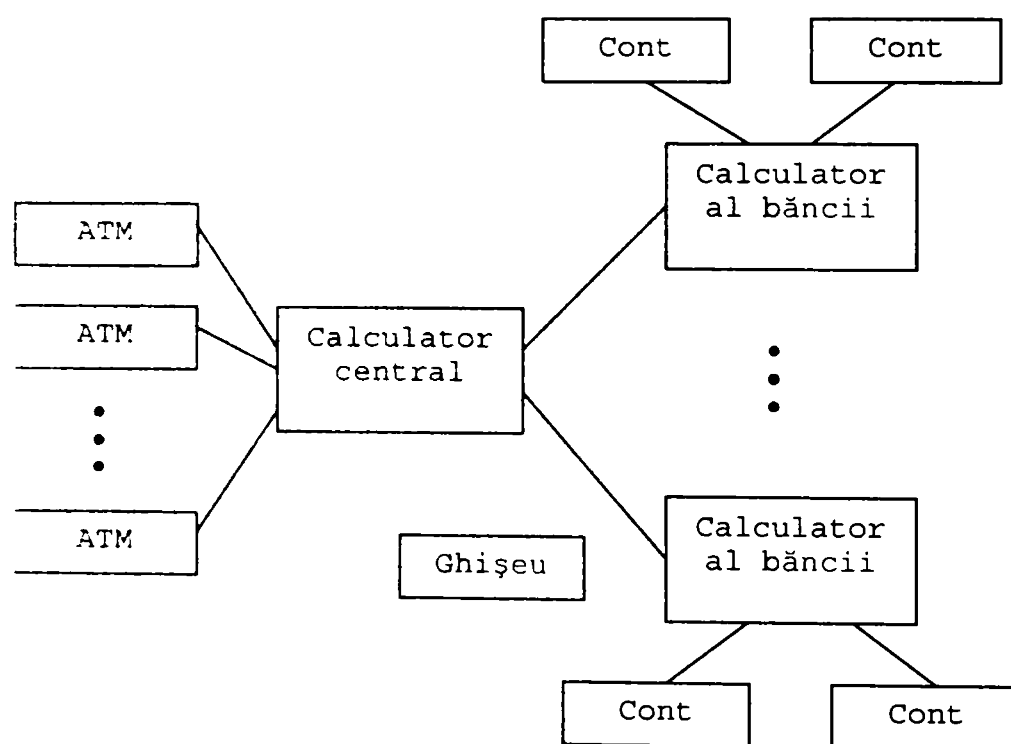


Figura 2.10 Diagrama sistem ATM

- *Modelul obiectelor* prezintă entităţile lumii reale sub forma unor dreptunghiuri ce conţin în partea superioară numele obiectului, atributele în partea centrală şi operaţiile în partea inferioară. Relaţiile între obiecte, denumite în cadrul unora dintre metode *asocieri*, sunt marcate ca şi conexiuni adnotate. Figura următoare prezintă modelul obiectelor pentru aplicaţia ATM.

BUPT

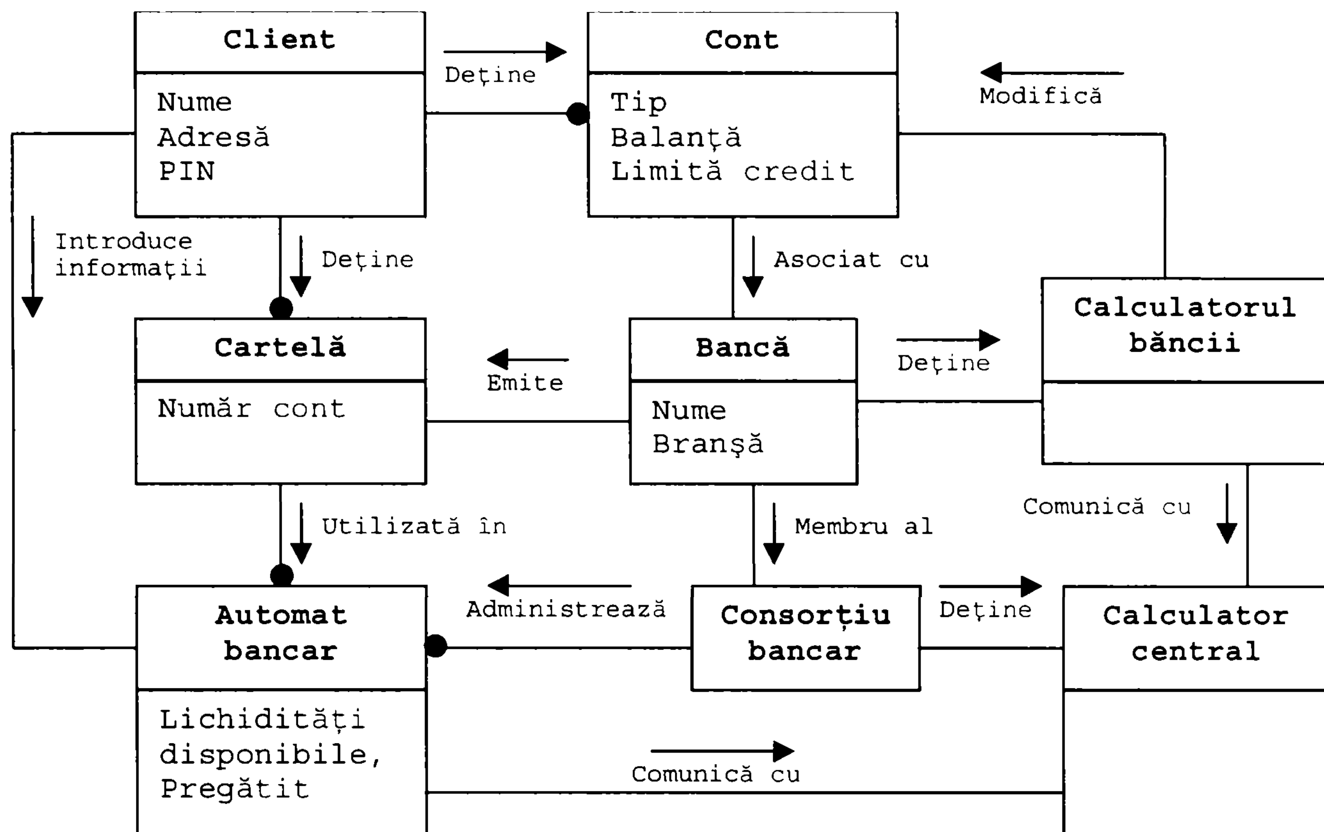


Figura 2.11 Modelul obiectelor pentru aplicaţia ATM

- *Modelul de moştenire şi agregare* prezintă descompunerea ierarhică a obiectelor ce compun sistemul implementat. Figura următoare exemplifică acest model pentru aplicaţia ATM. Agregarea este descrisă ca relaţie "parte a" între automatul bancar şi ecranul utilizator, tranzacţie şi imprimantă. Moştenirea este prezentată ca relaţie "este" între obiectul `Tranzacţie` şi obiectele `RetragereLichidităţi` şi `VerificareCont`.

BUPT

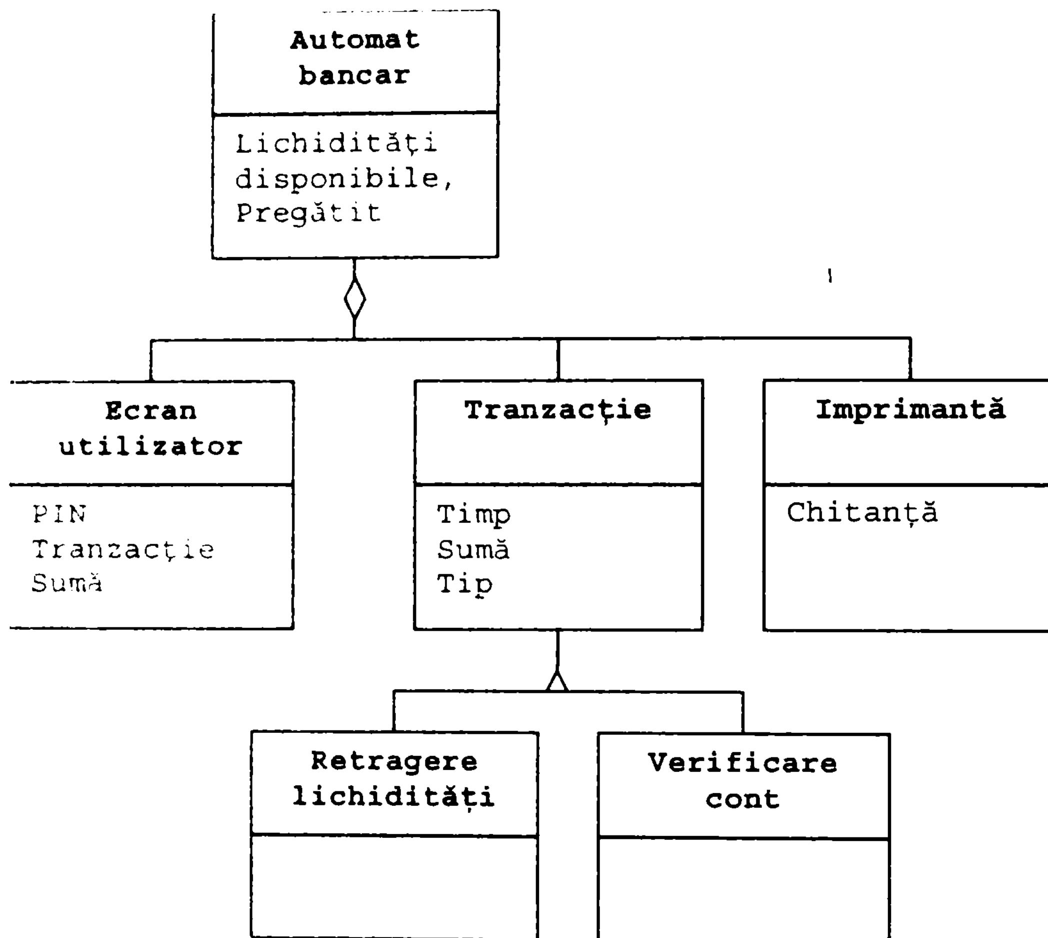


Figura 2.12 Model de moştenire şi agregare

- *Diagrama de arhitectură a modulelor* este utilizată în faza de proiectare şi prezintă entităţile arhitecturale statice şi interfeţele acestora. Fiecare simbol descrie interfaţa utilizată pentru transmiterea şi recepţionarea de mesaje de către obiectul încapsulat în interiorul frontierelor simbolului. Fluxul mesajelor poate fi indicat prin sensul săgeţilor ce conectează obiectele. Figura următoare ilustrează o diagramă de arhitectură a modulelor pentru aplicaţia ATM.

BUPT

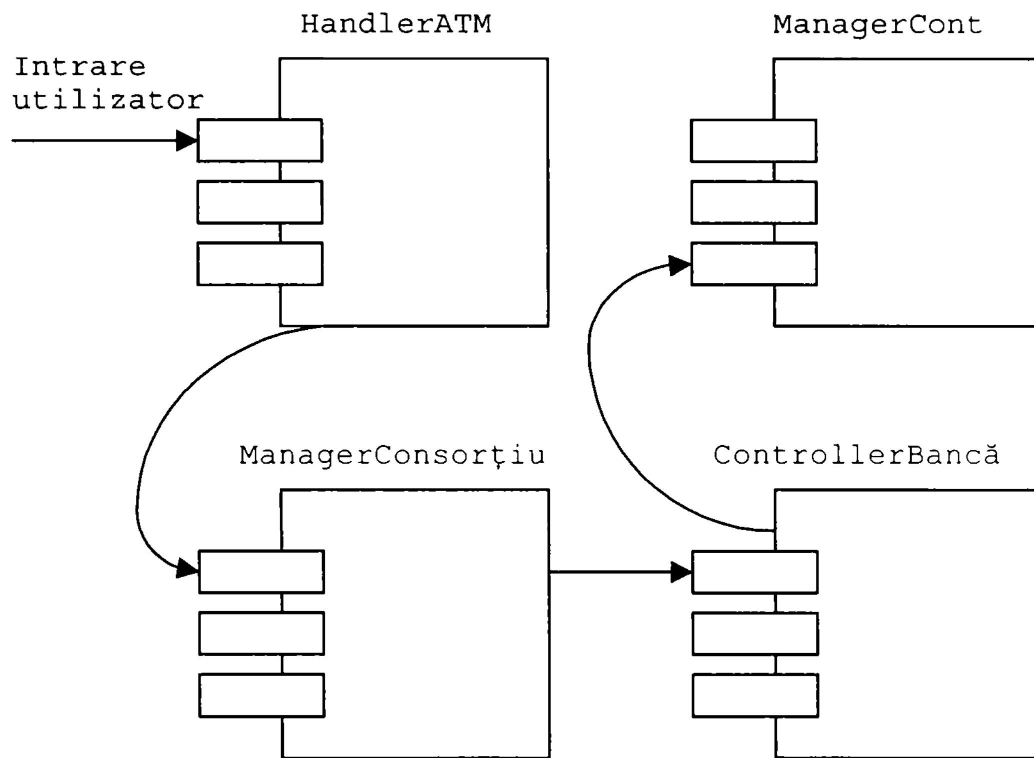


Figura 2.13 Diagrama de arhitectură a modulelor pentru aplicaţia ATM

2.5.3 Modelele dinamice

Modelele dinamice descriu obiectele şi relaţiile dintre acestea în contextul modificării în timp a sistemului. Ele reflectă schimbările de stare ale sistemului, secvenţierea evenimentelor, intrările şi ieşirile externe. Modelele dinamice prezentate vor fi exemplificate utilizând aplicaţia ATM.

- *Modelul de flux al obiectelor* descrie mesajele care circulă între obiecte. Unele metode orientate spre obiecte consideră obiectele ca fiind entităţi concurente de nivel înalt şi utilizează acest model pentru determinarea elementelor din cadrul subsistemelor.

BUPT

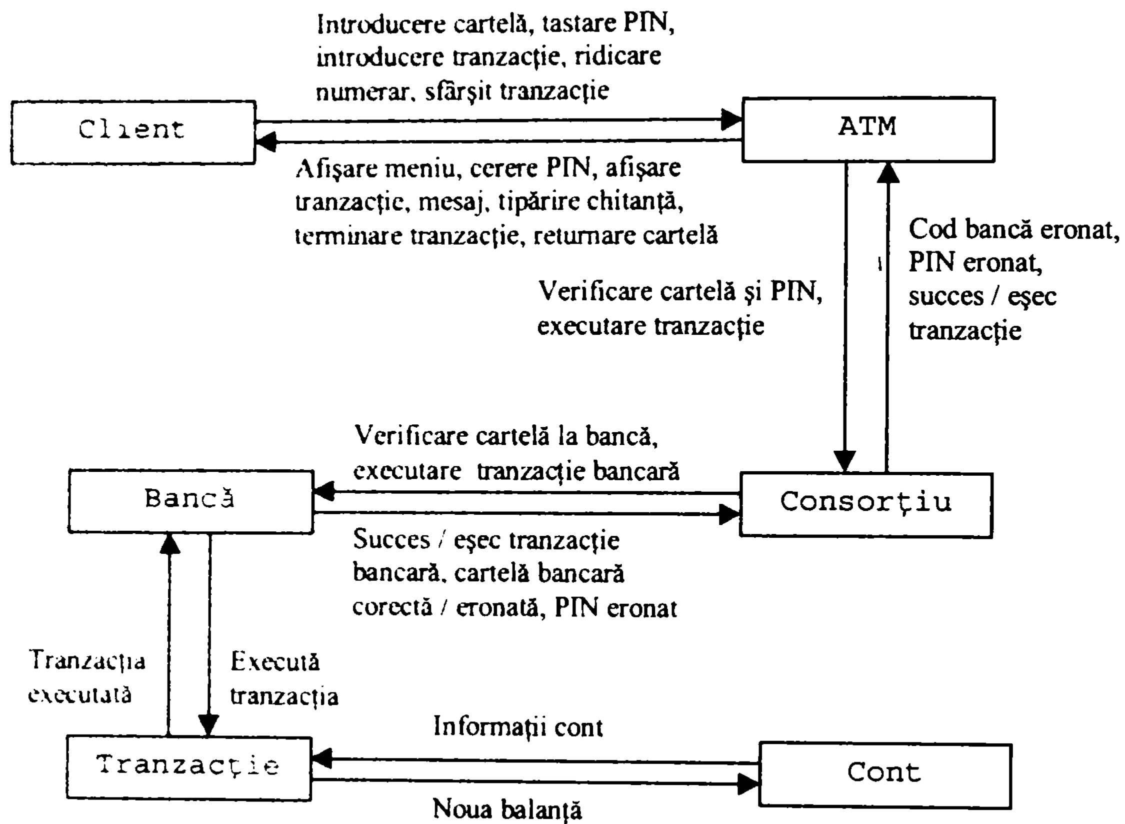


Figura 2.14 Modelul de flux al obiectelor pentru aplicaţia ATM

- *Diagrama de flux a evenimentelor* ilustrează evenimentele în ordinea lor cronologică la care participă obiectele lumii reale descrise în modelul obiectelor. Liniile verticale reprezintă obiectele iar săgeţile orizontale reprezintă evenimente la care participă acestea. Timpul este măsurat de sus în jos, reprezentarea ne fiind însă proporţională.

BUPT

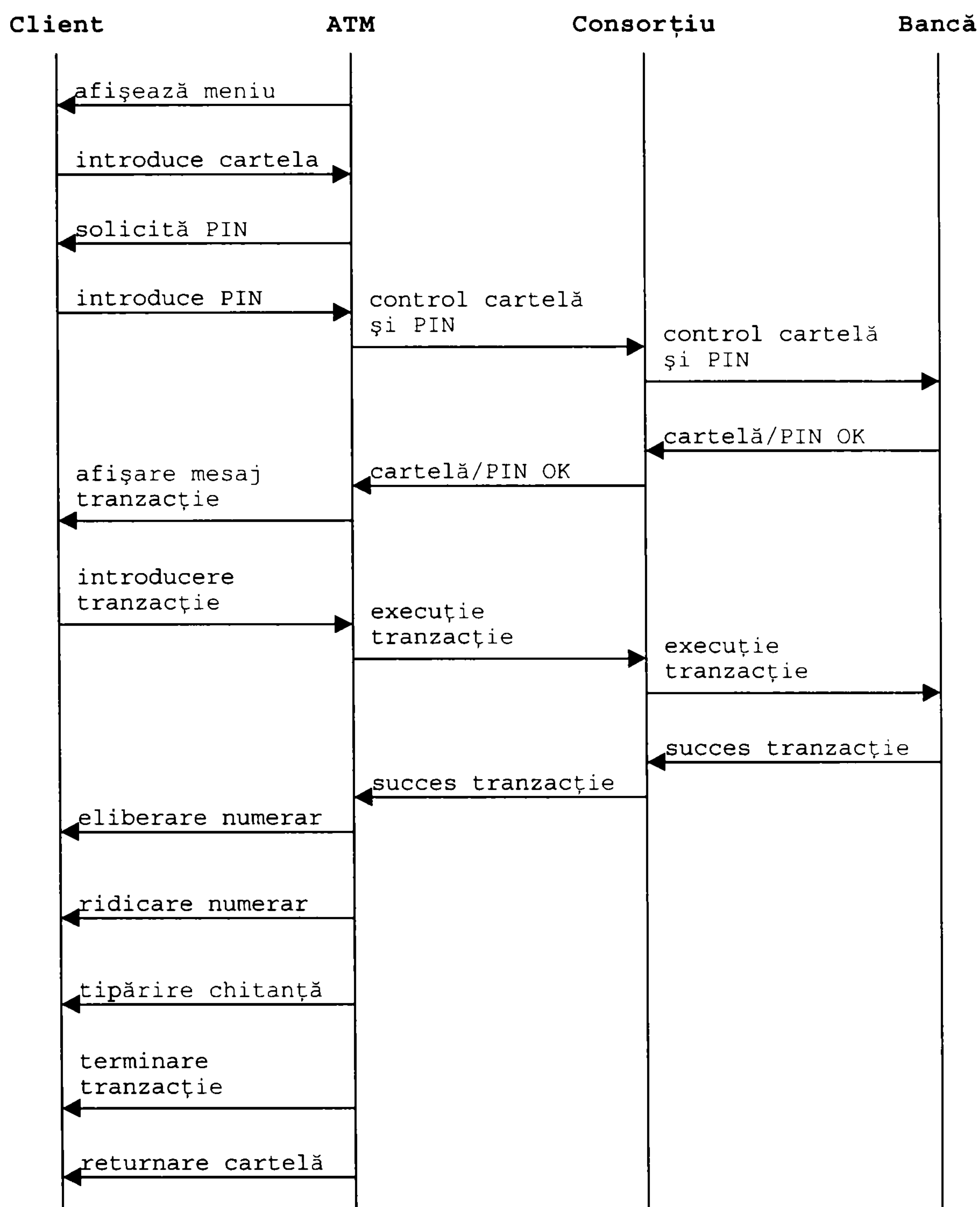


Figura 2.15 Diagrama de flux a evenimentelor pentru aplicaţia ATM

- *Diagrama de tranziţie a stărilor* reprezintă o altă formă de reprezentare a modelului dinamic. Fiecare dreptunghi reprezintă o stare dată a sistemului iar conexiunile etichetate reprezintă evenimentele care declanşează acţiunile care au ca efect tranziţia de la o stare la alta. Datorită numărului mare de stări ce caracterizează sistemele mari, în cazul acestora vor fi utilizate mai degrabă diagrame de tranziţie a stărilor la nivel de obiect şi nu la nivel de sistem.

BUPT

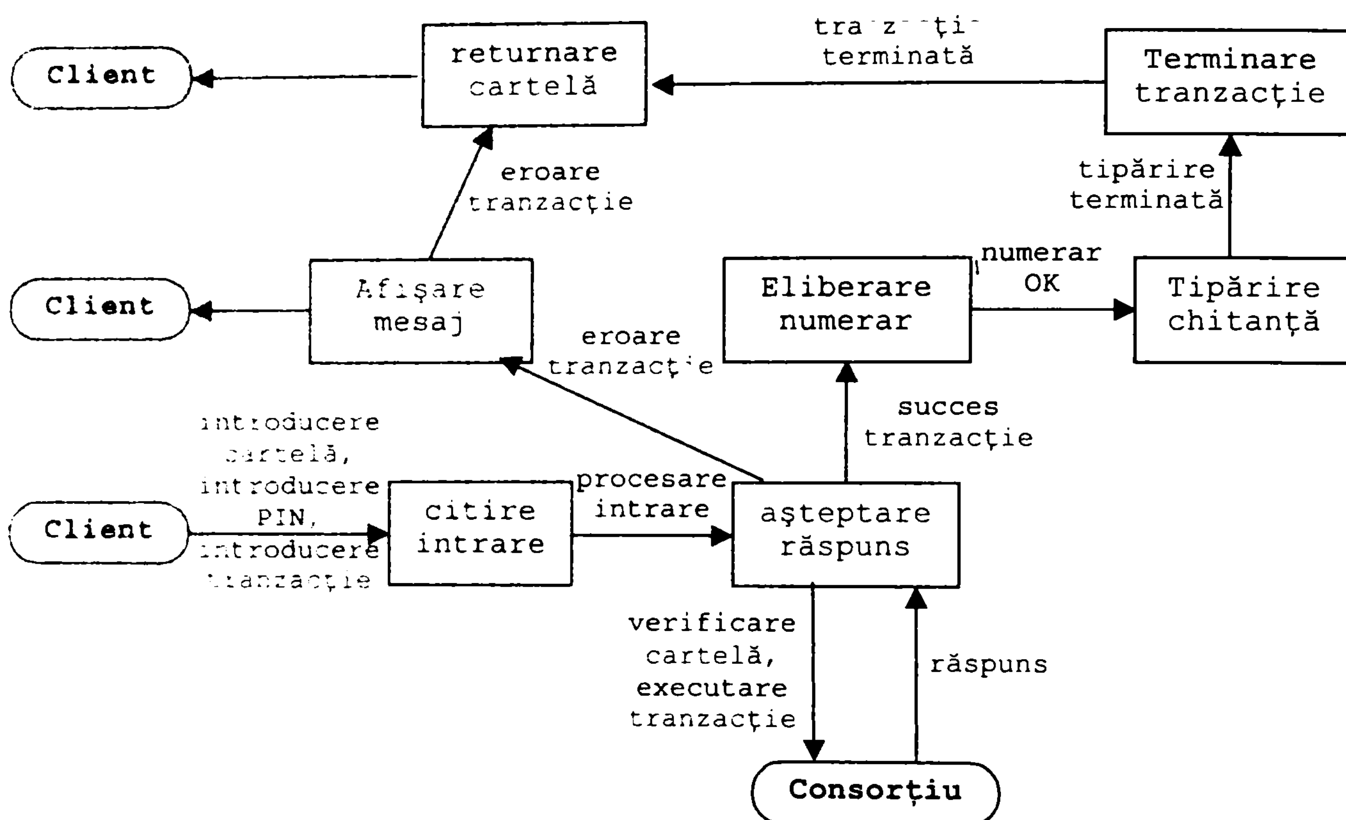


Figura 2.16 Diagramă de tranziţie a stărilor pentru aplicaţia ATM

2.5.4 Modelele de structură a sistemului

Modelele de structură a sistemului ilustrează arhitectura elementelor hardware şi interfeţele lor într-un sistem distribuit. Figura următoare prezintă un exemplu de model de structură a sistemului pentru aplicaţia ATM. Subsistemele reprezentând nodurile hardware în arhitectura distribuită sunt ilustrate sub forma unor dreptunghiuri cu colţuri rotunjite. Actualul model nu prezintă alocarea cerinţelor sistem componentelor hardware şi software, pentru acest aspect pot fi utilizate diagrame de flux a datelor la nivel de sistem.

BUPT

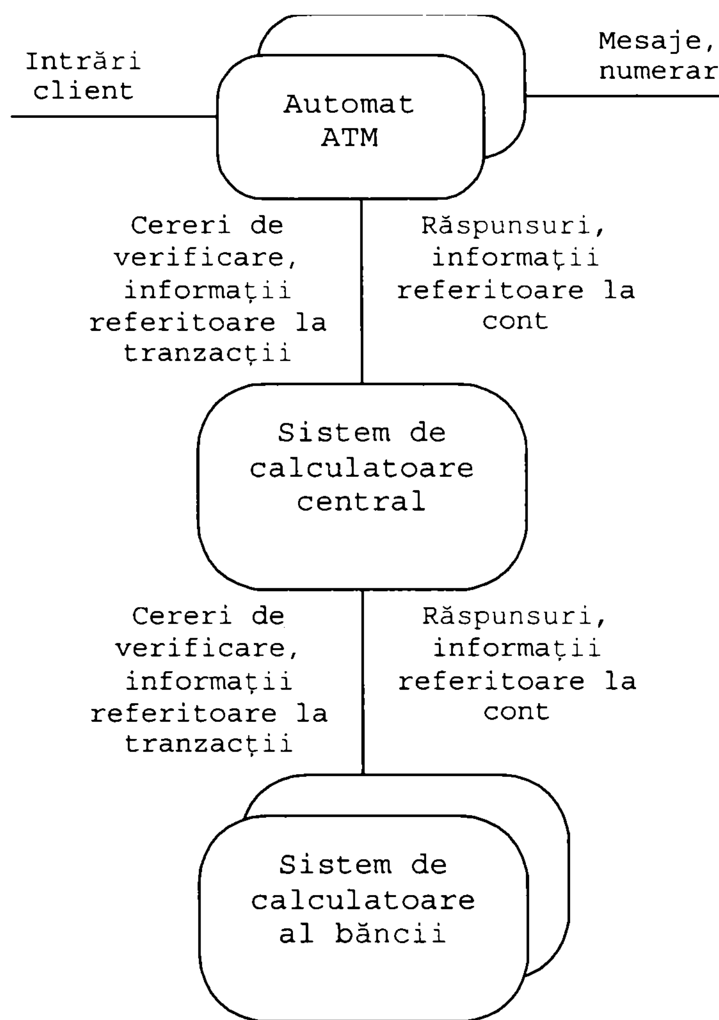


Figura 2. 17 Diagrama de structură a sistemului pentru aplicația ATM

2.5.5 Modelele de concurență

Unul dintre neajunsurile majorității metodelor de analiză şi proiectare orientate spre obiecte constă din lipsa abordării aspectelor legate de concurență ce apar în cadrul sistemelor de timp real. Una dintre paradigmele împrumutate din proiectarea structurată a sistemelor de timp real este abstractizarea proceselor [GOM84, NIE92]. Diagrama de structură a proceselor este utilizată pentru a descrie procesele ca entități planificabile independente aflate în competiție pentru aceeaşi unitate centrală de prelucrare.

Modul de implementare a proceselor nu rezultă din diagrama de structură a proceselor şi depinde de limbajul de programare utilizat şi de bibliotecile disponibile. De remarcat diferența dintre diagrama de structură a proceselor care se referă în mod explicit la comunicația şi sincronizarea proceselor şi diagrama de flux a obiectelor care consideră doar fluxul de mesaje

BUPT

care are loc intre obiectele concurente, obiecte ce pot ajunge în final mai degrabă subsisteme complete decât procese individuale.

2.5.6 *Modelele funcționale*

Modelele funcționale intâlnite în cadrul metodelor structurate pot fi utilizate pentru completarea modelelor specifice tehnologiei orientate spre obiecte. Astfel, *diagrama de tranziție a stărilor* poate fi utilizată la nivel de obiect pentru a descrie modificări dinamice de stare la acest nivel. *Diagrama de trasare a evenimentelor* conferă o perspectivă asupra sincronizării evenimentelor intre cele mai importante obiecte.

Diagramele de flux a datelor pot fi utilizate pentru a determina alocarea cerințelor sistem spre componente hardware şi software în cazuri în care un obiect este partajat între unul sau mai multe subsisteme. Diagrama sistem din figura 2.10 poate fi considerată o diagramă de flux a datelor la nivelul cel mai înalt.

Diagrama de structură a proceselor poate fi utilizată pentru a analiza transmiterea mesajelor intre task-uri şi pentru a determina controlul şi sincronizarea corespunzătoare a acestora.

2.6 Metode de analiză şi proiectare orientate spre obiecte

Mecanismele de bază ale tuturor metodelor de analiză şi proiectare orientate spre obiecte utilizează modelele OOA şi OOD descrise anterior. În cele ce urmează vor fi prezentate câteva dintre cele mai cunoscute metode de analiză şi proiectare orientate spre obiecte. Discuția se va axa pe modelele şi notațiile utilizate, pe elementele de euristică precum şi liniile directoare prezentate în cadrul metodelor. Unele dintre elementele prezente în cadrul acestor metode vor fi regăsite şi în cadrul procesului de dezvoltare software propus în cadrul capitolului următor.

2.6.1 *Tehnica modelării obiectelor (OMT)*

Tehnica modelării obiectelor (OMT) reprezintă o metodă aplicabilă atât în etapa OOA cât şi în etapa OOD [RUM91]. Notațiile utilizate sunt detaliate şi suportă toate reprezentările de modelare prezentate anterior precum şi relațiile clase-obiecte. Metoda OMT se bazează pe următoarele trei modele:

BUPT

- *modelul obiectelor*, reprezentând o descriere a claselor, a obiectelor şi a relaţiilor statice între acestea. Reprezentările suportate sunt cele de generalizare-specializare, asociere şi agregare. Diagrama claselor şi a obiectelor conţine cardinalitate, legături, atribute şi operaţii. Rezultatul activităţii de modelare a obiectelor reprezintă o perspectivă a abstractizărilor esenţiale ale lumii reale incluzând relaţiile statice dintre acestea.
- *Modelul dinamic* utilizează diagrama de tranziţie a stărilor la nivel de obiect pentru a descrie schimbările de stare şi diagrame de trasare a evenimentelor pentru a descrie cronologia evenimentelor. Diagramele de flux a evenimentelor sunt utilizate pentru a descrie fluxul evenimentelor între obiecte. Aceste diagrame reprezintă echivalentul dinamic al diagramelor statice de obiecte echivalente. Rezultatul activităţii de modelare dinamică constituie o reprezentare a cronologiei evenimentelor, fluxul evenimentelor între abstractizările cheie şi schimbările de stare care pot avea loc în cadrul fiecărui obiect.
- *Modelul funcţional* descrie descompunerea funcţională şi relaţiile între entităţile funcţionale. Notaţia utilizată în cadrul acestui model reprezintă o ierarhie de diagrame de flux de date. Fluxurile de date ilustrate în cadrul acestor diagrame corespund obiectelor şi atributelor în diagrama obiectelor. Deşi acest model este descris ca parte integrantă a metodei OMT, utilizarea sa este descurajată în favoarea modelelor obiectelor şi a celui dinamic [ACC93a].

Notaţiile legate da cardinalitatea unei asocieri în cadrul metodei OMT sunt ilustrate în figura următoare.

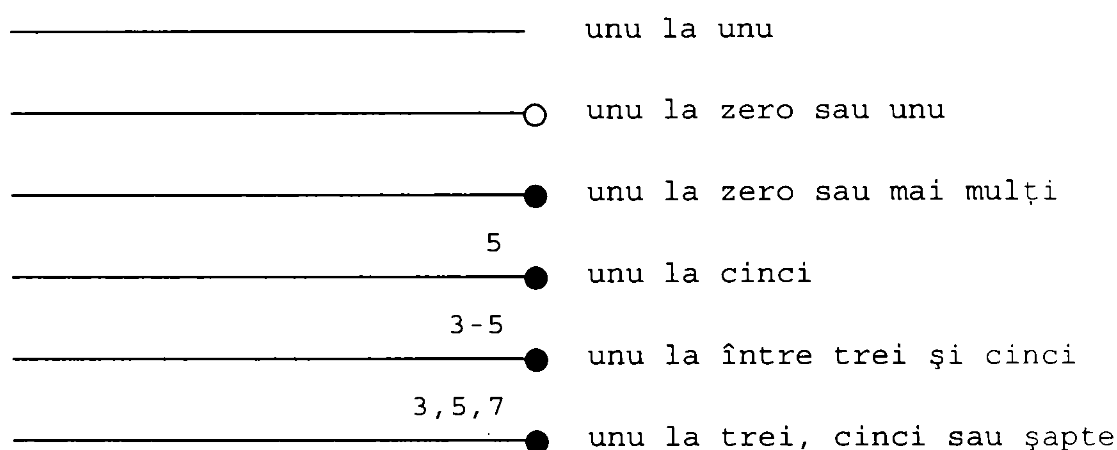


Figura 2.18 Reprezentarea cardinalităţii în cadrul metodei OMT

BUPT

2.6.1.1 Etapa OOA

Utilizarea metodei OMT constă din următorii trei paşi în etapa de analiză orientată spre obiecte:

1. Crearea modelului obiectelor
   - determinarea unui set de obiecte şi clase pornind de la specificarea cerinţelor sistemului
   - eliminarea claselor şi a obiectelor redundante, triviale şi irelevante
   - pregătirea unui dicţionar de date
   - determinarea asocierilor între clase şi obiecte
   - eliminarea asocierilor nedorite
   - identificarea atributelor şi operaţiilor
   - determinarea ierarhiilor de moştenire utilizând criterii de generalizare-specializare
   - gruparea claselor în module

2. Crearea modelului dinamic
   - pregătirea scenariilor
   - identificarea evenimentelor asociate unui anumit scenariu
   - pregătirea diagramelor de trasare a evenimentelor pentru fiecare scenariu
   - crearea de diagrame de tranziţie a stărilor pentru fiecare obiect

3. Crearea modelului funcţional
   - identificarea valorilor de intrare şi de ieşire şi pregătirea diagramei sistem
   - pregătirea diagramelor de flux a datelor indicând dependenţele funcţionale
   - descrierea entităţilor funcţionale
   - identificarea constrângerilor între obiecte
   - specificarea de criterii de optimizare în termenii parametrilor de spaţiu şi de timp

2.6.1.2 Proiectarea sistemului

Se recomandă ca faza de analiză să fie urmată de faza de proiectare a sistemului cu alocarea cerinţelor sistemului spre componente hardware şi software. Structura rezultantă reprezintă arhitectura sistemului. Această activitate trebuie finalizată înainte de începutul etapei OOD. Faza de proiectare a sistemului include următorii paşi:

- organizarea sistemului în subsisteme
- identificarea cazurilor inerente de concurenţă

BUPT

- alocarea subsistemelor la procesoare şi task-uri
- stabilirea unei abordări pentru stocarea datelor
- prescrierea mecanismelor de acces la resursele globale
- stabilirea unui mecanism pentru control software al evenimentelor interne şi externe
- stabilirea unui mecanism pentru inițializarea şi părăsirea sistemului şi pentru tratarea excepțiilor
- stabilirea de priorități de dezvoltare

O descriere detaliată a acestor paşi este prezentată în [RUM91]. Elemente suplimentare de euristică sunt prezentate în [ACC93b]. Subsistemele sunt create prin gruparea claselor şi a obiectelor înrudite care vor forma împreună o interfață client-server bine definită cu lumea exterioară. Nivelul inferior al unui subsistem este reprezentat de modul, constituit dintr-o grupare logică de clase, asocieri şi generalizări. Frontiera unui subsistem coincide de obicei cu frontiera unui modul. Se recomandă păstrarea unui număr cât mai mic de subsisteme, dacă este posibil acesta să fie mai mic de 20.

Tratarea aspectelor referitoare la concurență este legată de determinarea posibilității ca obiectele să poată fi active simultan, în cadrul unui task intrând doar obiecte mutual exclusive. Determinarea faptului dacă obiectele pot fi combinate se efectuează pe baza diagramelor de tranziție a stărilor, notația utilizată fiind foarte complexă (notația Harel).

Fiecare subsistem este considerat a fi o entitate concurentă care trebuie alocată unei componente hardware. Următorii paşi sunt incluşi în această activitate:

- estimarea cerințelor legate de resursele hardware
- trasarea unei diagrame de arhitectură hardware
- alocarea de cerințe hardware şi software fiecărui subsistem
- asignarea cerințelor software procesoarelor
- determinarea conectivității fizice a subsistemelor

După terminarea fazei de proiectare a sistemului toate cerințele sistemului au fost alocate subsistemelor, în cadrul fiecărui subsistem fiind alocate atât cerințele hardware cât şi cele software. Următoarele considerații se referă la alocarea subsistemelor concurente entităților hardware [RUM91, p.203]:

- se estimează cerințele de performanță şi resursele necesare pentru a le satisface
- se determină dacă un subsistem va fi implementat software sau hardware

BUPT

- se alocă subsistemele software procesoarelor care satisfac cerinţele de performanţă şi se minimizează comunicarea interprocesor
- se determină conectivitatea unităţilor fizice care formează subsistemele

2.6.1.3 Etapa OOD

Tranziţia de la etapa OOA la etapa OOD în cadrul metodei OMT este vag delimitată [RUM91]. Etapa OOD are rolul de a furniza o bază de implementare a modelului de analiză. Paşii specifici recomandaţi pentru etapa OOD sunt:

- combinarea modelelor obiect, dinamic şi funcţional pentru a obţine operaţii asupra claselor
- proiectarea algoritmilor pentru implementarea operaţiilor
- optimizarea căilor de acces la date
- implementarea controlului pentru interacţiuni externe
- ajustarea structurii de clase pentru utilizarea moştenirii
- proiectarea asocierilor
- determinarea reprezentării obiectelor
- gruparea claselor şi a asocierilor în module

2.6.1.4 Aspecte legate de operarea în timp real şi de concurenţă

Metoda OMT conţine puţine specificaţii legate de modalităţile de creare de elemente concurente pornind de la obiecte sau module. Unele sugestii menţionează examinarea diagramelor de tranziţie a stărilor şi gruparea obiectelor într-un task, fără însă a specifica criterii de tranziţie de la diagrama de tranziţie a stărilor la task. Se sugerează ca evenimentele să fie implementate ca apeluri inter-task, utilizând facilităţile puse la dispoziţie de limbajul de programare şi de sistemul de operare utilizat.

2.6.1.5 Instrumente CASE

Instrumente CASE care implementează metoda OMT şi care rulează pe calculatoare compatibile IBM PC sunt OMTool, C++ Designer, Rational Rose, etc. Suport pentru metoda OMT în cadrul platformelor UNIX se regăseşte în cadrul produselor Software through Pictures (StP) produs de IDE şi Cadre's Team Work.

BUPT

2.6.2 Metoda Object-Oriented Software Engineering (OOSE)

Metoda OOSE reprezintă mai mult decât o metodă orientată spre obiecte singulară, suportând abordarea completă a întregului proces de dezvoltare a unui produs software [JAC92]. Unul dintre aspectele cele mai importante ale metodei OOSE constă în „abordarea pe baza cazurilor de uz" (*use cases*). Cazurile de uz, compuse din scenarii, descriu interacţiunea utilizatorului cu sistemul. Se vor determina scenarii pentru toate interacţiunile utilizatorului cu sistemul, analiza, proiectarea şi implementarea concentrându-se în jurul fiecărui scenariu. Cerinţele sistemului pot fi astfel urmărite de la definiţia interfeţelor utilizator până la implementare şi testare.

Un alt avantaj direct al metodei OOSE constă în faptul că suportă o strategie de dezvoltare incrementală. Reprezentările de modelare a claselor şi obiectelor în cadrul metodei OMT sunt la nivel de sistem, pe când metoda OOSE prezintă reprezentări corespunzătoare la nivel de scenariu. Metoda OMT utilizează de asemenea scenarii, dar ele sunt stabilite doar după definirea modelului obiectelor şi ele sunt utilizate în principal pentru determinarea modelului dinamic. În cadrul metodei OOSE scenariile sunt definite pe baza cerinţelor sistemului, setul de scenarii fiind denumit model al cerinţelor. Indicaţii pentru crearea scenariilor se regăsesc în [JAC92].

Modelele OOA şi OOD utilizate în cadrul metodei OOSE sunt:

- *modelul de scenarii*, modelează rolul utilizatorilor şi al sistemelor externe ce sunt interfaţate cu sistemul ce se dezvoltă
- *modelul domeniului obiectelor*, formează baza pentru înţelegerea conceptelor legate de domeniul problemei
- *modelul de analiză*, identifică clase, obiecte, atribute, operaţii şi asocierile dintre ele; clasifică obiectele în obiecte de interfaţă, obiecte entitate şi obiecte de control
- *modelul de subsisteme*, grupează obiectele şi alte subsisteme în unităţi de gestiune
- *modelul de proiectare*, realizează tranziţia de la obiectele de analiză la implementări sub formă de "blocuri". Modulele obiect sunt dezvoltate ca şi clase.

2.6.2.1 Etapa OOA

Etapa de analiză descrisă în cadrul metodei OMT este precedată în cadrul metodei OOSE de o fază de modelare a cerinţelor pentru crearea de scenarii şi de o analiză a domeniului pentru înţelegerea conceptelor legate de domeniul problemei.

BUPT

Metoda OOSE utilizează toate modelele statice şi dinamice descrise anterior. Un aspect unic al metodei OOSE constă în modul de încadrare a obiectelor în cele trei dimensiuni ale spaţiului de prezentare. comportament, prezentare şi informaţie, după cum rezultă din figura următoare.

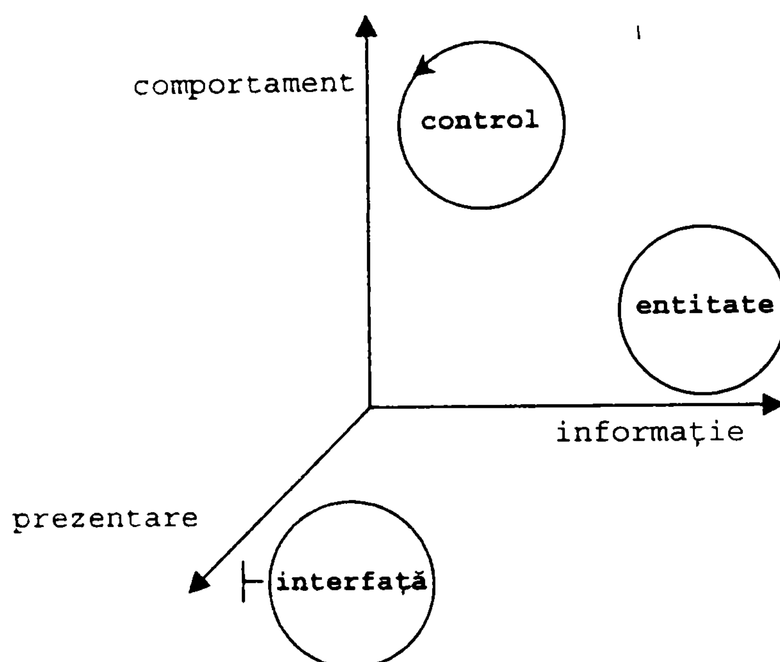


Figura 2.19 Obiectele încadrate în spaţiul informaţional

- *Obiectele entitate* reprezintă cerinţele abstractizărilor esenţiale exprimate în obiecte ale lumii reale şi sunt asociate axei informaţie.
- *Obiectele interfaţă* modelează informaţia legată de interfaţa cu sistemele externe şi sunt aliniate cu axa de prezentare.
- *Obiectele de control* sunt utilizate pentru manipularea schimbărilor de stare pe termen scurt ale sistemului şi sunt asociate axei comportamentului.

În cadrul metodei OOSE sunt utilizate modelele de clase şi obiecte descrise anterior pentru a genera reprezentările pentru fiecare scenariu. Modelele dinamice utilizate în cadrul metodei OOSE sunt diagramele de tranziţie a stărilor (notaţie Mealy [HAT87]), diagramele de interacţiune (diagrame de trasare a evenimentelor) şi diagrame de flux a evenimentelor.

2.6.2.2 Proiectarea sistemului

În faza de proiectare a sistemului, obiectele create în etapa de analiză sunt grupate în subsisteme pentru a reduce complexitatea sistemului şi pentru a genera interfeţe adecvate pentru

BUPT

integrarea modulelor. Euristica utilizată la gruparea obiectelor în subsisteme se bazează pe cuplarea redusă între subsisteme şi coeziune puternică între obiectele din cadrul unui subsistem. Determinarea gradului de coeziune funcţională între două obiecte se bazează pe răspunsul la următoarele probleme legate de proiectare [JAC92, p.191]:

- Modificările la nivelul unui obiect vor genera modificări la nivelul celuilalt obiect?
- Cele două obiecte comunică cu acelaşi actor?
- Sunt cele două obiecte dependente de un al treilea obiect cum ar fi un obiect de interfaţă sau entitate?
- Efectuează unul din obiecte operaţii asupra celuilalt?

Metoda OOSE nu specifică linii directoare explicite pentru alocarea cerinţelor hardware şi software la subsisteme.

2.6.2.3 Etapa OOD

În cadrul metodei OOSE se pune un accent deosebit pe separarea între etapa OOA şi etapa OOD. Pictogramele utilizate în etapa OOD sunt diferite faţă de cele utilizate în etapa OOA iar obiectele OOD poartă denumirea de "blocuri".

Faza de tranziţie de la etapa OOA începe cu transformarea obiectelor de analiză în obiecte de proiectare (blocuri). Obiectele de analiză evidenţiază cerinţele de modelare pe când în cazul blocurilor se pune accentul pe semantica mediului de implementare. În faza de rafinare a blocurilor iniţiale de obicei apar obiecte de proiectare noi care vor fi adăugate pentru a asigura o arhitectură implementabilă. Alte activităţi specifice etapei OOD includ stabilirea de consideraţii legate de performanţa sistemului, aspecte legate de operarea în timp real şi toleranţa la defecţiuni.

Unul din avantajele importante prezentate de metoda OOSE prin delimitarea etapei de proiectare de cea de analiză constă în faptul că cerinţele sistem pot fi urmărite din faza de analiză până în cadrul blocurilor specifice fazei de proiectare. Alte metode orientate spre obiecte în cadrul cărora această delimitare nu este atât de clar definită nu permit această urmărire, produsele fazei de analiză tinzând să se piardă, în cadrul dezvoltării predominând structura de proiectare. Se pierde astfel legătura între analiză, proiectare şi implementare în ambele direcţii.

Pentru fiecare scenariu se utilizează diagrame de interacţiune (diagrame de trasare a evenimentelor) pentru a vizualiza modul de interacţiune a obiectelor. În cadrul acestor diagrame se face distincţie între mesajele ce vor fi implementate ca apeluri C++ şi semnalele ce vor fi

BUPT

implementate ca mecanisme sincrone sau asincrone de tratare a întreruperilor şi a activităţilor concurente. Diagramele de tranziţie a stărilor (notaţia Mealy) sunt utilizate pentru a elucida comportamentul dinamic al obiectelor individuale.

Activitatea finală a etapei OOD constă în pregătirea interfeţelor blocurilor pentru implementare sub formă de module într-un limbaj de programare orientat spre obiecte. Această activitate include stabilirea nivelului de ascundere al informaţiei în ceea ce priveşte atributele şi operaţiile.

2.6.2.4 Aspecte legate de operarea în timp real şi de concurenţă

Metoda OOSE conţine şi consideraţii legate de sistemele ce operează în timp real şi de crearea de procese concurente. Structura de procese este stabilită în faza de analiză. Una dintre sugestiile metodei este legată de identificarea elementelor concurente între obiectele interfaţă. Semantica comunicării şi a sincronizării nu face parte din metodă şi va trebui implementată utilizând serviciile puse la dispoziţie de sistemul de operare sau de mediul de programare. Cerinţele legate de operarea în timp real sunt determinate prin asocierea de atribute de timp secvenţei operaţionale din cadrul scenariilor.

Concurenţa este asociată scenariilor în două moduri:

- activităţi simultane pot să aibă loc în cadrul unui scenariu
- scenariile pot să se desfăşoare în paralel.

2.6.2.5 Instrumente CASE

Din categoria instrumentelor CASE care implementează metoda OOSE se pot menţiona produsele Objectory (OBJectfactORY) al firmei suedeze Objectory AB şi Rose al firmei Rational Software.

2.6.3 Metoda Booch'93

Metoda Booch iniţială, prezentată în [BOO91] se concentra asupra modelării obiectelor în termeni de comportament, responsabilităţi şi colaborări şi era dedicată în principal proiectării orientate spre obiecte. Metoda a fost ulterior dezvoltată [BOO93] cu notaţii referitoare la atribute şi asocieri şi este aplicabilă acum atât fazei OOA cât şi fazei OOD.

BUPT

2.6.3.1 Etapa OOA/OOD

Metoda utilizează toate modelele statice şi dinamice prezentate anterior şi cuprinde următorii paşi:

- identificarea claselor şi obiectelor la nivelul adecvat de abstractizare
- determinarea semanticii claselor şi obiectelor
- determinarea relaţiilor între clase şi obiecte
- specificarea interfeţelor claselor şi obiectelor
- implementarea claselor şi obiectelor în limbajul de programare ales

Notaţiile utilizate diferă destul de mult de alte metode orientate spre obiecte iar crearea şi întreţinerea reprezentărilor de analiză şi proiectare necesită utilizarea unui instrument CASE. Una dintre diferenţele importante faţă de metoda OMT constă în reprezentarea cardinalităţii şi a diverselor relaţii între clase şi obiecte. O altă diferenţă este legată de forma pictogramelor ce reprezintă clasele şi obiectele. Raţiunea pentru care clasele sunt reprezentate sub forma unor nori ai căror contur este trasat cu linie întreruptă iar obiectele au aceeaşi reprezentare dar conturul este trasat cu linie continuă [BOO93, p. 177] este legată de faptul că acestea sugerează abstractizări care nu au frontiere clar definite iar dreptunghiurile utilizate în cadrul metodei OMT şi a altor metode sunt foarte frecvent supraîncărcate în cadrul diverselor modele cu diverse reprezentări.

Figura următoare prezintă un exemplu de diagramă de clase pentru aplicaţia ATM.

BUPT

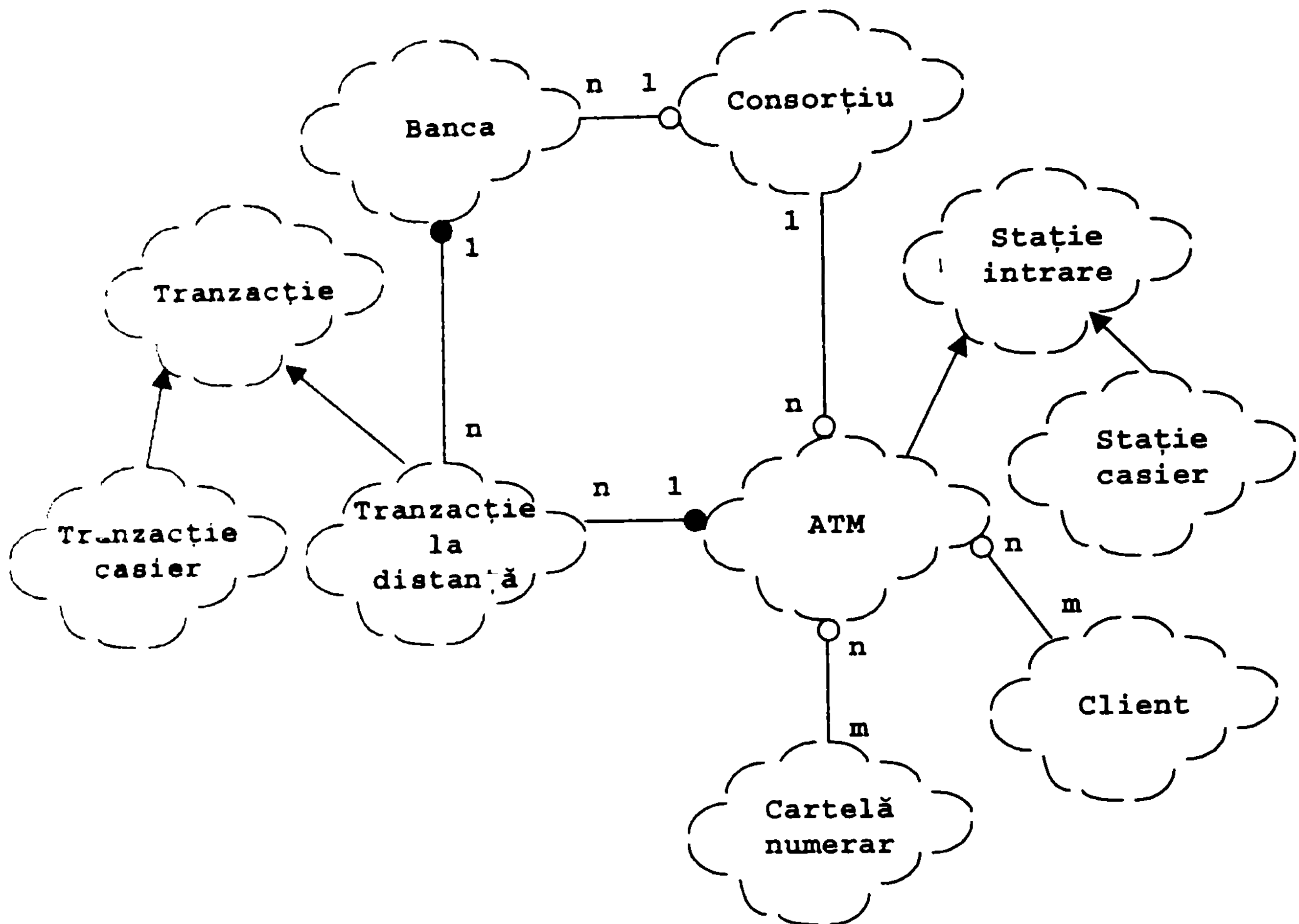


Figura 2.20 Diagrama de clase pentru aplicația ATM

Notația utilizată în cadrul metodei OMT pentru cardinalitate se folosește în cadrul metodei Booch'93 pentru relațiile de moștenire, agregare și utilizare. Cardinalitatea apare explicit în dreptul clasei sau obiectului corespondent. Diagrama de clase este utilizată în etapa OOA pentru documentarea claselor și a relațiilor dintre ele. În cadrul etapei OOD ea este utilizată pentru a determina structura de clase care va forma arhitectura sistemului.

Figura următoare prezintă o diagramă parțială a obiectelor în cadrul aplicației ATM. Această diagramă este similară diagramei de flux a obiectelor din cadrul metodei OMT. Diagrama obiectelor este utilizată în etapa OOA pentru a reprezenta o perspectivă a structurii de obiecte a sistemului pentru un set de instanțe de clase. În etapa OOD această diagramă este utilizată pentru a ilustra comportamentul dinamic al obiectelor. În general, diagrama de clase este utilizată pentru a determina descompunerea și ierarhia de clase în cadrul sistemului iar diagrama de obiecte este utilizată pentru a identifica interacțiunile în cadrul unui set tipic de obiecte derivate dintr-o structură de clase.

BUPT

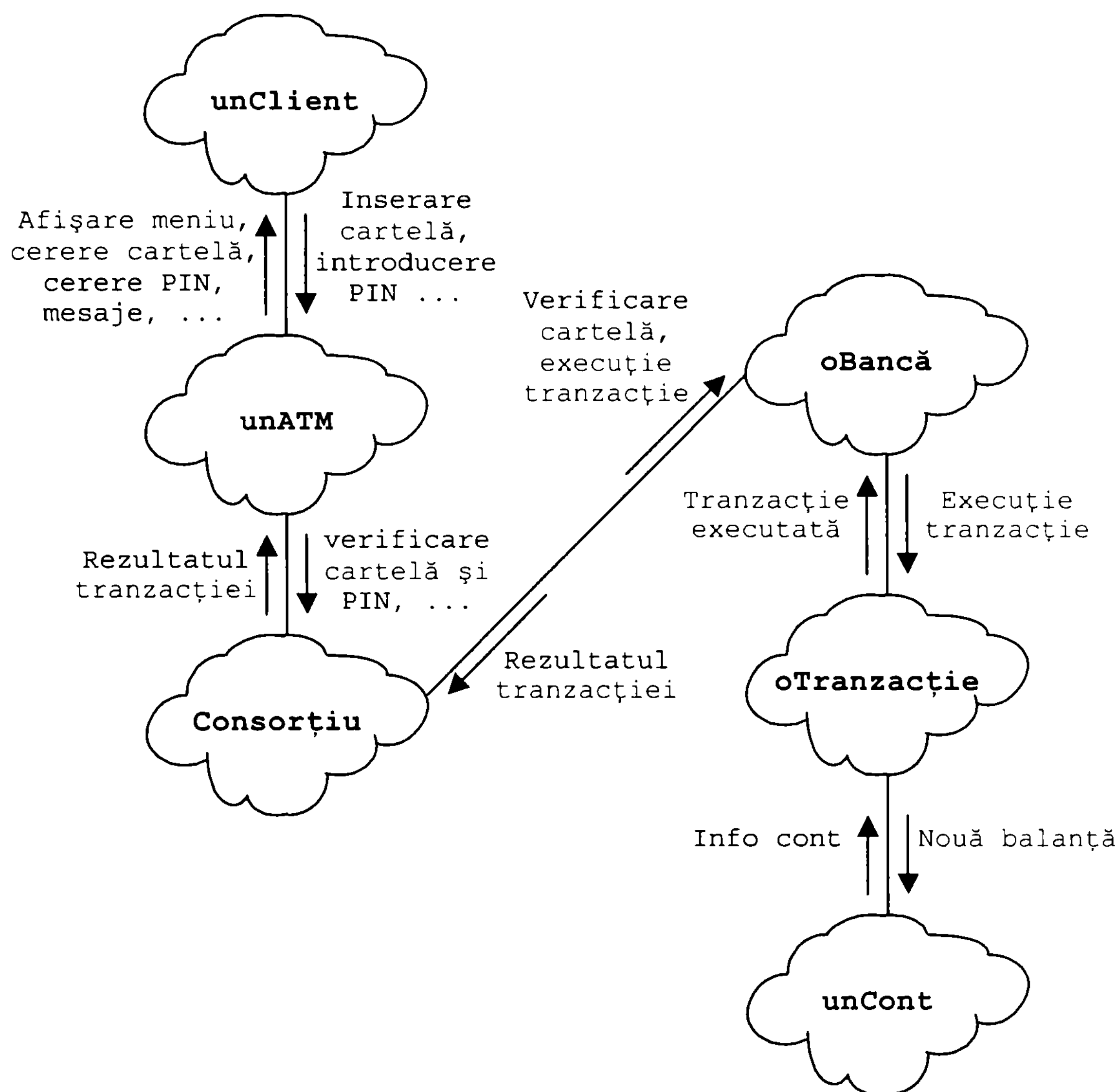


Figura 2.21 Diagrama obiectelor pentru aplicaţia ATM

Un alt model dinamic utilizat în cadrul metodei Booch'93 este diagrama de tranziţie a stărilor în cadrul căreia notaţia a fost adoptată de la Harel [HAR87] şi este relativ similară celei utilizate în cadrul metodei OMT. Metoda Booch'93 utilizează de asemenea diagrame de interacţiune a sistemului ce sunt aproape identice cu diagramele de trasare a evenimentelor utilizate în cadrul metodei OMT. Pentru descrierea scenariilor complexe se utilizează diagrame script similare diagramelor de interacţiune din cadrul metodei OOSE.

Metoda Booch'93 se aplică atât etapei de analiză cât şi celei de proiectare. Între cele două etape nu există o distincţie clară. Pentru identificarea claselor şi a obiectelor se sugerează un proces ce constă din trei paşi:

1. Identificarea claselor şi a obiectelor în primul rând pe baza proprietăţilor domeniului problemei, cum ar fi entităţi tangibile, roluri, evenimente, locuri, etc.
2. Dacă pasul anterior nu generează un set satisfăcător de abstractizări ale lumii reale, procesul de generare a claselor şi obiectelor va fi focalizat pe comportamentul dinamic

BUPT

al surselor primare pentru clase şi obiecte. Se încearcă formarea claselor din grupuri de obiecte care prezintă comportament similar.

3. Dacă ambii paşi anteriori eşuează se va încerca clasificarea pe bază de asociere. Grupurile de obiecte vor fi identificate pe baza asemănării cu un obiect prototip. Un exemplu pentru această categorie ar fi jocurile, pentru că nu există un grup comun de proprietăţi partajate de toate jocurile.

Utilizarea de cartele CRC este încurajată ca o metodă eficace de brain-storming în faza iniţială de identificare a claselor şi a obiectelor [NIE95].

2.6.3.2 Proiectarea sistemului

Faza de proiectare a sistemului este recomandată sistemelor ce utilizează în mod intensiv resurse hardware şi debutează prin identificarea hardware-ului ce va servi ca bază pentru implementarea software. Pentru descrierea structurii hardware se utilizează o diagramă de proces ce utilizează cuburi pentru a descrie entităţile hardware reprezentate de:

- *procesoare* (entităţi hardware capabile de a executa programe de aplicaţie)
- *dispozitive* (entităţi hardware ce nu sunt capabile să execute programe de aplicaţie ci doar programe firmware sau hardwired cum ar fi conversia analog-digital sau digital-analog)
- *conexiuni* (linii unidirecţionale de legătură între procesoare şi dispozitive).

Diagrama de proces reprezintă echivalentul unei diagrame de arhitectură hardware de nivel înalt. Diagrama de proces este utilizată şi pentru a ilustra alocarea proceselor la procesoare.

Metoda nu cuprinde sugestii pentru alocarea cerinţelor hardware şi software. Booch afirmă că [BOO93,p.295] "...modificarea frontierei hardware-software nu are semnificaţie materială pentru arhitectura orientată spre obiecte. Modificarea detaliilor legate de arhitectura hardware are impact doar asupra abstractizărilor de la nivelele inferioare ale sistemului."

2.6.3.3 Aspecte legate de operarea în timp real şi de concurenţă

Suportul metodei pentru sisteme de timp real include următoarele specificaţii legate de modul de planificare a proceselor:

- *Preemtive*: un proces cu prioritate mai mare care este pregătit de rulare poate întrerupe un proces de prioritate mai mică ce rulează în acel moment.

BUPT

- *Non preemtive*: procesul ce rulează în momentul respectiv va termina execuția și va returna controlul entității de planificare sau sistemului de operare.
- *Ciclice*: fiecărui proces îi este alocată o perioadă fixă de timp iar execuția continuă ciclic între procese.
- *Executive*: un algoritm specific controlează planificarea proceselor.
- *Manuale*: un operator specifică planificarea proceselor înainte de începerea execuției programului.

Metoda Booch'93 conține sugestii pentru selecția proceselor ca entități independente planificabile.

2.6.3.4 Instrumente CASE

Din categoria instrumentelor CASE care suportă metoda Booch'93 poate fi amintit produsul Rose [QUA98] [KRU98] al firmei Rational ce rulează atât pe calculatoare PC cât și pe calculatoare Unix. Produsul suportă diagramele de clase și de obiecte, diagramele de tranziție a stărilor, diagrame de arhitectură și diagrame de proces. Instrumentul face distincția între diagramele de clase și cele de obiecte și poate fi utilizat atât în etapa de analiză cât și în cea de proiectare.

2.6.4 Metoda Shlaer-Mellor

Evoluția metodei Shlaer-Mellor are ca punct de plecare tehnica de modelare a datelor [SHL88] dezvoltată pentru a trata comportamentul dinamic [SHL92] ajungând la abordarea proiectării recursive (RD) [SHL90, SHL93]. Metoda de dezvoltare este cunoscută sub denumirea de OOA/RD și reprezintă una dintre cele mai dificile metode pentru cel ce se inițiază în metode de analiză și proiectare orientate spre obiecte. Un exemplu de aplicare cu succes a metodei pentru un sistem de comunicații este prezentat în [HIN94].

Abordarea generală a aplicării metodei OOA/RD poate fi descrisă după cum urmează [SHL93]:

- partiționarea sistemului în domenii
- analiza fiecărui domeniu (OOA)
- verificarea analizei prin simulare

BUPT

- specificarea tranziției modelelor OOA în entități structurale de proiectare (specificare RD)
  construirea elementelor de tranziție (construirea elementelor RD)
- tranziția modelelor OOA pentru fiecare domeniu (aplicarea RD)

Pașii enumerați mai sus nu vor fi parcurși într-o ordine strict secvențială, pot să apară suprapuneri și iterații între pași.

2.6.4.1 Etapa OOA

Etapa OOA debutează cu o partiționare a sistemului prin crearea unui număr de partiții independente și distincte:

- *Domeniul aplicației*: descrie partea sistemului aflată în interacțiune directă cu utilizatorul sistemului livrat. Pentru sisteme mari pot să existe mai multe domenii ale aplicației.
- *Domeniile generale*: includ interfețele grafice cu utilizatorul, domenii senzor și actuator pentru sistemele de timp real, etc.
- *Domeniul arhitecturii*: reprezintă o descriere a structurii sistemului.
- *Domenii de implementare*: includ elemente dependente de tehnologie, cum ar fi sisteme de operare, rețele și limbajele de programare utilizate pentru implementare.

Rezultatul acestei partiționări este descris în schema domeniilor care indică domeniile (reprezentate sub forma unor elipse) și relațiile client-server (reprezentate sub forma unor legături orientate, denumite și punți sau *bridge*-uri) dintre aceste domenii. Descrieri textuale sunt utilizate pentru a furniza informații suplimentare referitoare la domenii și la punți.

BUPT

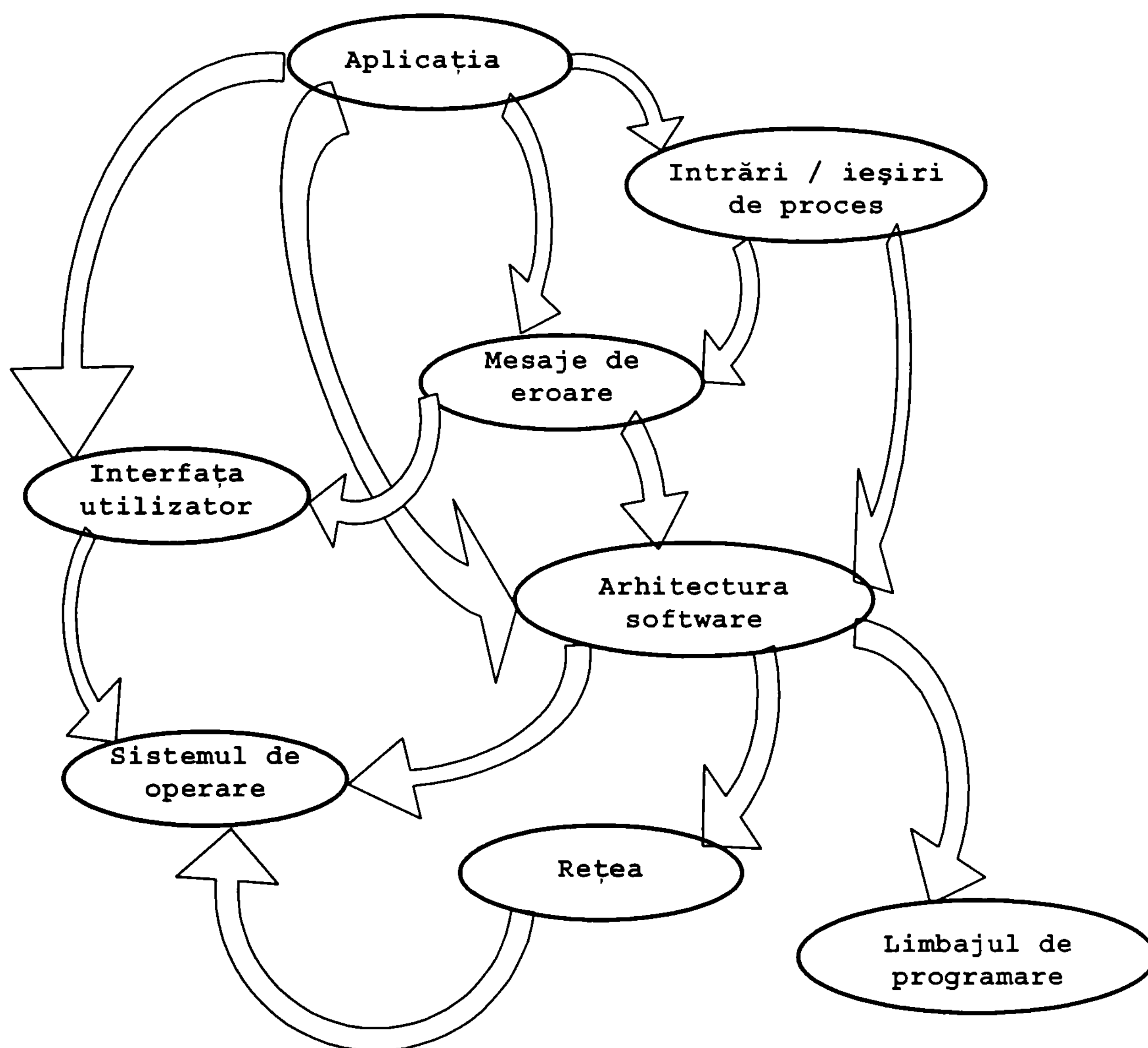


Figura 2.22 Schema domeniilor

Domeniile aplicaţiei sunt analizate în continuare utilizând versiunile orientate spre obiecte ale tehnicilor structurate. Rezultatul analizei cuprinde:

- un *model informaţional* (foarte asemănător diagramei de relaţii între entităţi, entităţile tradiţionale fiind înlocuite cu obiecte ale lumii reale)
- *diagrame de tranziţie a stărilor* pentru fiecare obiect
- *diagrame de acţiune a tranziţiei datelor* (descriu transformări pentru fiecare stare a diagramei de tranziţie a stărilor iar acţiunile pentru starea corespunzătoare sunt reprezentate de fluxul de date) pentru fiecare stare din diagrama de tranziţie a stărilor.

Pasul următor constă în verificarea analizei efectuate la nivelul fiecărui domeniu, simulând execuţia modelelor de analiză. Metoda propune următorii paşi pentru a realiza această simulare:

BUPT

- Crearea stării iniţiale dorite a sistemului utilizând valori ale datelor în cadrul modelului informaţional.
- Iniţierea comportamentului dinamic dorit cu un eveniment transmis unui model de stare.
- Executarea procesării utilizând transformările şi acţiunile din cadrul diagramelor de acţiune a tranziţiei datelor şi secvenţele din diagramele de tranziţie a stărilor corespunzătoare.
- Evaluarea valorilor de ieşire în comparaţie cu rezultatele aşteptate.

Domeniile generale rezultă pe măsura analizei domeniului aplicaţiei. Acestea pot include mecanisme pentru asigurarea persistenţei datelor, comunicaţii de date şi comunicaţii operator. Aceeaşi analiză se aplică domeniilor generale după finalizarea analizei domeniului aplicaţiei.

Ca şi modalitate de formalizare a etapei OOA au fost create 72 de reguli (condiţii) [LAN93] pentru a determină validitatea şi completivitatea etapei OOA .

2.6.4.2 Proiectarea sistemului

Proiectarea sistemului reprezintă subiectul domeniului arhitecturii. Activitatea primară a acestui domeniu este legată de tranziţia modelelor OOA în entităţi structurale de proiectare, incluzând specificaţii ale mecanismelor şi structurilor pentru manipularea datelor, controlul sistemului ca întreg şi cerinţe legate de performanţă. Crearea subsistemelor se realizează prin divizarea domeniilor în zone mai mici, gestionabile de una până la patru persoane. O altă abordare pentru creare subsistemelor constă în gruparea obiectelor înrudite în entităţi cu grad înalt de coeziune care pot forma o interfaţă bine definită cu alte subsisteme. Nu sunt incluse sugestii concrete pentru alocarea cerinţelor componentelor hardware respectiv software.

2.6.4.3 Etapa OOD

În etapa OOD, pe baza obiectelor din cadrul modelul informaţional sunt generate clasele şi atributele asociate acestora. *Modelele de stare* sunt utilizate pentru a specifica operaţiile disponibile pentru fiecare clasă (interfaţa clasei). *Modelul procesului* (utilizând diagramele acţiune de tranziţie a datelor) este utilizat pentru a specifica modul de efectuare a operaţiilor (implementarea operaţiilor la nivelul clasei).

BUPT

Etapa OOD utilizează un limbaj dedicat, OODLE (*O*bject-*O*riented *D*esign *L*anguag*E*) [SHL92] bazat pe următoarele diagrame:

- *Diagrama de moştenire*: ilustrează schema de moştenire a claselor cu nivelele ierarhice utilizate şi în cadrul altor metode.
- *Diagrama de dependenţe*: ilustrează relaţiile client-server între clase, incluzând şi relaţiile de tip "friend". În C++, o clasă friend are acces la membrii privaţi ai clasei server.
- *Diagrama clasei*: ilustrează partea vizibilă a unei clase şi este similară notaţiei utilizate de Booch pentru a descrie pachetele Ada [BOO87]. Operaţiile vizibile sunt prezentate în cadrul unor dreptunghiuri în interiorul frontierei clasei, datele ascunse sunt reprezentate în interiorul unor poligoane în interiorul frontierelor clasei iar intrările şi ieşirile sunt reprezentate deasupra şi dedesubtul unei linii orizontale de referinţă. Descrierea din cadrul poligoanelor cuprinde numele datei membre şi tipul acesteia dedesubt.
- *Schema structurii de clase*: ilustrează structura internă a implementării operaţiilor asociate unei clase. Operaţiile consemnate în cadrul diagramei clasei apar sub formă de dreptunghiuri iar elementele de date transferate între operaţii apar sub formă de poligoane.
- *Diagrama de comunicare între obiecte*: ilustrează fluxul evenimentelor transmise între obiectele cele mai importante. Fiecare eveniment este prefixat de o etichetă ce va fi regăsită în tabela de evenimente ale obiectului corespunzător, tabelă ce conţine lista de evenimente asociate acestuia. Obiectele interne sunt reprezentate sub forma unor dreptunghiuri cu colţuri rotunjite iar obiectele terminale apar sub forma unor dreptunghiuri cu colţuri ascuţite. Fiecare obiect este considerat a fi un element concurent care poate fi utilizat ca bază pentru determinarea de subsisteme sau de task-uri aflate în competiţie pentru serviciile unei unităţi centrale de prelucrare.

2.6.4.4 Proiectarea recursivă

Proiectarea recursivă (RD) reprezintă un concept specific metodei Shlaer-Mellor. În practica programării, noţiunea de recursivitate implică apelul unei proceduri de către ea însăşi până când este îndeplinit un criteriu de oprire. În cazul metodei Shlaer-Mellor, un set de reguli se aplică în mod repetat (recursiv) pentru fiecare domeniu până când toate regulile au fost aplicate (criteriul de oprire). Regulile sunt utilizate pentru a transforma obiectele OOA în obiecte OOD. Domeniile primare pentru aceste transformări sunt domeniul aplicaţiei şi domeniile generale. Aplicarea aceluiaşi set de reguli pentru toate aceste domenii asigură o tranziţie uniformă de la analiză la proiectare, permiţând o anumită automatizare a acestui proces.

BUPT

Avantajul primar al metodei OOA/RD constă în faptul că o modificare în procesul de tranziție de la entități OOA la entități OOD necesită doar modificarea regulii de tranziție care va fi aplicată ulterior tuturor domeniilor. O perspectivă asupra metodei OOA/RD [MEL93] este ilustrată în figura următoare. Regulile de proiectare definite în domeniul arhitecturii software sunt aplicate modelelor OOA în fiecare domeniu. Se pot utiliza şabloane pentru generarea unui schelet de cod plecând de la entitățile OOD. Acest schelet va fi completat cu codul propriu-zis în faza de implementare.

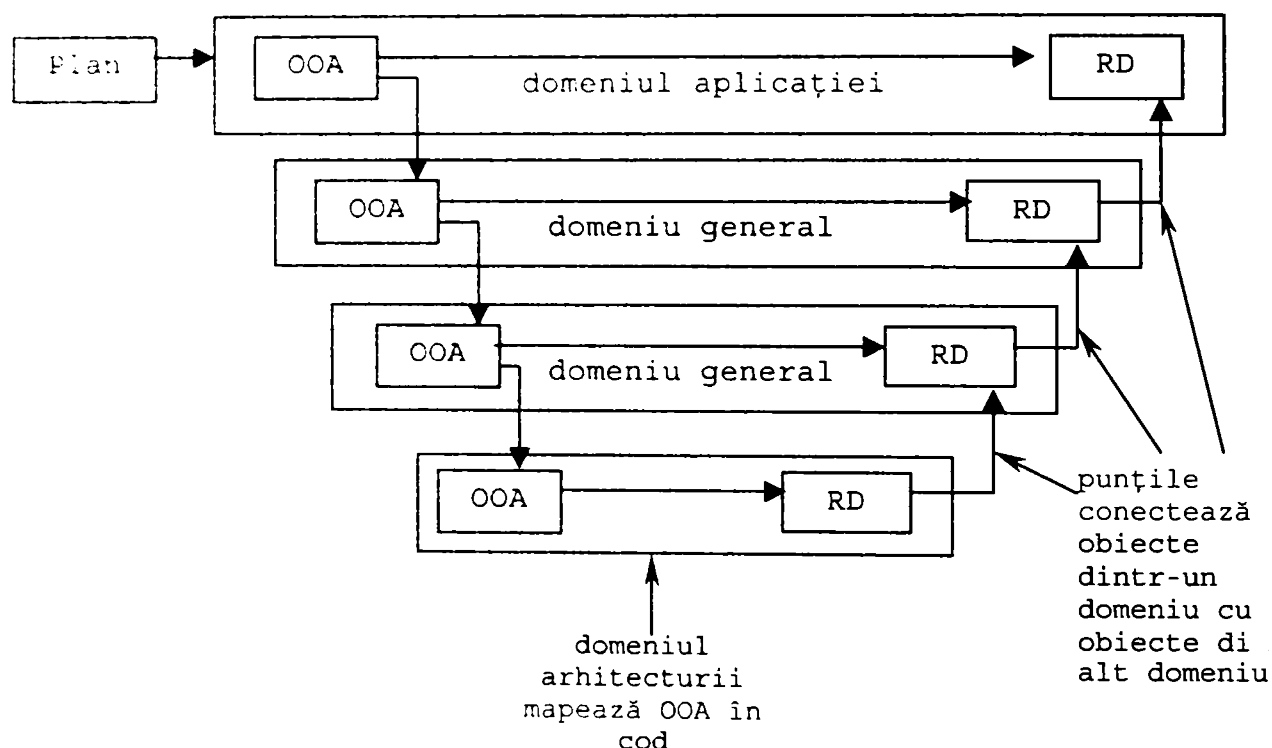


Figura 2.23 Aplicarea metodei RD domeniilor de analiză

2.6.4.5 Aspecte legate de operarea în timp real şi de concurență

Modelul de comunicare a obiectelor (OCM) reprezintă suportul pentru operarea în timp real şi pentru concurență în ceea ce priveşte metoda Shlaer-Mellor. Acesta ilustrează obiectele şi mesajele (evenimentele) transmise între ele. Fiecare obiect se presupune a fi o entitate independentă ce poate opera simultan cu celelalte obiecte. Modelul de comunicare între obiecte include elemente de comunicare şi de sincronizare între obiecte.

Selecția proceselor are loc prin intermediul unui mecanism orientat spre nivele:

- Obiectele interfațate cu hardware-ul sunt plasate la nivelul cel mai coborât şi ascund mecanismul de interfațare.

BUPT

- Obiectele care manipulează evenimente sistem ale lumii reale şi ghidează obiecte la nivelul cel mai coborât sunt plasate la nivelul cel mai înalt şi sunt considerate actori în sensul orientării spre obiecte (asupra lor se acţionează de către utilizatori şi de către sisteme externe interfaţate cu sistemul care se dezvoltă).
- Obiectele nivelului intermediar sunt considerate agenţi; ele de obicei recepţionează evenimente de la obiectele de la nivelul superior şi le direcţionează spre obiecte situate la nivelul cel mai coborât.

2.6.4.6 Instrumente CASE

Paleta de instrumente CASE care implementează elemente ale metodei OOA/RD include produsele Team Work de la Cadre, ObjectiveAnalyst de la Objective Solutions, TurboCASE/Sys [HAT97] de la StructSoft şi ObjectBench de la SES.

2.6.5 Metoda RDD (Responsibility-Driven Design)

Metoda RDD, orientată spre responsabilităţi se aplică atât etapei OOA cât şi etapei OOD. Metoda a fost descrisă de către Wirfs-Brock [WIR90] şi se bazează pe utilizarea de cartele CRC (*C*lase, *R*esponsabilităţi şi *C*olaborări). O cartelă CRC include numele clasei, responsabilităţile (operaţiile) asociate şi colaborările cu alte clase (modul în care clasa interacţionează cu alte clase).

2.6.5.1 Etapa OOA

Etapa OOA a metodei RDD presupune existenţa specificaţiilor referitoare la cerinţele sistemului sau a unei echipe de experţi în domeniu care sunt capabili să transforme aceste cerinţe în obiecte ale lumii reale şi în clase. O cartelă CRC este pregătită pentru fiecare entitate a lumii reale. Entităţii îi este asociat un nume de clasă şi un set de operaţii (responsabilităţi). Colaboratorii clasei sunt de asemenea consemnaţi pe cartelă, specificându-se tipurile de interfaţă pe care clasa dată le realizează cu alte clase. Colaborările sunt văzute ca şi relaţii client-server, clasa a cărei nume este specificat pe cartelă fiind serverul iar clasele enumerate ca şi colaboratori fiind clienţii. Figura următoare ilustrează un exemplu de cartelă CRC pentru clasa client în cadrul aplicaţiei ATM.

BUPT

Clasa: Client	
Responsabilități:	Colaborări:
Furnizează cartela ATM	ATM
Extrage cartela ATM	ATM
Specifică suma de ridicat	ATM
Acceptă numerarul	ATM
Furnizează număr de cont	ATM
Acceptă factura	ATM
Acceptă cererea de continuare	ATM

Figura 2.24 Cartelă CRC pentru clasa Client

Următoarea activitate în cadrul acestei abordări orientate spre responsabilități constă în gruparea responsabilităților în contracte. Un *contract* reprezintă interfața serverului, adică setul de mesaje (apeluri) pe care clientul poate să le transmită serverului. Semnificația primară a unui contract constă în faptul că în faza de tranziție de la analiză la proiectare contractul devine interfața vizibilă a clasei. Considerațiile legate de contractul unei clase pot fi implementate în C++ utilizând cuvintele cheie `friend` și `protected`.

Procesul de realizare a cartelelor CRC este unul iterativ, dat fiind faptul că o colaborare cu o clasă nu poate fi complet specificată înainte ca această clasă să fie definită.

Notația utilizată în cadrul metodei CRC nu este deosebit de puternică dar metoda reprezintă un instrument adecvat pentru determinarea inițială a claselor, pentru identificarea entităților sistemului care pot fi văzute ca și obiecte ale lumii reale și a relațiilor dintre acestea. Un aspect interesant al metodei este faptul că se concentrează doar asupra operațiilor și a interfețelor claselor. Atributele nu sunt importante pentru deciziile luate la nivel de analiză și de proiectare. Ele pot să constituie adnotări la cartelele CRC sau pot fi păstrate în dicționare de date.

2.6.5.2 Etapa OOD

Metoda RDD nu face distincție între etapa OOA și etapa OOD. Ea poate fi utilizată împreună cu orice altă metodă de analiză și proiectare orientată spre obiecte pentru a determina setul inițial de clase și obiecte în etapele OOA și OOD.

BUPT

2.6.5.3 Proiectarea sistemului

Din punctul de vedere al etapei de proiectare a sistemului metoda poate fi aplicată pentru alocarea claselor la subsisteme, dar nu cuprinde specificații de alocare a cerințelor componentelor hardware și software. Diagrama de colaborare prezentată în figura de mai jos poate fi utilizată atât pentru subsisteme cât și pentru interfețe individuale ale claselor. Fiecare dreptunghi reprezintă o clasă iar legăturile orientate reprezintă relațiile client-server. Numărul asociat fiecărui semicerc reprezintă numărul contractului și reprezintă colecția de responsabilități pentru respectiva clasă.

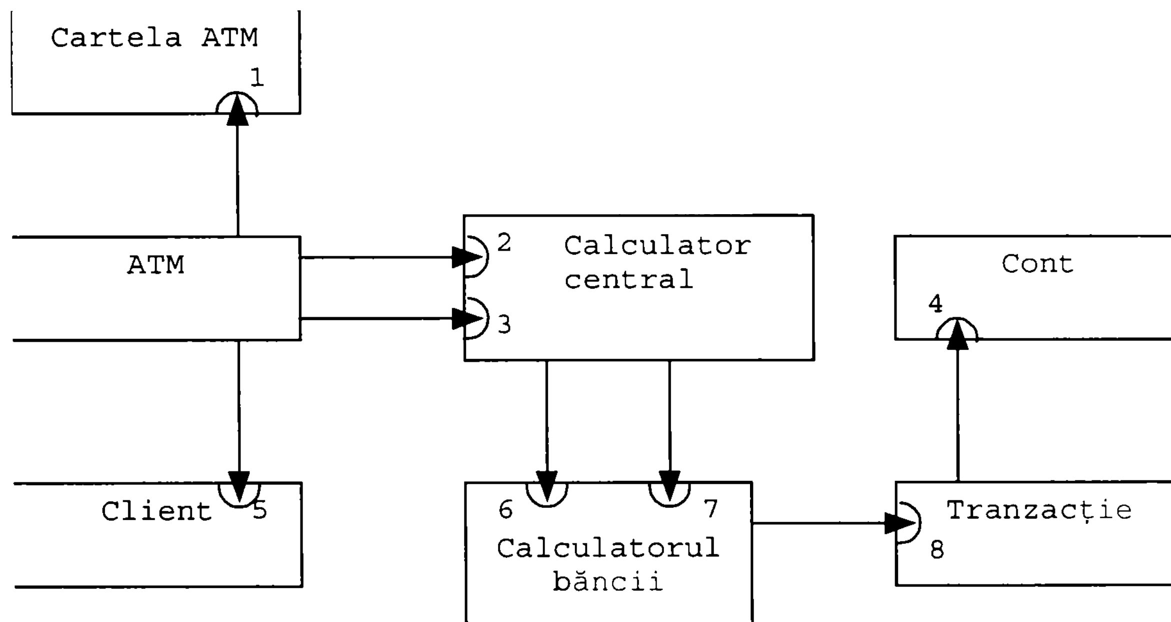


Figura 2.25 Diagrama de colaborări pentru aplicația ATM

2.6.6 Metoda Coad-Yourdon

Elementele metodei Coad-Yourdon specifice etapei de analiză orientată spre obiecte sunt prezentate în [COA90 și COA91] iar cele specifice etapei de proiectare sunt prezentate în [COA91a]. Metoda Coad-Yourdon diferă de metoda Shlaer-Mellor prin faptul că numărul de reguli aplicat este redus iar diagramele utilizate sunt puține la număr și simple.

2.6.6.1 Etapa OOA

Modelul OOA este descris prin cinci nivele distincte:

- *nivelul de clase și obiecte*: clasele și obiectele sunt identificate ca abstractizări ale obiectelor lumii reale prin identificarea de structuri, sisteme, dispozitive, evenimente,

BUPT

roluri, proceduri operaţionale, unităţi de organizare, etc. Clasele şi obiectele (instanţele) sunt descrise cu ajutorul unui dreptunghi interior pentru clasă şi a unuia exterior pentru instanţă. În cadrul acestei pictograme pot fi consemnate atât atributele cât şi operaţiile

- *nivelul de structură*: structura de generalizare-specializare descrie relaţii de moştenire (is-a) între clase. Structura de parte componentă descrie relaţiile de agregare (has-a) între clase. Figura următoare ilustrează notaţiile utilizate pentru descrierea acestor structuri în cadrul aplicaţiei ATM.

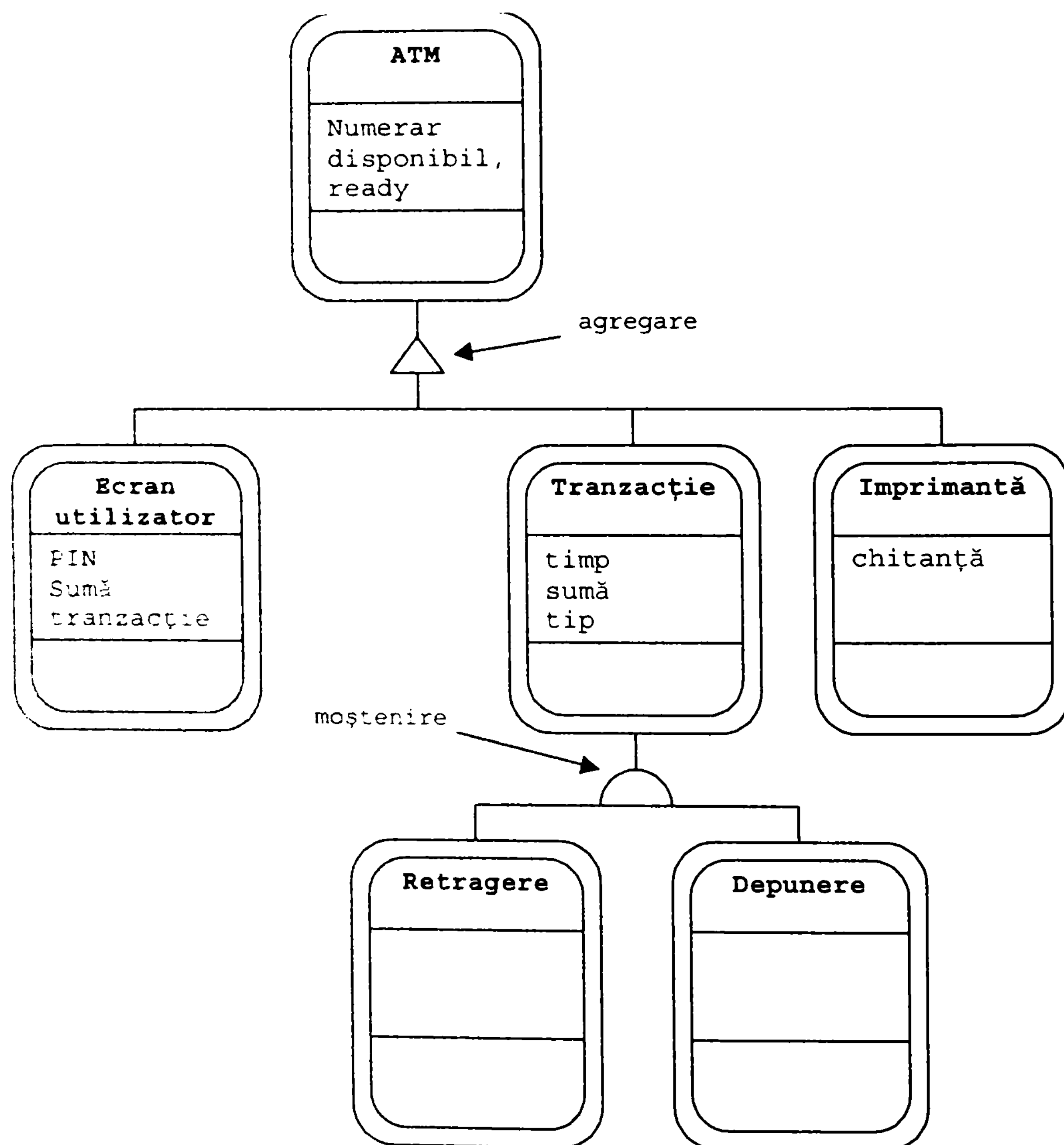


Figura 2.26 Structuri de clase

BUPT

- *conexiunile între instanțe*: reprezintă echivalentul modelului static al obiectelor utilizat în cadrul altor metode orientate spre obiecte şi include notații de cardinalitate după cum se ilustrează în cadrul figurii următoare.

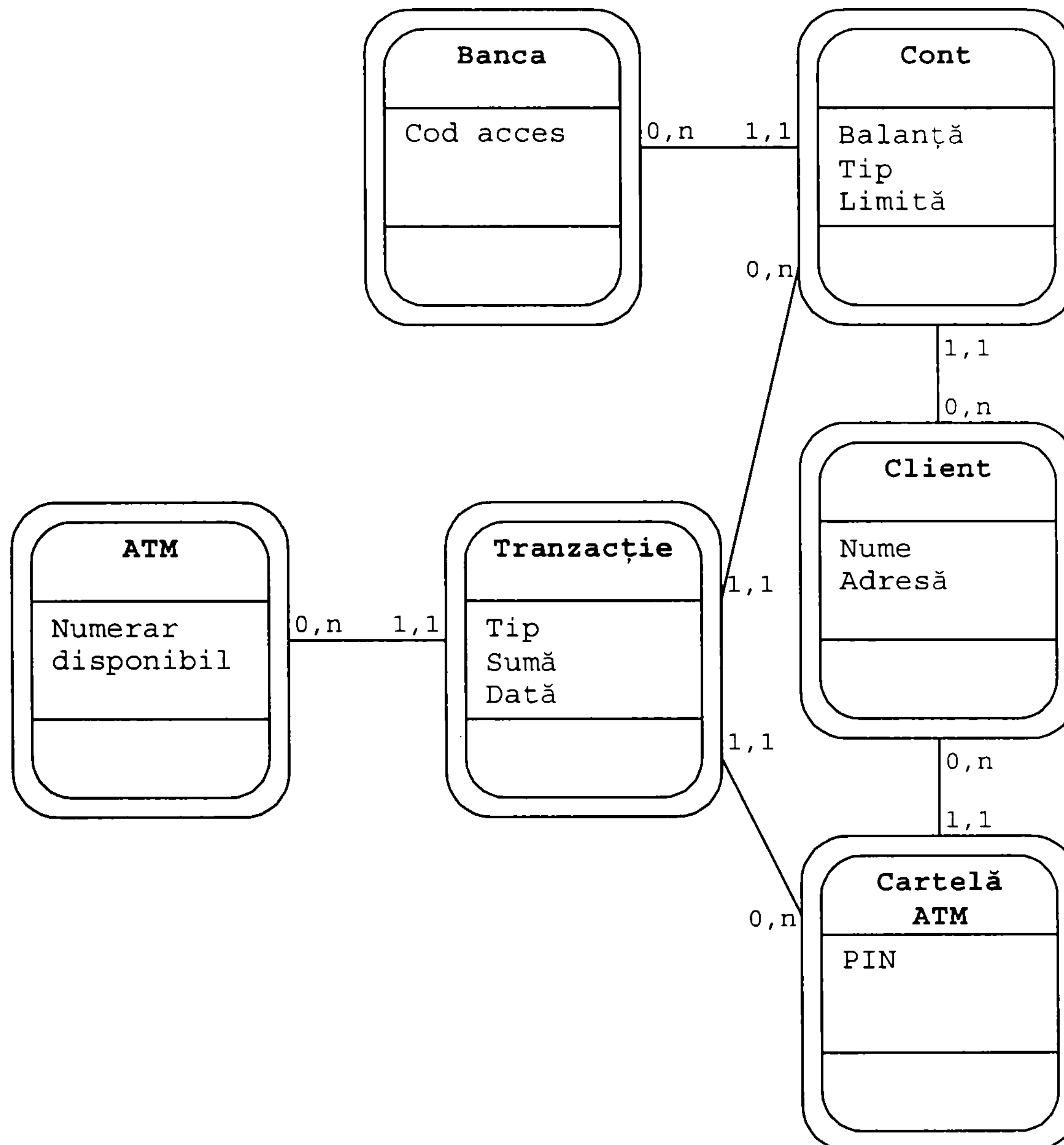


Figura 2.27 Conexiuni între instanțe pentru aplicația ATM

- *nivelul subiectului*: reprezintă uzual o grupare de cinci până la nouă obiecte având ca scop reducerea numărului de obiectelor la un număr gestionabil şi reducerea nivelelor multiple de diagrame. O modalitate de a selecta un subiect este de a alege clasa situată la nivelul cel mai înalt din fiecare structură iar apoi cuprinderea în structură a fiecărei clase ce nu face parte din nici o structură.

- *nivelul atributelor*: atributele clasei sunt identificate ca şi variabile de stare asociate fiecărui obiect. Ele sunt consemnate în partea centrală a pictogramei reprezentând clasa şi

BUPT

obiectul. Sugestii pentru specificarea atributelor se referă la unităţi, domeniu de valori, precizii şi valori implicite.

- *nivelul serviciilor*: serviciile (operaţiile) sunt definite ca elemente ce afectează comportamentul obiectului căruia îi sunt asociate. Serviciile sunt create aplicând următorii paşi:
  - identificare stărilor obiectului
  - identificarea serviciilor necesare
  - identificarea legăturilor prin mesaje
  - specificarea serviciilor
  - realizarea documentaţiei OOA

2.6.6.2 Proiectarea sistemului

În cadrul metodei nu este descrisă explicit o activitate de proiectare a sistemului ci se face referinţă la "subiecte" ca echivalent al subsistemelor din cadrul metodelor OOSE, OMT, Shlaer-Mellor şi a categoriilor de clase din metoda Booch'93. Nu sunt incluse specificaţii pentru alocarea cerinţelor componentelor hardware şi software.

2.6.6.3 Etapa OOD

Metoda Coad-Yourdon se bazează pe o notaţie uniformă a modelelor OOA şi OOD. Pretinsul avantaj major al metodei constă în faptul că nu există o tranziţie de la etapa OOA la etapa OOD, astfel că în etapa OOD sunt rafinate clasele, obiectele şi relaţiile dintre ele determinate în etapa OOA. Clase şi obiecte adiţionale rezultate în această etapă sunt adăugate entităţilor de proiectare. Etapa OOD realizează o aplicare a paradigmei de agregare pentru a descrie următoarele patru modele principale de proiectare:

- *Componenta domeniului problemei*: rezultatele etapei OOA sunt plasate direct în această componentă.
- *Componenta de interacţiune umană*: include intrările şi ieşirile necesare interacţiunii om-maşină
- *Componenta de gestiune a task-urilor*: include definirea task-urilor, mecanismele de comunicare şi controlul concurenţei, precum şi consideraţii de alocare a resurselor hardware şi protocoalele de comunicaţie cu sisteme şi dispozitive externe.

BUPT

- *Componenta de gestiune a datelor*: include accesul şi gestiunea datelor persistente.

Activitatea principală a etapei OOD constă în crearea celor patru modele enumerate mai sus, în termenii celor cinci nivele utilizate în cadrul etapei OOA: nivelul de clase şi obiecte, de structură, al subiectului, al atributelor şi a serviciilor. Cea mai importantă referinţă pentru etapa OOD ([COA91a]) furnizează anumite elemente de îndrumare pentru această activitate, dar rezultatele etapei OOD sunt dificil de determinat, dată fiind absenţa criteriilor de oprire. Rezultatul acestei etape se regăseşte în clasele şi obiectele rafinate conţinute în domeniul problemei precum şi în clase şi obiecte adiţionale din cadrul celorlalte trei componente.

Sunt prezentate criterii de evaluare a gradului de cuplare şi de coeziune între entităţile OOD. Astfel, gradul de coeziune în contextul paradigmei de generalizare-specializare implicată de moştenire se măsoară prin numărul de atribute şi de servicii pe care le moşteneşte clasa specializată. Îndeplinirea dezideratului de cuplare redusă între module se sugerează a se realiza prin limitarea la cel mult trei a parametrilor ce fac parte dintr-un mesaj (procedură sau apel de funcţie). Coeziunea puternică pentru un serviciu se obţine atunci când serviciul efectuează o singură operaţie. Coeziunea puternică în cadrul unei clase se reflectă în servicii care efectuează operaţii referitoare la aceeaşi abstractizare. Coeziunea în cadrul relaţiei de generalizare-specializare este cu atât mai puternică cu cât relaţia dintre clasa specializată şi clasa generală este mai strânsă.

2.6.6.4 Aspecte legate de operarea în timp real şi de concurenţă

Metoda Coad-Yourdon conţine referiri la aspecte ale operării în timp real şi concurenţă. Următoarele reguli de selecţie a taskurilor sunt utilizate pentru a popula componenta de gestiune a taskurilor:

- *Identificarea taskurilor declanşate de evenimente*: aceste task-uri sunt de obicei declanşate de evenimente provenind de la un dispozitiv sau sistem extern sub forma unei întreruperi.
- *Identificarea taskurilor declanşate de ceas*: aceste taskuri sunt executate în intervale de timp definite în raport cu ceasul sistemului şi pot fi periodice sau aperiodice.
- *Identificarea taskurilor prioritare şi a taskurilor critice*: taskuri utilizate în cadrul sistemelor de timp real necesită nivel ridicat de prioritate pentru a preveni pierderea de date critice. Taskurile ce rulează în background primesc prioritate scăzută iar execuţia lor poate avea loc în partaj de timp.
- *Identificarea unui coordonator*: un task coordonator încapsulează activitatea de coordonare şi poate fi adăugat unui sistem concurent cu trei sau mai multe taskuri. Noul

BUPT

task încapsulează activitatea de coordonare dar are ca efect nedorit solicitarea suplimentară a sistemului.

- *Stabilirea necesității fiecărui task*: numărul total de taskuri din sistem (care sunt în competiție pentru timpul aceleiași unități centrale de prelucrare) trebuie să fie minim, dat fiind faptul că fiecare task suplimentar are ca efect încărcarea suplimentară a sistemului.
- *Definirea fiecărui task*: fiecare task este definit în termeni de coordonare, periodicitate, priorități, comunicație și servicii puse la dispoziție.

2.6.6.5 Instrumente CASE

Din categoria instrumentelor CASE care suportă metoda Coad-Yourdon se poate aminti produsul ObjecTool ce rulează pe platforme PC, Macintosh și sisteme Unix.

2.6.7 UML – un standard în formare în domeniul analizei și proiectării orientate spre obiecte

Metodele de analiză și proiectare prezentate până în acest moment prezintă un dezavantaj major, legat de lipsa de unitate în ceea ce privește notațiile utilizate. Acest fapt generează confuzii, îngreunează trecerea de la o metodă la alta în cazuri în care o anumită metodă nu oferă soluția cea mai bună pentru aplicația dezvoltată.

O posibilă soluție pentru aceste probleme este oferită de grupul OMG (*Object Managment Group*). Acesta reprezintă o organizație internațională formată din peste 800 de membri, incluzând distribuitori de sisteme informatice, dezvoltatori software și utilizatori. Fondată în 1989, organizația promovează teoria și practica tehnologiei orientate spre obiecte în dezvoltarea de produse software. Activitatea organizației include stabilirea de indicații pentru industrie și specificații de gestiune a obiectelor pentru a furniza o structură comună pentru dezvoltarea aplicațiilor. Scopurile principale sunt reutilizabilitatea, portabilitatea și interoperabilitatea software-ului bazat pe obiecte în medii distribuite, eterogene. Conformitatea cu aceste specificații face posibilă dezvoltarea unui mediu eterogen de aplicații peste toate platformele și sistemele de operare importante. Obiectivele OMG sunt de a încuraja dezvoltarea tehnologiei orientate spre obiecte și de a-i influența direcția de dezvoltare prin stabilirea arhitecturii de gestiune a obiectelor (OMA = *Object Management Architecture*), constituind infrastructura conceptuală pe care se bazează toate specificațiile OMG.

BUPT

Soluția propusă de OMG este limbajul unificat de modelare UML (*Unified Modelling Language*) [BOO99]. Acesta pune la dispoziția arhitecților de sistem ce lucrează în domeniul analizei şi proiectării orientate spre obiecte un limbaj consistent pentru specificarea, vizualizarea, construcția şi documentarea elementelor constitutive ale sistemelor software, precum şi pentru alte domenii de modelare. Această specificație reprezintă convergența celor mai bune practici din industria tehnologiilor orientate spre obiecte. UML reprezintă succesorul a trei dintre cele mai performante metode orientate spre obiecte: Booch, OMT şi OOSE. UML reprezintă mai mult decât reuniunea celor trei limbaje de modelare, dat fiind faptul că include modalități de exprimare menite să abordeze probleme de modelare pe care aceste trei metode nu le rezolvau.

Unul dintre principalele scopuri ale UML este de a contribui la dezvoltarea industriei prin facilitarea interoperabilității între instrumentele vizuale de modelare a obiectelor.

UML satisface următoarele cerințe [OMG99]:

- Definirea formală a unui metamodel comun de analiză şi proiectare orientat spre obiecte (OA&D) pentru a reprezenta semantica modelelor OA&D, incluzând modele statice, modele comportamentale, modele de utilizare şi modele arhitecturale.
- Specificații IDL pentru mecanisme de interschimbare a modelelor între instrumente OA&D. Acest document cuprinde un set de interfețe IDL care suportă construcția şi parcurgerea dinamică a unui model utilizator.
- O notație direct lizibilă pentru reprezentarea modelelor OA&D. Notația UML reprezintă o sintaxă grafică elegantă pentru a exprima în mod consistent bogata semantică UML. Notația reprezintă un element esențial pentru modelarea OA&D şi pentru UML.

UML a fost dezvoltat de către Rational Software şi de către partenerii săi. Multe companii incorporează UML ca standard al procesului de dezvoltare a produselor lor, acoperind diversele domenii, cum ar fi modelarea, managementul cerințelor, analiza şi proiectarea, programarea şi testarea.

2.7 Concluzii. Contribuții

Dezvoltarea teoretică a conceptului de programare orientată spre obiecte ca succesor al programării structurate în evoluția limbajelor de programare a permis definirea şi dezvoltarea unui set de paradigme ce se regăsesc în implementările unor limbaje de programare orientate spre obiecte cum ar fi Smalltalk, Eiffel, C++. Aceste paradigme au o importanță deosebită pentru

BUPT

intregul proces de dezvoltare a unei aplicaţii, atât în etapa de analiză şi proiectare cât şi în cea de programare.

Actualul capitol are menirea de a identifica, de a defini şi de a detalia acele aspecte ale conceptului de orientare spre obiecte ce vor putea fi utilizate în cadrul procesului de analiză şi proiectare orientată spre obiecte.

O importanţă deosebită este acordată definiţiilor conceptului de clasă şi a celui de obiect, ca elemente de bază ale întregului proces de dezvoltare software.

Modelele statice, dinamice şi funcţionale ce stau la baza metodelor de analiză şi proiectare software sunt detaliate insistând asupra modului în care redau informaţia ce conduce la generarea configuraţiei finale de clase şi obiecte ce se regăseşte în cadrul codului sursă al aplicaţiei.

Informaţiile prezentate in cadrul actualului capitol vor constitui baza pentru metoda de dezvoltare software orientată spre obiecte prezentată în capitolul următor. Această metodă va încerca să sintetizeze câteva din avantajele prezentate în cadrul metodelor de analiză şi proiectare consacrate prezentate până acum. Selecţia soluţiei potrivite pentru procesul de dezvoltare a unui anumit tip de aplicaţie este facilitată de cunoaşterea acestor metode, a avantajelor şi a dezavantajelor specifice lor.

Actualul capitol reprezintă în primul rând un studiu bibliografic, dar îşi propune să structureze informaţia pentru a crea premisele abordării orientate spre obiecte a întregului proces de dezvoltare software. Sunt propuse abordări şi sinteze originale, dintre care amintim:

- Identificarea şi definirea principalelor paradigme ale orientării spre obiecte urmărind modul in care acestea îşi găsesc suport nativ la nivelul implementării în cadrul limbajului de programare C++.
- Identificarea definiţiilor conceptului de clasă şi a celui de obiect subliniind conexiunea cu paradigmele enunţate anterior.
- Prezentarea unor modalităţi de creare a claselor şi a obiectelor în contextul unei aplicaţii.
- Prezentarea principalelor modele ce descriu un context al obiectelor unei aplicaţii din punct de vedere static, dinamic şi funcţional
- Realizarea unui studiu comparativ al câtorva din cele mai cunoscute metode de analiză, proiectare şi implementare orientate spre obiecte analizând în cadrul fiecăreia dintre acestea etapele:
  - OOA
  - Proiectarea sistemului
  - OOD

BUPT

pornind de la exemplul oferit de aplicația ATM.

- Analiza măsurii în care diversele metode oferă sprijin pentru implementarea sistemelor de timp real şi a aspectelor legate de procese concurente.
- Identificarea unor instrumente CASE care implementează aceste metode automatizând procesul de dezvoltare software.

Abordarea prezentă în cadrul acestui capitol pune un accent deosebit pe potențialul de reutilizare şi de mentenabilitate pe care îl oferă o separare clară între etapele consecutive ale procesului de dezvoltare software orientat spre obiecte. Acest fapt se reflectă în modul în care a fost analizată fiecare dintre metodele de dezvoltare software prezentate.

BUPT

Abordare bazată pe tehnologia orientată spre obiecte a procesului de dezvoltare software

3.1 Introducere

Evoluția limbajelor de programare consemnează programarea orientată spre obiecte, succesor al programării structurate, ca soluție credibilă pentru îmbunătățirea calității şi a fiabilității produselor software. În sprijinul acestei afirmații vin avantajele pe cere le oferă trecerea la programarea orientată spre obiecte [NIE95]:

- *Grad înalt de reutilizabilitate*: entitățile orientate spre obiecte pot fi structurate prin intermediul interfețelor lor, permițând mai multor aplicații din acelaşi domeniu să utilizeze aceleaşi entități. Un exemplu în acest sens îl constituie bibliotecile de clase.
- *Comunicație îmbunătățită cu beneficiarul*: „obiectele" descriu entități ale lumii reale. Acestea pot constitui baza de discuție cu beneficiarul produsului software.
- *Scurtarea timpului de dezvoltare*: abordările orientate spre obiecte sunt adecvate prototipizării rapide pe măsura dezvoltării conceptului, pe care beneficiarul îl poate inspecta şi confirma înainte ca echipa de dezvoltare să fi abordat o cale greşită.
- *Uşurința întreținerii*: corespondența strânsă între conceptele definite de beneficiar şi entitățile software implementate face ca modificările cerințelor şi extensiile să poată fi izolate la nivelul unui număr redus de obiecte diminuând astfel efortul legat de codare şi de testare.

Abordarea orientată spre obiecte poate fi deci utilizată chiar începând cu faza de definire a problemei, obiectele constituind atomii unui limbaj comun clientului nespecialist în programare şi specialistului implicat în realizarea produsului finit. Tehnologia orientată spre obiecte a permis delimitarea unor etape precise în dezvoltarea produsului: analiza (OOA = *Object Oriented Analysis*), proiectarea (OOD = *Object Oriented Design*), respectiv implementarea (OOP = *Object Oriented Programming*) folosind un limbaj de programare orientat spre obiecte. Aceste etape utilizează un set de paradigme legate de tehnologia orientată spre obiecte, paradigme enumerate în cadrul capitolului precedent.

BUPT

Actualul capitol propune o abordare completă a procesului de dezvoltare software pornind de la paradigmele, modelele şi metodele de dezvoltare software prezentate în cadrul capitolului anterior. Această abordare este enunţată aici la un nivel preponderent teoretic, exemplele utilizate vizând aplicaţii de mici dimensiuni pentru a demonstra consistenţa ideilor enunţate. Aplicarea în practică a procesului de dezvoltare software propus aici va fi ilustrată în cadrul capitolului următor, unde sunt urmăriţi paşii dezvoltării unei biblioteci de ferestre şi a unui editor 2D, ambele integrate în cadrul unui sistem de proiectare şi execuţie asistată de calculator pentru construcţii în lemn.

3.2 Comparaţie între metodele structurate şi metodele orientate spre obiecte

O justificare a alegerii unui proces de dezvoltare orientat spre obiecte pentru produsul software dezvoltat în cadrul capitolului următor este dată de analiza comparativă a metodelor de proiectare structurate şi a celor orientate spre obiecte.

Programarea structurată reprezintă precursorul programării orientate spre obiecte. Dezvoltarea metodelor de analiză şi proiectare a urmat apariţiei limbajelor de programare. Aceste metode au ca scop optimizarea procesului de dezvoltare a produselor software, asigurând o calitate şi o mentenabilitate corespunzătoare a acestuia.

Metodele structurate sunt orientate spre funcţii, tratând entităţile de date separat de funcţii. Figura următoare ilustrează un model de proiectare structurată.

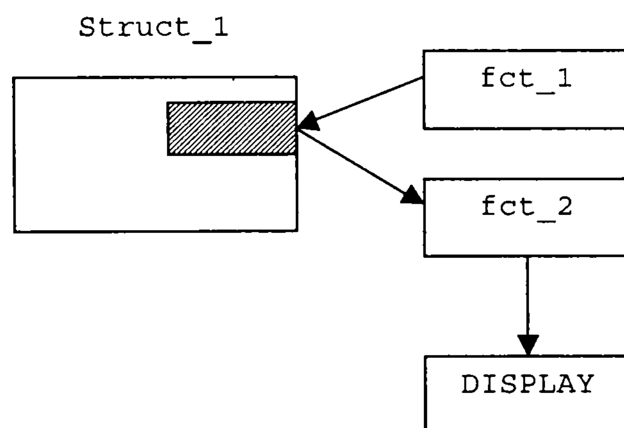


Figura 3.1 Model de proiectare structurată

Funcţiile `fct_1()` şi `fct_2()` accesează un element din cadrul structurii `Struct_1`. Eventualele modificări în cadrul acestei structuri au ca efect necesitatea modificării funcţiilor `fct_1()` şi `fct_2()`. Tehnici de proiectare structurată sunt abordate în cadrul lucrărilor:

BUPT

- *Friendly Interface for Wood Manufacturing C.A.D.*, prezentată de autor în cadrul conferinței "International Conference on Technical Informatics", Timişoara, 16-19. 11. 1994 [FAR94]
- *Dialog Manager for Applications Running under MS*-DOS, prezentată de autor la Sesiunea de comunicări ştiinţifice ACADRES'96, Fundaţia Universitară "Banatul", Universitatea Banatului Timişoara, 16 noiembrie 1996 [FAR96]

Implementarea dialog-managerului prezentat în cadrul lucrării [FAR96] reprezintă o abordare structurată ce se apropie de paradigme ale orientării spre obiecte, îngrădită fiind de limitările inerente limbajului C.

Modelul de proiectare orientat spre obiecte este prezentat în figura următoare. Se remarcă faptul că singura interacţiune între obiecte şi restul aplicaţiei are loc prin intermediul interfeţei acestora, fiind posibilă astfel eliminarea necesităţii de intervenţie în codul apelant în eventualitatea modificărilor la nivelul obiectelor. Obiectele încapsulate ascund structurile de date A şi B.

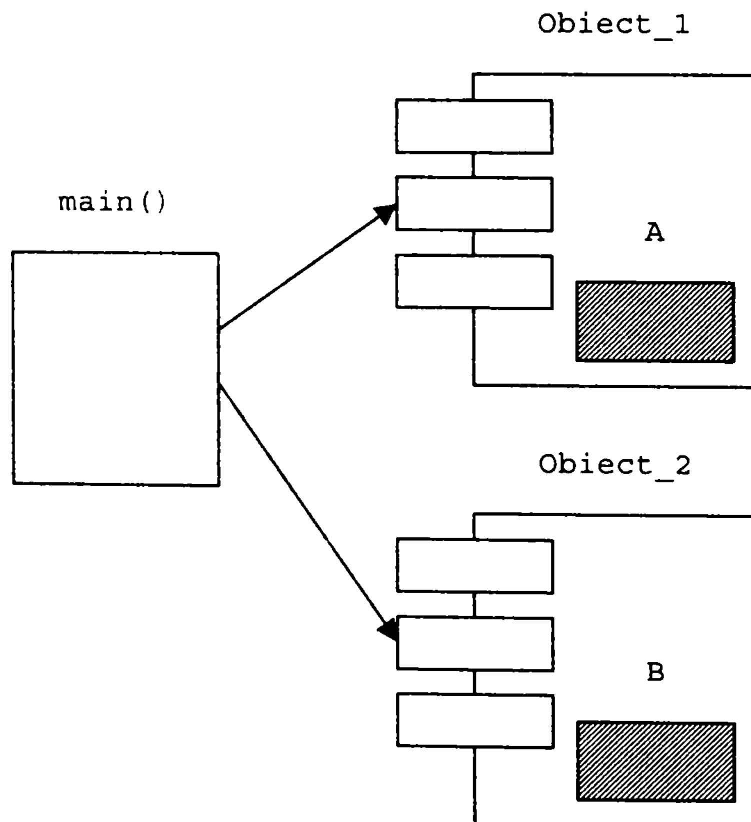


Figura 3.2 Model de proiectare orientat spre obiecte

Contribuţii personale în domeniul utilizării tehnologiei obiectuale în procesul de dezvoltare a unei aplicaţii se regăsesc în lucrarea:

BUPT

- *Object-Oriented Development Process based upon Use Cases for a 2D Editor in a CAD/CAM System*, prezentată de autor în cadrul sesiunii de comunicări științifice CONTI'2000, 12-13 oct.2000, Timişoara [SAV2000].

3.3 Procesul de dezvoltare a unei aplicații orientate spre obiecte

Dezvoltarea unui produs software de complexitate ridicată presupune o descompunere a sa în subsisteme. Această descompunere poate fi:

- *funcțională* (procedurală sau algoritmică) care generează nivele ierarhice de funcții
- *orientată spre obiecte* ce generează entități ce modelează obiectele lumii reale. Totalitatea acestor entități formează sistemul ce se implementează.

Soluția pentru procesul de dezvoltare software propusă în cadrul actualei lucrări se bazează pe descompunerea orientată spre obiecte. Etapele parcurse în cadrul acestui proces se bazează pe schema procesului de dezvoltare a unui sistem informatic [NIE95], prezentată în figura următoare.

BUPT

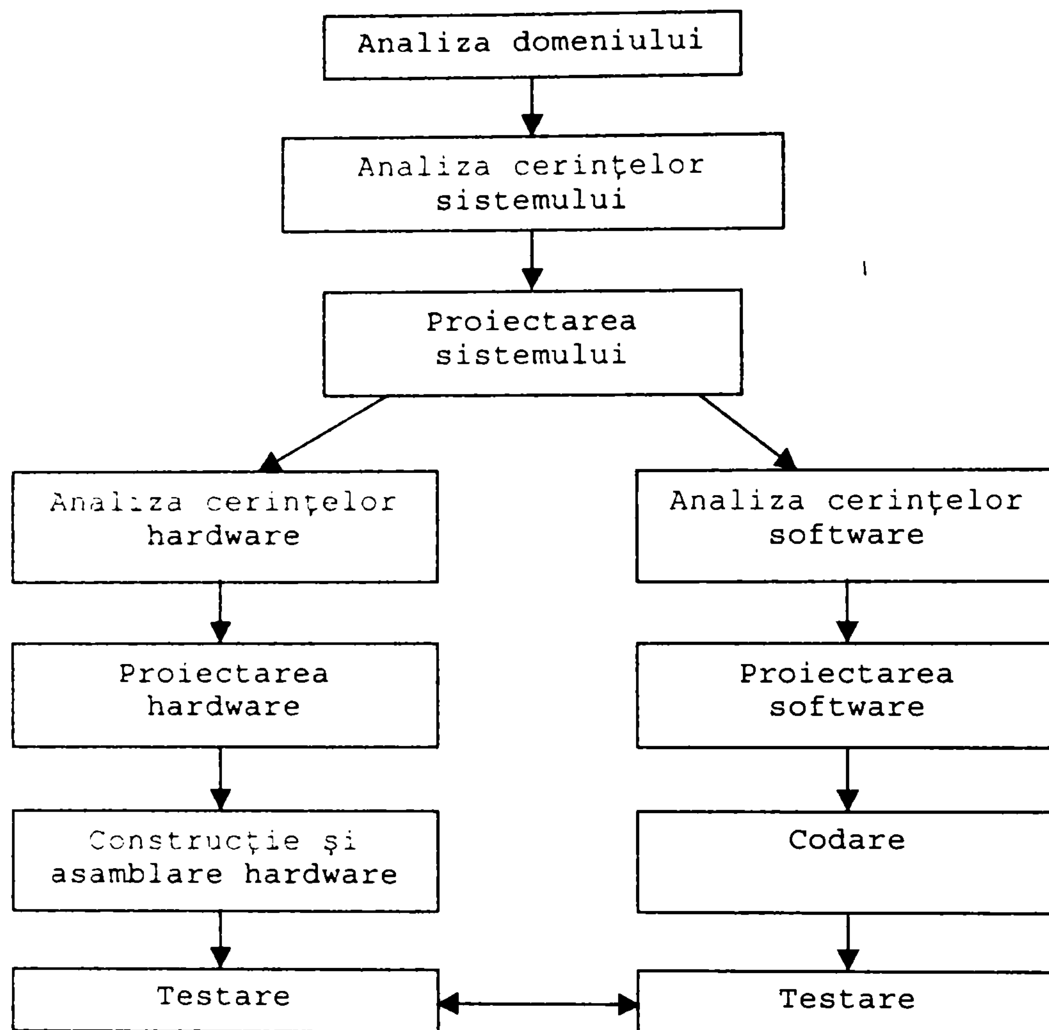


Figura 3.3 Etapele de dezvoltare ale unui sistem informatic

După cum rezultă şi din această figură, abordarea orientată spre obiecte este capabilă să furnizeze soluţii complete pentru implementarea unui sistem informatic, atât din punct de vedere al arhitecturii sistemului cât şi din cel al implementării software.

Tratarea unitară a componentei hardware şi a celei software în cadrul tehnologiei orientate spre obiecte este posibilă datorită faptului că programarea orientată spre obiecte poate fi privită ca o transpunere în domeniul software a conceptului de arhitectură modulară a sistemelor de calcul. Entităţile utilizate în cadrul programării orientate spre obiecte interacţionează cu sistemul prin intermediul unor interfeţe neambigue, facilitând modularitatea, încapsularea, interschimbabilitatea şi mentenabilitatea. Exemple pentru aplicarea în practică a acestui concept îl reprezintă bibliotecile consacrate de obiecte dezvoltate în ultimii ani, cum ar fi "Microsoft Foundation Classes" (MFC) sau "ObjectWindows Library" (OWL 2.0).

BUPT

Procesul de dezvoltare software urmărit în cadru prezentului capitol reprezintă secvenţa ce se referă la componenta software din cadrul schemei prezentate în figura 3.3. Etapele parcurse în cadrul acestui proces de dezvoltare software vor fi detaliate în cele ce urmează.

3.3.1 Analiza domeniului

Analiza domeniului reprezintă punctul de plecare logic în procesul de dezvoltare al oricărui sistem software. Scopul analizei domeniului este legat de necesitatea înţelegerii complete a sistemului ce urmează a fi dezvoltat. Sunt identificate elementele hardware, software, documentaţia şi experienţa ce pot fi reutilizate în procesul de dezvoltare a produsului. Deja în această etapă este abordată problema întocmirii unui plan de reducere a riscului şi a unei strategii de dezvoltare incrementală a produsului.

3.3.1.1 Înţelegerea domeniului problemei

Analiza domeniului precede analiza cerinţelor sistemului. Intrările primare pentru analiza domeniului sunt specificaţiile disponibile referitoare la sistem, precum şi cerinţele cunoscute şi anticipate ale beneficiarului. Documentul cuprinzând specificaţiile sistemului reprezintă sursa primară pentru identificarea de clase şi obiecte ale lumii reale.

Clasele şi obiectele identificate în această etapă constituie elemente pe baza cărora se poate evalua potenţialul de reutilizare atât a unor eventuale entităţi software existente cât şi a celora care urmează să fie dezvoltate ca şi componente noi în cadrul sistemului. Obiectele esenţiale pot fi utilizate pentru a defini subsisteme în cadrul sistemului global care poate să conţină şi soluţii distribuite.

Identificarea cerinţelor comune creează un set de capabilităţi funcţionale care poate constitui baza pentru crearea de clase şi obiecte. Elementele considerate în cadrul analizei sunt:

- *Trăsăturile sistemului.* Trăsăturile esenţiale ce trebuie determinate sunt operarea în timp real, distribuţia, orientarea spre baze de date, orientarea spre mesaje etc.
- *Cerinţele sistemului.* Se determină ariile funcţionale majore în cadrul domeniului problemei.
- *Arhitectura sistemului.* Include o determinare a dimensiunii sistemelor ce urmează a fi dezvoltate în contextul funcţionalităţii cerute. Se specifică arhitectura hardware preconizată şi sistemele de operare asociate.

BUPT

- *Clase şi obiecte ale lumii reale.* Acestea reprezintă abstractizări ale domeniilor funcţionale principale şi reprezintă un element auxiliar în perceperea complexităţii sistemului. Atributele trebuie evaluate din punctul de vedere al volatilităţii.

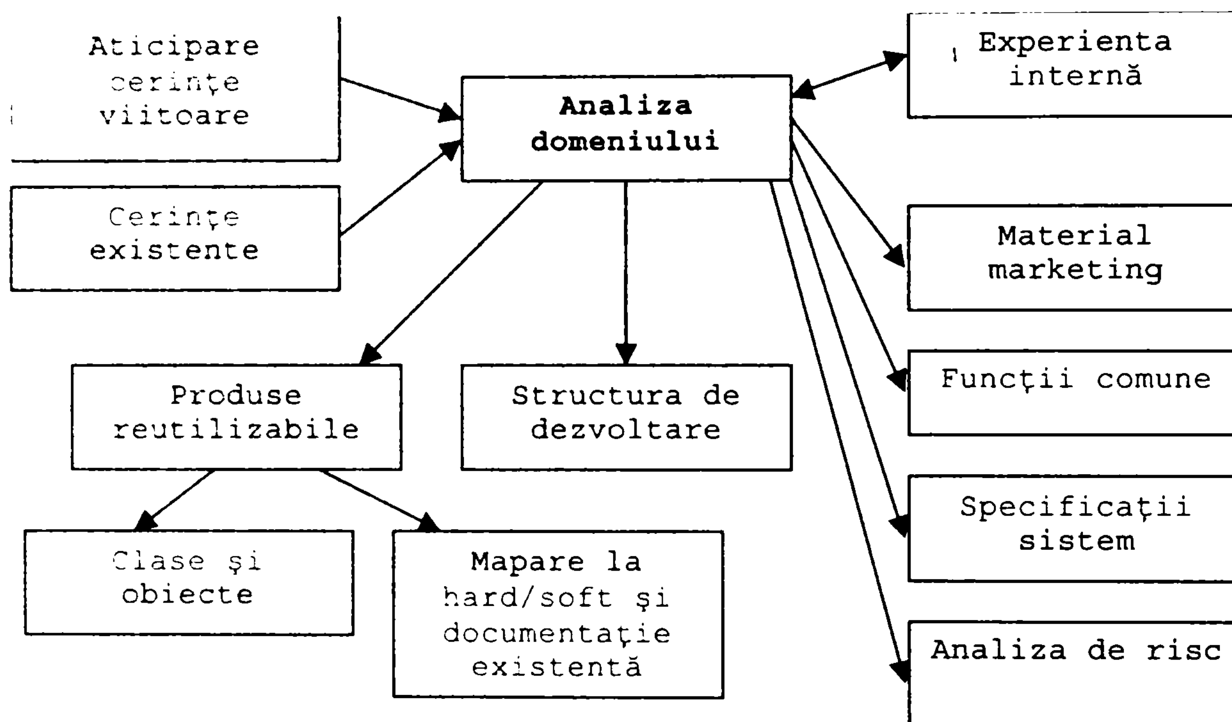


Figura 3.4 Relaţii în cadrul analizei domeniului

Figura de mai sus ilustrează structura şi relaţiile existente în cadrul analizei domeniului. Coroborarea cerinţelor existente, a anticipării cerinţelor viitoare şi a experienţei existente conduce la identificarea unui set de *funcţii comune*. Aceste funcţii vor fi descrise în documente ce vor forma specificaţia funcţională a sistemului. Funcţiile comune vor deveni ulterior abstractizările cheie din care sunt derivate clasele şi obiectele. Analiza domeniului reprezintă baza pentru planificarea cantităţii de software ce va trebui dezvoltat, precum şi funcţionalitatea software-ului potenţial reutilizabil.

3.3.1.2 Crearea claselor şi obiectelor lumii reale

Identificarea claselor şi a obiectelor lumii reale reprezintă rezultatul unui proces bazat pe clasificarea prezentată în cele ce urmează.

BUPT

3.3.1.2.1 Clasificare

Având în vedere diversitatea sistemelor ce pot face subiectul analizei orientate spre obiecte se impune definirea unei scheme generale de clasificare. Procesul de determinare iniţială a claselor şi obiectelor pe baza cerinţelor funcţionale are la bază existenţa următoarelor tipuri de clase [NIE95]:

- *Clase interfaţă dispozitiv*: modelează abstractizarea interfaţării componentei software cu dispozitivele hardware.
- *Clase sistem extern*: modelează interacţiunea unui sistem cu sisteme similare lui.
- *Clase interfaţă utilizator*: modelează interacţiunea între sistem şi utilizator.
- *Clase computaţionale*: intră în compoziţia sistemelor ce conţin o componentă computaţională semnificativă.
- *Clase rol*: sunt utilizate pentru a modela un rol cheie jucat de un personaj în cadrul sistemului
- *Clase abstractizare a datelor*: reprezintă colecţii de entităţi similare ce pot fi identificate în etapa de analiză utilizând tehnici de modelare a datelor ce se bazează pe diagrame de relaţii între entităţi.

3.3.1.2.2 Identificarea claselor şi a obiectelor

Clasele şi obiectele ce corespund tipurilor enumerate mai sus pot fi derivate direct din specificaţiile sistemului. Clasele abstractizare a datelor sunt deduse de obicei ca şi cerinţe derivate ce pot fi identificate la acest nivel pe baza diagramelor de relaţii între entităţi. Clasele şi obiectele sunt identificate de obicei în faza de prototipizare sau pe parcursul discuţiilor legate de detalii de implementare a sistemului.

3.3.1.3 Evaluarea obiectelor

Obiectele de nivel înalt create în faza de analiză a domeniului trebuie evaluate pentru a stabili în ce măsură sunt adecvate ca entităţi abstracte. Criteriile de evaluare trebuie să se refere la următoarele aspecte [NIE95]:

- *Entităţi ale lumii reale*: fiecare obiect identificat trebuie să aibă un corespondent în cadrul cerinţelor domeniului problemei. Obiectele trebuie să aibă denumiri ce pot fi asociate noţiunilor definite în cadrul cerinţelor domeniului.

BUPT

- *Completitudine*: setul de obiecte identificate trebuie să acopere complet setul de cerinţe ale domeniului problemei. Obiecte care nu pot fi identificate cu cerinţe enunţate în domeniul problemei pot să reprezinte cerinţe derivate şi vor fi incluse în analiză.
- *Dimensiunea*: granularitatea obiectelor abstracte trebuie să fie pronunţată. Obiecte de dimensiuni prea mici având un nivel înalt de detaliu vor fi tratate în faza de proiectare. Ele pot fi reunite în entităţi mai mari ce vor forma o abstractizare mai bună. Obiecte cu un nivel de complexitate prea mare vor fi tratate ca subsisteme reutilizabile sau vor fi descompuse în abstractizări mai mici.
- *Categoria de dezvoltare*: fiecare obiect trebuie identificat cu una din următoarele categorii de dezvoltare:
  - produs reutilizabil intern
  - produs comercial
  - produs ce necesită dezvoltare proprie
- *Clasificarea obiectelor*: fiecare obiect va fi încadrat în clasificarea bazată pe cerinţele funcţionale. Dacă vreunul din tipurile de clase din această clasificare nu conţine nici un obiect, atunci e posibil ca analiza să fie incompletă.
- *Atribute*: Un atribut a cărui descriere este prea generală va fi convertit într-un obiect individual care va reprezenta de obicei o cerinţă derivată.
- *Operaţii*: Fiecare din operaţiile asociate unei clase în actuala fază de dezvoltare trebuie să conţină o descriere unică ce identifică o singură acţiune asociată unui obiect.

3.3.1.4 Produse de lucru

Produsele de lucru generate în etapa de analiză a domeniului sunt:

- *Specificaţia funcţională a sistemului*: acest document conţine cerinţele sistemului în contextul domeniului problemei. El va servi ca precursor pentru un document formal de specificare a cerinţelor sistemului generat pe parcursul etapei de analiză a cerinţelor sistemului. Utilizările specificaţiei funcţionale a sistemului sunt legate de:
  - estimarea iniţială pentru durata de dezvoltare a produsului
  - activitatea de marketing a produsului
  - evaluarea dimensiunii şi a costurilor componentelor software.

Informaţii suplimentare vor fi adăugate acestui document pe măsura dezvoltării componentelor software.

BUPT

- *Clasele şi obiectele*: sunt generate abstractizări conceptuale de nivel înalt ale principalelor elemente din cadrul sistemului.
- *Fişe pentru clase şi obiecte*: aceste fişe sunt utilizate ca instrument în evaluarea eficacităţii claselor şi obiectelor
- *Software existent*: pentru fiecare set de clase şi de obiecte se va întocmi o listă a entităţilor software existente ce corespund funcţionalităţii dorite. Va fi evaluat potenţialul de reutilizare şi documentaţia existentă pentru respectivele entităţi software.
- *Experienţa internă*: identificarea experţilor în diversele arii ale domeniului problemei este importantă pentru stabilirea posibilităţii de implicare a acestora în dezvoltarea proiectului.
- *Planul de reducere a riscului*: sunt identificate zonele de risc ridicat (comunicaţiile interproces, toleranţa la erori, proiectarea bazelor de date etc.) pentru a fi avute în vedere în activitatea de generare de prototipuri. Interfeţele prototipurilor de funcţii trebuie definite astfel încât ele să poată fi integrate mai târziu în cadrul sistemului.
- *Planul preliminar de dezvoltare incrementală*: în procesul de dezvoltare a sistemelor mari funcţionalitatea este dezvoltată incremental. Fiecare pas de dezvoltare include un set de funcţionalităţi suplimentare. Acest plan constituie baza pentru analiza thread-urilor şi pentru determinarea scenariilor [JAC92] în etapa de analiză a cerinţelor sistem.

Produsele de lucru rezultate în etapa de analiză a domeniului reprezintă datele iniţiale pentru etapa de analiză a cerinţelor sistemului. În acest moment funcţionalitatea sistemului este cunoscută şi se poate face o evaluare rezonabilă a amplorii efortului de dezvoltare software, efort ce poate fi divizat în:

- sarcini legate de generarea de prototipuri
- sarcini legate de dezvoltarea sistemului.

3.3.1.5 Domenii de risc

Utilizarea tehnologiei orientate spre obiecte în procesul de analiză a domeniului nu implică nici un risc suplimentar faţă de abordarea structurată. Notaţiile specifice tehnologiei orientate spre obiecte sunt minime, reprezentările grafice nu sunt utilizate decât în cazul în care modelul obiectelor se utilizează pentru modelarea datelor iar utilizarea conceptelor orientării spre obiecte nu reclamă o experienţă deosebită. Conceptele analizei orientate spre obiecte sunt utilizate doar în faza de creare a obiectelor pe baza unui set de categorii de clase.

BUPT

3.3.2 Analiza cerinţelor sistemului

Metoda descrisă în cadrul capitolului actual pentru procesul de dezvoltare software conferă analizei cerinţelor sistemului rolul de *structurare a sistemului independent de mediul de implementare*. Această etapă va determina comportamentul şi limitările sistemului iar structura creată va trebui să fie suficient de robustă pentru a face faţă modificărilor cerinţelor. Robusteţea este realizată prin crearea unei structuri logice şi stabile bazată pe obiecte independente având interfeţe bine definite.

3.3.2.1 Utilizarea scenariilor

Abordarea etapei de analiză a cerinţelor sistemului propusă în cadrul actualului capitol se bazează pe conceptele *caz de uz* şi *scenariu* [JAC92] utilizate în cadrul metodei OOSE care a fost prezentată în cadrul capitolului anterior. Etapa actuală a analizei este iniţiată de identificarea scenariilor. În pasul următor sunt identificate obiectele care participă la fiecare scenariu. Utilizarea scenariilor facilitează crearea unei structuri care permite dezvoltare incrementală a sistemelor mari, reprezentând în acelaşi timp o bază pentru activitatea de generare a prototipurilor.

3.3.2.2 Definiţia scenariului

Un scenariu reprezintă o secvenţă de acţiuni care au loc în cadrul sistemului. Desfăşurarea unui scenariu este declanşată de un *stimul* generat de către o sursă externă iar finalul scenariului este reprezentat de terminarea logică a secvenţei de acţiuni aferente acestuia.

Stimulii externi pentru un scenariu sunt generaţi de *actori* (clase) respectiv *utilizatori* (instanţe specifice ale unui actor). *Utilizatorul* reprezintă persoana sau mesajul care stimulează sistemul. De fiecare dată când un *utilizator* stimulează sistemul, în cadrul acestuia are loc o secvenţă de acţiuni reprezentând un scenariu. Diferenţă majoră între *utilizator* şi alte obiecte în cadrul sistemului este faptul că *utilizatorul* reprezintă un sistem nedeterminist. Obiectele dezvoltate pentru a participa la scenariu sunt deterministe, deci se cunoaşte starea şi comportamentul acestora. Un scenariu poate fi considerat o instanţă a unei clase de scenarii. Fiecare scenariu va determina o secvenţă cunoscută de schimbări de stare a sistemului.

BUPT

3.3.2.3 Proprietățile scenariilor

Studii referitoare la conceptul de scenariu au mai fost efectuate de Harel [HAR87] şi Deutsch şi Nielsen [DEU88, DEU91]. Scenariul reprezintă un descriptor al comportamentului sistemului în termeni ai aplicației. Scenariile pot de asemenea să joace un rol în evidențierea incertitudinilor, erorilor şi omisiunilor în comportamentul sistemelor. Un scenariu poate fi definit ca şi o entitate care:

- conectează un stimul extern cu un răspuns intern pentru a descrie o cale comportamentală perceptibilă de către utilizator
- conține evenimente de date, evenimente de control şi condiții
- reprezintă atât o specificație cât şi o implementare
- poate fi implementat atât în hardware, cât şi în software, prin acțiuni manuale sau o combinație a acestora.

Figura următoare ilustrează structura unui scenariu.

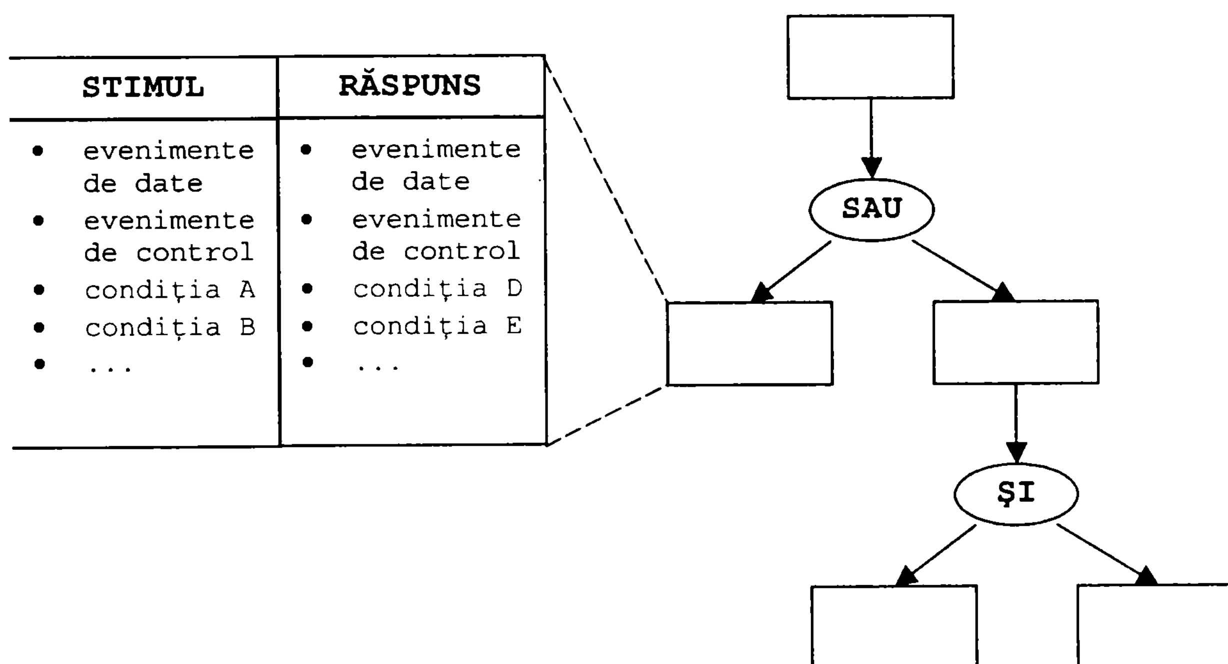


Figura 3.5 Structura unui scenariu

După cum se poate observa, atât stimulul cât şi răspunsul pot conține evenimente de date, evenimente de control şi condiții:

- *Evenimentele de date* reprezintă simple intrări şi ieşiri.
- *Un eveniment de control* poate fi reprezentat fie de un semnal, fie de un inițiator sau terminator de proces.

BUPT

- *O condiție* reprezintă o "fotografie" a stării tranzitorii a fiecărei entități externe şi eventual interne care joacă un rol în cadrul scenariului.

În cadrul unui scenariu sunt reprezentate în paralel condiții ale mai multor obiecte relevante. Acestea descriu evoluția sistemului din momentul aplicării stimulului până în momentul generării răspunsului.

Structura de concurență în cadrul unui sistem orientat spre evenimente este determinată de relația dintre răspunsul scenariului curent şi stimulii scenariilor succesoare. Este de preferat o relație unu la unu între un scenariu şi un element structural reprezentând o colecție de unul sau mai multe obiecte abstracte. Acestea pot fi transpuse mai târziu în obiecte de proiectare şi implementate sub formă de clase C++ sau de entități ale altor limbaje orientate spre obiecte.

Se urmăreşte obținerea unei corespondențe directe între arhitectura sistemului şi scenariile operaționale de utilizare a acestuia.

3.3.2.4 Identificarea scenariilor

Pentru determinarea scenariilor se stabilesc mai întâi *interfețele* sistemului cu utilizatorul sau cu alte sisteme externe. Se trasează activitățile declanşate în cadrul sistemului de către fiecare stimul şi se înregistrează secvența de evenimente rezultate.

Activitatea de identificare a scenariilor poate fi facilitată de răspunsurile la următoarele întrebări legate de actorii care interacționează cu sistemul [JAC92, pag 155]:

- Care sunt sarcinile principale ale fiecărui actor?
- Actorul va trebui să citească / scrie / modifice informația din cadrul sistemului?
- Actorul va trebui să informeze sistemul cu privire la schimbări externe?
- Actorul doreşte să fie informat cu privire la modificări neaşteptate?

Una din posibilitățile de identificare a actorilor în cadrul unui sistem se bazează pe diagrama de context a sistemului. Acest proces a fost exemplificat în cadrul unui sistem de control al traficului aerian (ATCS) [NIE95], a cărui diagramă de context a sistemului este prezentată în figura următoare.

BUPT

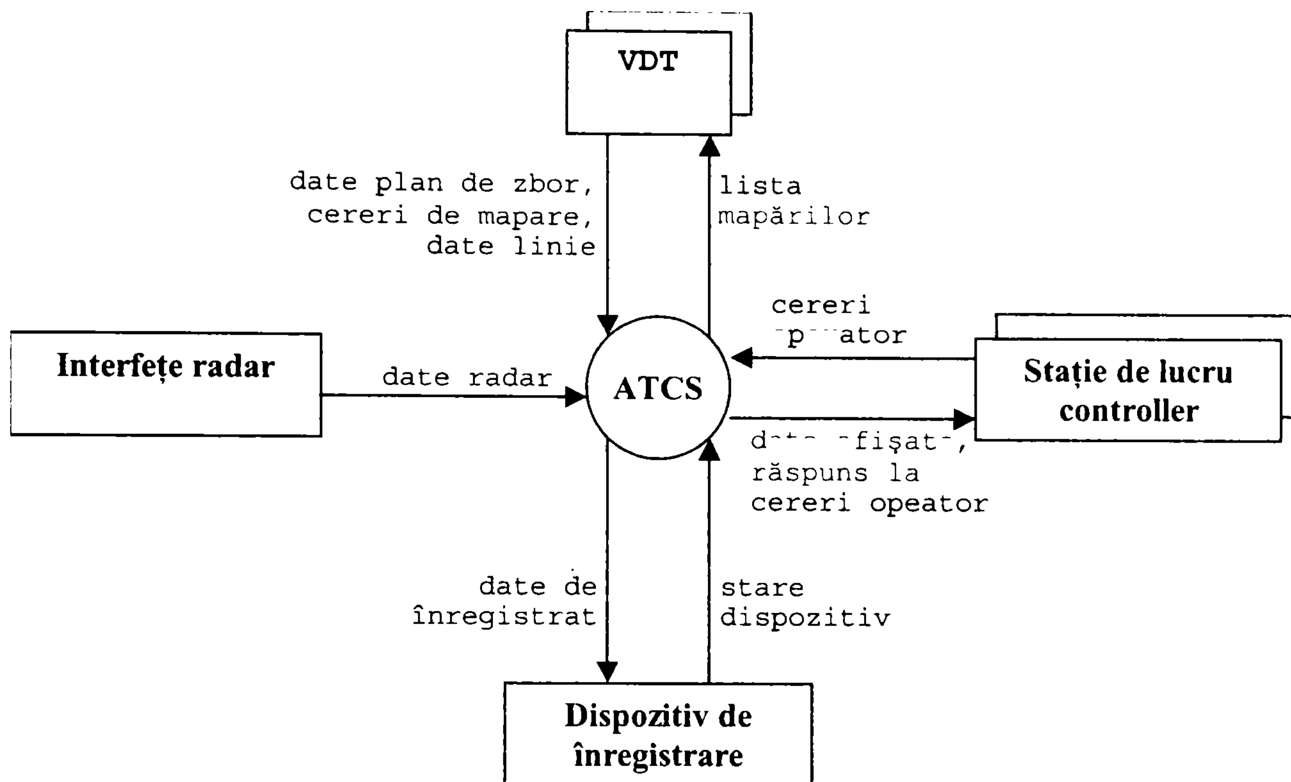


Figura 3.6 Diagrama de context a unui sistem de control al traficului aerian

Pentru identificarea scenariilor se poate porni cronologic, considerând un singur obiect fizic din mediul extern. De exemplu datele planului de zbor (FP) pot fi introduse de către un actor prin intermediul unui terminal clasic (VDT). *Caracteristica stimul – răspuns* pentru acest scenariu este prezentată în tabelul următor:

Identificator scenariu: 1 **Titlu:** Introducere normală date zbor	
Stimul	**Răspuns**
Operatorul primeşte cererea de plan de zbor	Se generează un nou plan de zbor
Operatorul introduce datele planului de zbor	Mesajul "acceptat" este returnat operatorului
Sistemul în stare de capacitate normală	

Alte scenarii sunt create luând în considerare alţi stimuli provenind de la fiecare actor din cadrul mediului extern.

Fiecărui scenariu îi este atribuit un nume şi un identificator unic. Fiecărei faze din planul de dezvoltare incrementală îi este asociată o colecţie de scenarii, faza de dezvoltare fiind reţinută fie în cadrul numelui, fie în cadrul identificatorului de scenariu. Unele scenarii evoluează ca

BUPT

urmare a modificării condiţiilor, fiind posibilă analogia cu automatele finite. Fiecărui scenariu îi pot fi asociate informaţii referitoare la cerinţele de performanţă temporală.

3.3.2.5 Documentarea scenariilor

Formele de documentare a scenariilor depind de nivelul de abstractizare. La cel mai înalt nivel de abstractizare se află *diagrama scenariilor*, reprezentând setul de scenarii împreună cu actorii care reprezintă stimulii pentru aceste scenarii. Un exemplu în acest sens este prezentat în [JAC92, pag. 155-156] pentru o maşină de reciclare. Actorii sunt reprezentaţi de clienţii care predau obiectele de reciclat şi un operator care întreţine maşina. Scenariile sunt reprezentate sub formă de elipse adnotate cu denumirea scenariului.

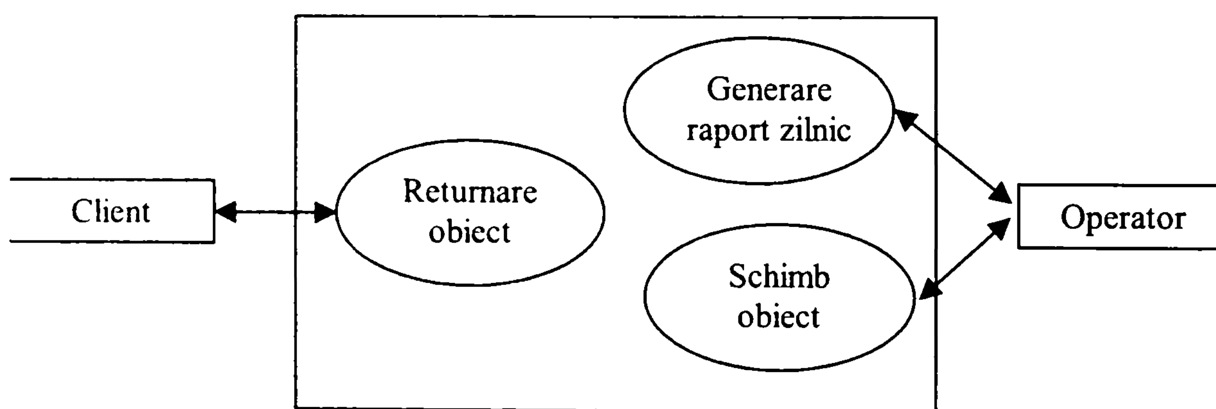


Figura 3.7 Diagrama scenariilor pentru sistemul "maşină de reciclare"

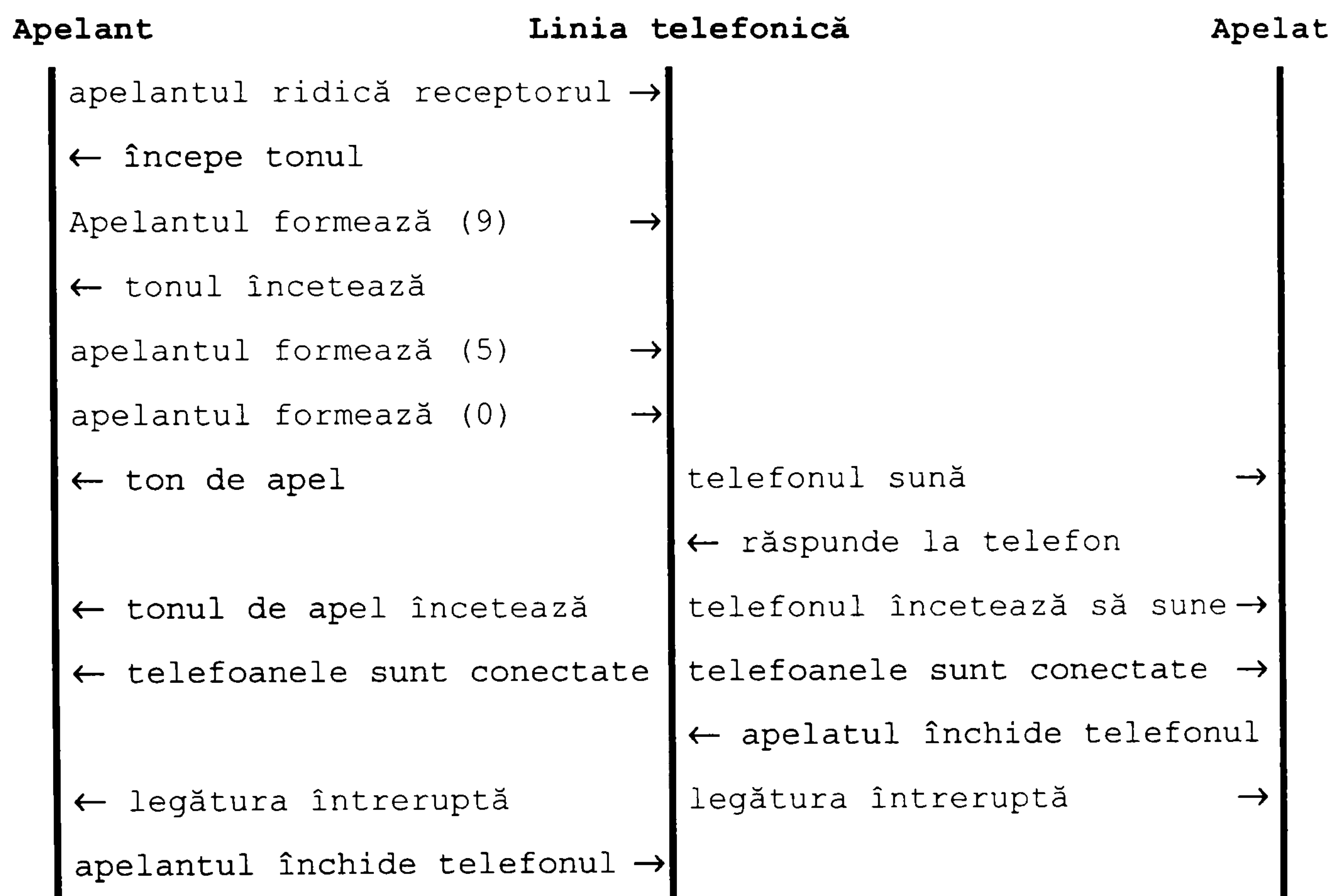
O altă abordare în descrierea unui scenariu constă în *diagrama de trasare a evenimentelor* în cadrul cărora se descrie interacţiunea între obiectele componente ale sistemului. Această abordare este exemplificată în [RUM91] în contextul utilizării unui telefon pentru a forma un număr. Descrierea narativă a scenariului completează de cele mai multe ori diagrama de trasare a evenimentelor:

```
Apelantul ridică receptorul
Începe tonul pentru selecţia numărului
Apelantul formează (9)
Tonul încetează
Apelantul formează (5)
Apelantul formează (0)
Telefonul apelat începe să sune
Tonul de apel se aude la telefonul apelant
Apelatul răspunde
Telefonul apelat încetează să sune
Tonul de apel dispare la telefonul apelant
```

BUPT

```
Telefoanele sunt conectate
Apelatul închide telefonul
Telefoanele sunt deconectate
Apelantul închide telefonul
```

Diagrama de trasare a evenimentelor pentru scenariul apel telefonic ilustrată în continuare descrie interacțiunea între obiectele participante. Fiecărui obiect îi este rezervată o coloană, săgețile orizontale reprezintă mesajele transmise între obiecte. Ordinea cronologică a acestor mesaje este dată de parcurgerea de sus în jos a diagramei. Această reprezentare nu conține informații cantitative legate de durata fiecărui eveniment.



Documentarea scenariilor poate fi făcută sub diferite forme, în funcție de nivelul de abstractizare dorit și de tipul informației care se dorește a fi transmis.

3.3.2.6 Proiectarea bazată pe scenarii

Modelul sistemului se bazează pe setul de scenarii determinate. Acestea constituie baza planului de dezvoltare incrementală și a procesului de generare a prototipurilor. Avantajele acestei abordări sunt următoarele:

BUPT

- *Impactul modificărilor* ce pot surveni în cadrul sistemului *se va limita la remodelarea actorilor şi a scenariilor* afectate de către acestea, fără a necesita reproiectarea întregului sistem.
- *Revizuirea proiectării* poate fi efectuată în mai multe etape la nivelul fiecărui scenariu, înlocuind o revizuire globală la nivelul întregului sistem care poate să fie mai puţin eficientă.
- *Setul de scenarii* odată dezvoltat, poate să devină un *instrument valoros pentru demonstrarea conceptelor* şi capabilităţilor operaţionale ale sistemului.

Scenariile sunt utilizate în principal la proiectarea sistemelor ce presupun interacţiune cu lumea exterioară. O abordare bazată pe scenarii nu este utilă în cazul proiectării unei biblioteci de clase ce vor fi utilizate în cadrul altor programe şi care nu conţin interfeţe cu sisteme externe. Această observaţie va fi aplicată în procesul de dezvoltare a unei biblioteci de ferestre pentru reprezentări geometrice 2D şi 3D, prezentată în cadru capitolului următor.

3.3.2.7 Analiza orientată spre obiecte

Obiectele generate în faza de analiză a domeniului au fost derivate strict din setul de cerinţe ale sistemului, fără a avea în vedere o structură logică. Aceste obiecte vor fi rafinate în faza de analiză a cerinţelor sistemului şi vor forma baza modelului de analiză.

Una din modalităţile de reprezentare a informaţiei în cadrul modelului de analiză este legată de considerarea unui *spaţiu informaţional tridimensional* [NIE95] constând din axele de prezentare, a informaţiei şi a comportamentului:

- *Axa de prezentare* conţine informaţie asociată interfeţelor externe al sistemului
- *Axa informaţiei* descrie stările interne ale sistemului
- *Axa comportamentului* este asociată modului de răspuns a sistemului la schimbările de stare.

Modelul de analiză este determinat prin identificarea obiectelor ce sunt aliniate la aceste trei axe ale spaţiului informaţional tridimensional.

Pentru clasificarea obiectelor se va adopta o schemă de propusă de Jacobson [JAC92]. Obiectele vor fi împărţite în trei categorii:

BUPT

- *obiectele entitate*: reprezintă cerințele sistemului reflectate în obiecte ale lumii reale şi sunt asociate axei informației.
- *obiectele de interfață*: modelează informația reprezentând interfețele cu sistemele externe şi sunt aliniate axei de prezentare.
- *obiectele de control*: sunt responsabile cu tratarea schimbărilor stării sistemului şi sunt aliniate axei comportamentului.

Entitățile de nivel înalt determinate pe parcursul analizei domeniului vor fi încadrate în aceste trei categorii. Eventualele modificări ale cerințelor sistemului pot fi astfel localizate mai uşor la nivelul obiectelor individuale. Pentru sistemele mari, obiectele din cele trei categorii vor fi grupate în subsisteme.

Puține din metodele consacrate de analiză orientată spre obiecte suportă utilizarea scenariilor ca elemente esențiale pentru crearea modelului de analiză. Astfel, metode cum ar fi OMT, Shlaer-Mellor, sau Booch'93 îşi regăsesc utilitatea în actuala fază doar cu scopuri de documentare. Un utilitar capabil să realizeze structurarea obiectelor pornind de la scenarii este *Objectory* [JAC92] care implementează metoda OOSE.

3.3.2.8 Modelul de analiză

Scopul modelului de analiză este de a crea baza pentru etapa de proiectare a sistemului. Modelul de analiză se bazează pe modelul scenariilor. Fiecare scenariu conține un număr de obiecte de analiză ce corespund celor trei categorii prezentate mai sus. Anumite obiecte vor participa la mai multe scenarii, acest fapt fiind determinant pentru ordinea în care sunt dezvoltate obiectele.

Tipurile de scenarii considerate pentru generarea modelului de analiză sunt prezentate în [JAC92]:

1. *Scenariile dependente în mod direct de mediul sistemului* vor implica *obiectele interfață*. Unele dintre aceste obiecte vor rezulta direct din categoriile de obiecte identificate în cadrul analizei domeniului:
   - clase interfață dispozitiv
   - clase sistem extern
   - clase interfață utilizator
   - clase rol

BUPT

2. *Scenariile responsabile cu stocarea şi manipularea datelor* în mod independent de mediul sistemului sunt implementate prin intermediul *obiectelor entitate*. Unele dintre aceste obiecte pot fi derivate din clasele computaţionale şi de abstractizare a datelor de la nivelul analizei domeniului.

Funcţionalităţile sistemului care nu pot fi încadrate în cele două tipuri de scenarii prezentate mai sus vor fi implementate pe baza *obiectelor de control*. Obiectele de control nu sunt indispensabile pentru dezvoltarea unui sistem, ele pot să nu facă parte deloc din structura sistemelor de complexitate redusă. Se poate crea o întreagă structură de analiză care să conţină doar obiecte de interfaţă şi obiecte entitate [NIE95].

Procesul de creare a obiectelor va începe cu obiectele entitate şi cele de interfaţă, fără a putea stabili o cronologie strictă între aceste două tipuri, procesul fiind mai degrabă iterativ. Crearea obiectelor de control va avea loc după ce obiectele din primele două categorii au fost identificate.

3.3.2.8.1 Obiecte interfaţă

Obiectele interfaţă sunt asociate evenimentelor unui scenariu ce implică interacţiunea între sistem şi mediului său exterior. Actorul care declanşează un scenariu comunică cu sistemul prin intermediul obiectelor de interfaţă. Rolul obiectelor de interfaţă este de a traduce stimulii generaţi de către actor în evenimente sistem şi de a transmite actorului răspunsurile sistemului.

Metode de identificare a obiectelor de interfaţă sunt prezentate în [JAC92] şi [NIE95]:

1. Se consideră un *actor* şi se determină *funcţia de interfaţă* între stimulul generat de acesta şi o funcţie sistem.
2. Dacă sunt disponibile *obiecte de analiză a domeniului*, atunci aceste obiecte din categoriile interfaţă dispozitiv, sistem extern, interfaţă utilizator şi rol vor fi transpuse direct în obiecte de interfaţă.
3. Obiectele de interfaţă pot fi identificate plecând de la o *descriere textuală a scenariului* provenind direct de la modelul cerinţelor.

Obiectele interfaţă pot fi la rândul lor compuse din alte obiecte de interfaţă, după cum rezultă din următoarea ilustraţie. Agregarea prezentată utilizează notaţia OMT, la care s-a adăugat

BUPT

notația "I" încadrată în colțul din dreapta jos al fiecărui obiect interfață dat fiind faptul că metoda OMT nu suportă acest tip de clasificare a obiectelor.

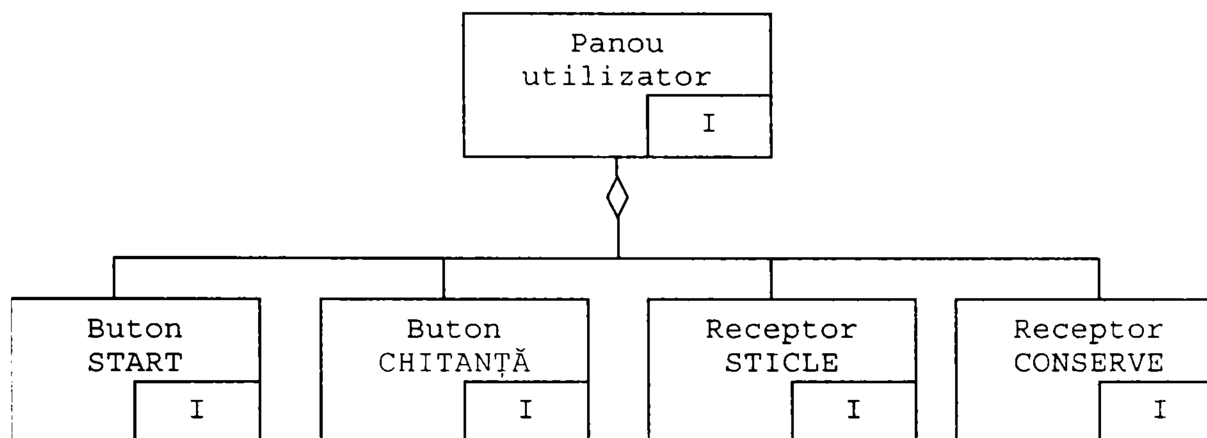


Figura 3.8 Agregarea obiectelor de interfață în cadrul panoului utilizator al mașinii de reciclare

Un actor poate fi reprezentat și de un sistem extern. Atunci când obiectele de interfață comunică cu alte sisteme externe, interfața poate fi descrisă printr-un protocol de comunicație. Protocolul poate reprezenta un standard sau poate fi un set de mesaje specifice aplicației. Dacă protocolul se modifică, modificările sunt izolate la nivelul obiectelor interfață care suportă funcțiile protocolului.

3.3.2.8.2 Obiecte entitate

Obiectele entitate sunt utilizate pentru modelarea informației ce va persista și după ce scenariul și-a parcurs întreaga secvență de evenimente. Pentru identificarea obiectelor entitate se poate aplica una din următoarele metode [NIE95]:

1. Obiectele entitate sunt *identificate* pe baza *descrierii scenariului*
2. Obiectele entitate pot fi *deduse* din obiectele analizei domeniului *din categoriile computațională și de abstractizare a datelor*.

Figura următoare ilustrează obiectele entitate în cadrul scenariului de returnare entitate din cadrul mașinii de reciclare. Este utilizată notația din cadrul metodei OMT la care s-a adăugat simbolul "E" în cadrul fiecărui obiect entitate.

BUPT

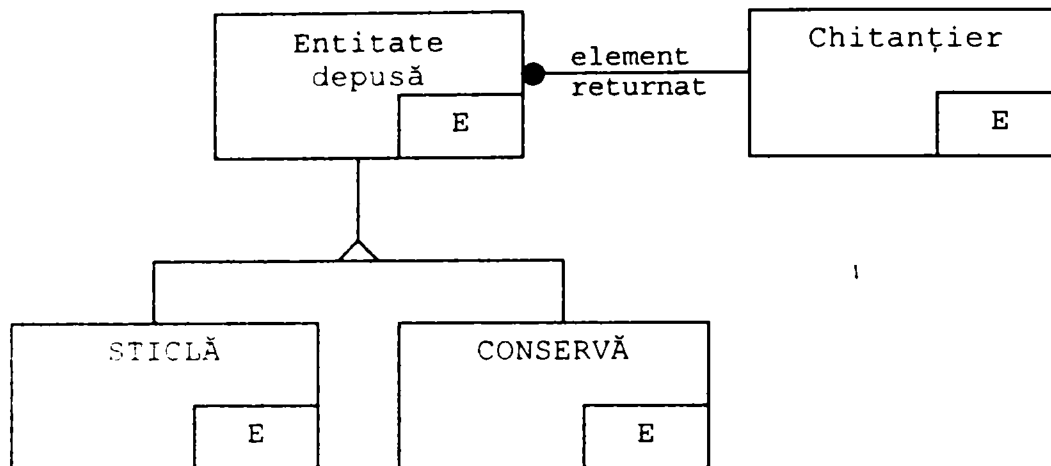


Figura 3.9 Obiectele entitate în cadrul scenariului returnare entitate

Este posibil ca anumite obiecte entitate să facă parte din mai multe scenarii. Devine astfel importantă analiza similarității obiectelor entitate din totalitatea scenariilor considerate pentru a identifica situațiile de reutilizabilitate directă.

3.3.2.8.3 Obiecte de control

Obiectele de control rezultă din alocarea funcționalităților rămase neacoperite după identificarea obiectelor interfață și a obiectelor entitate. Scenariile pentru care totalitatea funcționalităților au fost alocate obiectelor interfață și obiectelor entitate nu conțin obiecte de control. Informația asociată obiectelor de control este de scurtă durată și persistă de regulă într-un interval de timp relativ scurt în cadrul unui scenariu. Funcții tipice plasate în cadrul obiectelor de control sunt:

- comportament legat de tranzacții
- secvențe de control
- funcții destinate izolării obiectelor entitate de obiectele de interfață.

3.3.2.8.4 Corelarea obiectelor de analiză

După identificarea tuturor obiectelor din cadrul unui scenariu urmează analiza relațiilor între aceste obiecte. Această analiză este ilustrată pe baza figurii următoare ce reprezintă diagrama de flux a evenimentelor pentru aplicația „mașină de reciclare" conform notației metodei OMT. Categoria fiecărui obiect este consemnată în colțul inferior drept (C = control, E = entitate, I = interfață).

BUPT

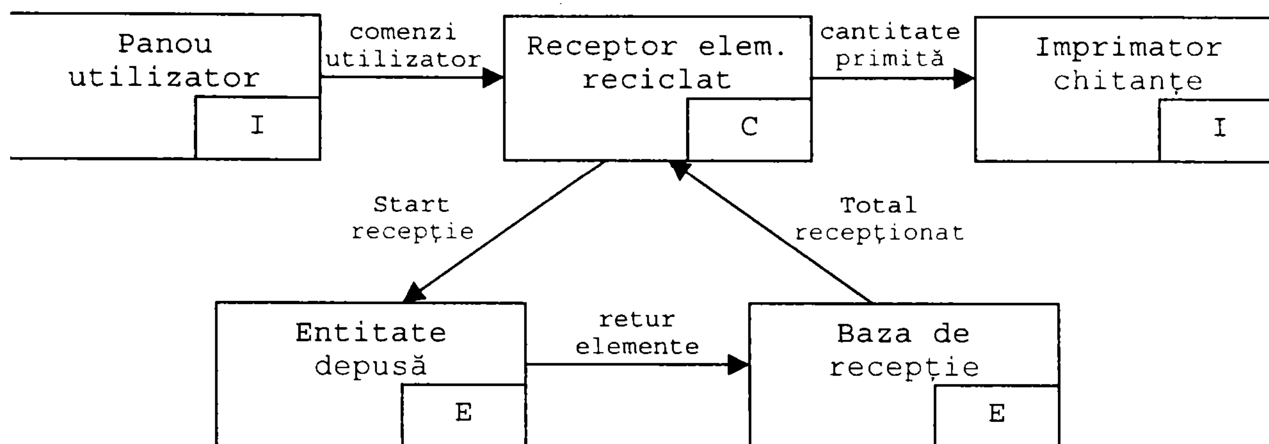


Figura 3.10 Diagrama de flux a evenimentelor pentru scenariul returnare entitate

Receptorul de elemente reciclate controlează intrările de la panoul utilizator şi transmite un mesaj obiectului entitate depusă pentru a semnala incidenţa elementelor returnate. În momentul când toate elementele au fost returnate receptorul de elemente reciclate recepţionează totalul de la baza de recepţie şi transmite un mesaj imprimatorului de chitanţe pentru tipărirea unei chitanţe.

3.3.2.8.5 Criterii de încheiere a fazei de analiză

Una dintre problemele importante în ceea ce priveşte dezvoltarea unui sistem utilizând tehnologia orientată spre obiecte este legată de modul în care se face delimitarea între etapa de analiză şi etapa de proiectare. O analiză prea detaliată duce direct în faza de proiectare, afectând negativ formarea structurii logice care se doreşte a fi obţinută. Lipsa unei demarcaţii clare între faza de analiză şi cea de proiectare poate duce la situaţii în care modificarea cerinţelor are ca efect reproiectarea unei porţiuni semnificative a sistemului. Structura bazată pe scenarii la care participă obiecte entitate, interfaţă şi control are ca finalitate *localizarea oricăror modificări ale cerinţelor la nivelul scenariilor şi a obiectelor asociate, fără a afecta structura aplicaţiei.*

Un criteriu de completitudine a etapei de analiză este definit de:

- identificarea tuturor scenariilor
- asocierea de obiecte fiecărui scenariu
- identificarea acelor obiecte care suportă mai mult de un scenariu pentru analiza funcţionalităţii comune.

Etapa de analiză nu abordează decât în măsură redusă atributele şi operaţiile asociate obiectelor, cu excepţia acelor obiecte ce pot fi asociate mai multor scenarii. Aceste elemente vor fi

BUPT

abordate în etapa de proiectare, îmbunătăţind astfel delimitarea între faza de analiză şi cea de proiectare.

Numărul scenariilor din cadrul sistemelor mari este ridicat, la fiecare scenariu participând un număr semnificativ de obiecte. Pentru a putea gestiona un asemenea număr de obiecte, acestea vor fi grupate în subsisteme. Partiţionarea sistemelor mari în subsisteme reprezintă obiectul etapei de proiectare a sistemului. O primă identificare a subsistemelor poate avea loc în cadrul analizei domeniului, atunci când sunt identificate entităţile reutilizabile.

3.3.2.9 Euristica creării obiectelor de analiză

Metodele de creare a obiectelor de analiză descrise până în acest moment au la bază utilizarea scenariilor. Abordarea propusă pentru procesul de dezvoltare software are în vedere şi proiectele în cazul cărora analiza bazată pe scenarii nu reprezintă o soluţie adecvată. Din această categorie fac parte bibliotecile de obiecte. Generarea obiectelor de analiză pentru acest tip de aplicaţii [NIE95] se bazează pe:

- *Utilizarea substantivelor.* Una din primele abordări ale dezvoltării software orientate spre obiecte [ABB83] propune sublinierea tuturor substantivelor din cadrul specificaţiei cerinţelor ca bază pentru determinarea obiectelor. Această metodă poate genera multe *obiecte irelevante* în cazul sistemelor mari. O metodă de triere a acestor obiecte este sugerată în [RUM91] şi [COA91], constând în *eliminarea acelor obiecte ce reprezintă doar atribute sau se situează în afara sistemului* ce se dezvoltă.
- *Entităţile din cadrul diagramelor de relaţii între entităţi (ERD).* Dacă sunt disponibile diagrame de relaţii între entităţi, entităţile descrise în aceste diagrame pot fi transformate direct în obiecte de analiză. Relaţiile între aceste entităţi pot fi transformate în asocieri între obiecte.
- *Diagramele de flux de date.* Elementele de memorare a datelor din cadrul unei diagrame de flux de date pot deveni obiecte de analiză. Transformările asociate unui anumit element de memorare a datelor pot deveni operaţiile respectivului obiect.
- *Rafinarea obiectelor de analiză a domeniului.* Obiectele lumii reale determinate în etapa de analiză a domeniului au granularitate mare şi reprezintă abstractizări de nivel înalt ce pot fi rafinate. Unele din obiectele analizei domeniului vor fi transpuse în obiecte entitate, altele pot deveni obiecte de interfaţă sau de control. Un singur obiect de analiză a domeniului poate fi descompus în mai multe obiecte corespunzătoare etapei de analiză a cerinţelor sistemului.

BUPT

- *Roluri*. Anumite roluri jucate de utilizatori ai sistemului pot fi utilizate în procesul de identificare a claselor şi obiectelor. Astfel, rolul controlorului în cadrul sistemului de control al traficului aerian poate fi văzut ca şi o clasă, având ca instanţe specifice supervizorul, controlorul de sector, etc.

Criteriile enunţate mai sus pentru crearea obiectelor de analiză se regăsesc în cadrul mai multor metode consacrate de dezvoltare software orientată spre obiecte. Specificaţiile metodei OMT [ACC93] sugerează următorii paşi pentru procesul de identificare a obiectelor de analiză:

1. Se subliniază toate substantivele în cadrul enunţului problemei utilizând următoarele criterii:
    - se începe cu substantivele din aria problemei; singurul criteriu pentru selecţie este relevanţa pentru problemă
    - se includ atât entităţi fizice cât şi concepte; nu se încearcă organizarea utilizând agregarea sau generalizarea, nu se face distincţia între obiecte şi atribute
2. Se generează o diagramă de context pentru a facilita vizualizarea frontierelor sistemului
    - se selectează un nume descriptiv pentru sistem
    - se caută informaţie existentă referitoare la interfeţe cu entităţi externe sistemului.
    - se identifică datele transferate între sistem şi fiecare dintre interfeţele sale externe
3. Modificarea listei iniţiale de obiecte
    - dacă lista iniţială conţine obiecte externe sistemului acestea trebuie excluse din analiza ce urmează
    - dacă din lista iniţială lipsesc obiecte importante de interfaţă acestea vor fi adăugate la aceasta
    - diagrama de context se utilizează pentru a ghida procesul de selecţie
4. Aplicarea cunoştinţelor lumii reale
    - identificarea obiectelor adiţionale
    - adăugarea cerinţelor adiţionale la specificaţia cerinţelor
5. Selecţia numelor
    - numele vor avea valoare descriptivă şi vor fi neambigue
    - se va stabili un ghid pentru stilul de atribuire a numelor

BUPT

3.3.2.10 Selecția claselor

Selecția claselor are loc după identificarea obiectelor de analiză. Următoarea desfăşurare a procesului de selecție a claselor este specifică metodei OMT [ACC93a]:

1. Obiectele de analiză se atribuie unui set inițial de clase.
2. Clasele redundante sunt eliminate:
   - se aplică generalizarea / specializarea
   - clase de dimensiuni foarte mici vor fi grupate în clase mai mari
   - două clase care descriu aceeaşi informație sunt redundante
   - se va păstra clasa cu numele cel mai sugestiv
3. Clasele irelevante sunt eliminate:
   - se vor elimina clasele care nu sunt în legătură cu domeniul problemei
   - relevanța unei clase depinde de context
4. Clasele care reprezintă atribute sunt eliminate:
   - numele care descriu proprietăți ale obiectelor reprezintă mai degrabă atribute decât obiecte
5. Clasele care reprezintă operații sunt eliminate:
   - dacă un nume descrie o operație aplicată unei clase, atunci acesta nu reprezintă o clasă în sine
6. Structurile de implementare sunt eliminate:
   - entitățile care nu aparțin lumii reale vor fi eliminate cu excepția cazului în care reprezintă cerințe derivate necesare, independente de implementare (structurile de date de tipul "tablourilor" sau a "cozilor" reprezintă aproape întotdeauna structuri de implementare)
7. Clarificarea claselor vag definite:
   - se vor modifica denumirile inexacte de clase
   - se vor identifica clasele cu vizibilitate largă (vagă)
   - se vor identifica clasele care au limite vag definite

3.3.2.11 Evaluarea obiectelor de analiză

Obiectele determinate în etapa de analiză a cerințelor sistemului au asociate mai multe detalii decât cele identificate în etapa de analiză a domeniului. Trebuie luate în considerare obiecte adiționale, dat fiind faptul că deja există o clasificare în obiecte de interfață, entitate şi de control.

BUPT

O schemă de evaluare a obiectelor de analiză şi clasele asociate lor se bazează pe fişa de obiect de analiză prezentată în continuare [NIE95]:

	Element revizuit		Comentarii
1.	Categoria obiectului		
2.	Completitudine		
3.	Dimensiune		
4.	Atribute		
5.	Operații		
6.	Identificator scenariu		
7.	Nivel de detaliu		
8.	Volatilitate		
9.	Dependența de implementare		
	- hardware		
	- software		
	- DBMS		
10.	Subsistem candidat		
11.	Agregare		
12.	Asociere		

3.3.2.12 Domenii de risc

Abordarea orientată spre obiecte a etapei de analiză a cerințelor sistemului presupune deja un nivel adecvat de cunoştinţe şi o experienţă corespunzătoare în ceea ce priveşte tehnologia obiectuală. Această abordare implică deci un factor de risc pentru succesul procesului de dezvoltare software, risc ce se amplifică odată cu dimensiunea proiectului. Acest factor de risc trebuie avut în vedere şi evaluat înaintea începerii dezvoltării unui proiect.

Una dintre deciziile importante în această etapă este legată de metoda OOA ce va fi utilizată în etapa de analiză. Selecția metodei trebuie să se bazeze pe o evaluare imparţială a

BUPT

capabilităţilor metodelor candidate. Structurarea sistemelor mari este abordată doar superficial în cadrul metodelor de dezvoltare software prezentate în capitolul anterior. Soluţia oferită este de grupare a obiectelor în subsisteme. În cadrul acestor metode nu se fac referiri la alocarea cerinţelor hardware şi software, toleranţă la erori sau soluţii distribuite. Este foarte important ca abordarea acestor teme să constituie o activitate distinctă în cadrul proiectării.

Diversele tehnici OOA utilizează notaţii diferite pentru a descrie aceleaşi concepte, aceste notaţii fiind în ansamblul lor diferite de notaţiile utilizate în cadrul metodelor structurate. Există notaţii care pot fi implementate utilizând instrumente grafice uzuale, pe când alte notaţii se ridică la un nivel de complexitate atât de ridicat încât utilizarea lor presupune existenţa unui instrument CASE, cum este cazul notaţiilor folosite în cadrul metodelor OOA Booch, Shlaer-Mellor, Coad-Yourdon.

Abordarea bazată pe scenarii prezentată mai sus utilizează notaţii dintre cele mai familiare şi uşor de gestionat:

- *utilizarea simbolurilor speciale* pentru cele trei tipuri de obiecte (interfaţă, entitate şi control) poate fi simplificată recurgând la notaţia OMT îmbunătăţită, după cum s-a exemplificat anterior
- *diagrama de context* poate să constituie un important element auxiliar în identificarea actorilor sistemului. Notaţia este întâlnită şi în analiza structurată şi nu reprezintă un element de risc.
- *dicţionarul de date* reprezintă un alt element prezent şi în cadrul analizei structurate, fiind în măsură să diminueze factorul risc.

Un alt posibil factor de risc este reprezentat de *insuficienta delimitare între etapa de analiză şi etapa de proiectare.* Există pericolul de a se pierde informaţia din categoria "ce" , specifică etapei de analiză în favoarea informaţiei din categoria "cum" specifică etapei de proiectare, generând pentru aplicaţie o structură dependentă de implementare. Abordarea orientată spre scenarii prezentată mai sus se caracterizează printr-o bună separare a etapei de analiză de cea de proiectare datorită criteriilor de încheiere a etapei de analiză enunţate în cadrul paragrafului 10.1.2.8.5.

3.3.2.13 Produse de lucru

Cele mai importante produse de lucru rezultate în etapa de analiză a cerinţelor sistemului sunt următoarele:

BUPT

- *Diagrama de context*: este utilizată pentru identificarea actorilor.
- *Diagrama de scenarii*: ilustrează actorii şi scenariile.
- *Descrierile scenariilor*: pot fi textuale sau sub formă de secvenţă de evenimente. Relaţiile dintre obiecte şi evenimente sunt redate în cadrul diagramei de trasare a evenimentelor. Răspunsul la un stimul iniţiat de un actor este documentat în cadrul diagramei stimul-răspuns.
- *Diagrama de secvenţă a scenariilor*: o versiune dată a aplicaţiei este compusă dintr-un set de scenarii
- *Diagrama obiectelor*: descrie relaţia dintre actori şi obiectele care suportă un anumit scenariu.
- *Obiectele de analiză şi relaţiile lor*: sunt descrise diversele obiecte de analiză care suportă un anumit scenariu şi relaţiile dintre acestea.
- *Planul revizuit de dezvoltare incrementală şi prototipizare*: versiunea iniţială şi planurile de generare a prototipurilor dezvoltate în etapa de analiză a domeniului sunt revizuite din punctul de vedere al funcţionalităţii, al termenelor şi a descrierilor prototipurilor.
- *Analiza riscului*: analiza riscului determinată în etapa de analiză a domeniului este revizuită prin prisma scenariilor care au fost identificate şi a priorităţilor stabilite pentru dezvoltarea obiectelor.
- *Dicţionarul de date*: formatul acestuia este dependent de proiect şi include atribute asociate obiectelor ce suportă un anumit scenariu.
- *Fişe pentru obiectele de analiză*: sunt utilizate pentru evaluarea obiectelor de analiză. Această evaluare a fost derivată din analiza obiectelor lumii reale, dat fiind faptul că misiunea obiectelor de analiză este de a suporta diverse scenarii.
- *Specificaţia cerinţelor sistemului:* reprezintă documentul care precizează "ce" se dezvoltă în cadrul sistemului, formatul de prezentare a documentului adaptându-se contextului în care acesta va fi utilizat.

3.3.3 Proiectarea sistemului

Etapa de proiectare a sistemului abordează *alocarea cerinţelor sistemului componentelor hardware şi software*. Cele două activităţi de bază care au loc în această etapă sunt:

- *Partiţionarea* setului de cerinţe în subseturi ce vor forma baza arhitecturii sistemului, adică *în subseturi arhitecturale*.

BUPT

- *Configurarea* sistemului prin alocarea cerinţelor din cadrul fiecărui modul componentelor hardware şi software.

Cele două etape au uzual un caracter iterativ. Partiţionarea obţinută poate fi modificată pentru a obţine o cât mai bună arhitectură a sistemului, dat fiind setul de oerinţe şi de restricţii.

Proiectarea sistemului începe după finalizarea analizei cerinţelor sistemului. *Datele de intrare* sunt:

- specificaţia cerinţelor sistemului
- documentaţia generată în etapa de analiză a cerinţelor sistemului

Tranziţia de la analiza cerinţelor sistemului la proiectarea sistemului reprezintă un proces nontrivial, dat fiind faptul că în puţine cazuri soluţia este unică şi evidentă.

3.3.3.1 Partiţionarea

Procesul de partiţionare şi conceptul de partiţie depind în mod esenţial de perspectiva dezvoltatorului asupra sistemului proiectat. *Metoda utilizată* în procesul de partiţionare [NIE95] poate fi:

- *pur analitică* utilizând metode structurate bazate pe diagrame de flux a datelor, diagrame de flux a controlului, specificaţii de proces şi specificaţii de control [HAT87]
- *pur intuitivă* selectând subsistemele pe baza experienţei anterioare
- *gruparea obiectelor* bazată pe analiza orientată spre obiecte efectuată în etapele de analiză a domeniului respectiv de analiză a cerinţelor sistemului.

Scopul primar al partiţionării este de a descompune cerinţele unui sistem în unităţi mai mici, ce pot fi proiectate ca şi entităţi arhitecturale de către echipe distincte de ingineri hardware şi software. Entităţile individuale vor fi integrate în cadrul sistemului distribuit, soluţie capabilă de a implementa cerinţele sistemului şi condiţiile operaţionale specificate. Sistemul distribuit reprezintă soluţia arhitecturală preferată pentru a obţine flexibilitate şi extensibilitate.

BUPT

3.3.3.1.1 Perspective

Activitatea de partiționare este privită din perspective diferite în funcție de experiența și de formația dezvoltatorului. Există cel puțin trei perspective diferite asupra activității de partiționare [NIE95]:

- *Perspectiva subsistemelor.* Un dezvoltator obişnuit să dezvolte sisteme similare pentru diverse aplicații nu va insista asupra unei etape de partiționare formală. Ideea principală este de a identifica modificările ce trebuie efectuate asupra unor "black-box"-uri existente pentru a satisface noul set de cerințe.
- *Perspectiva elementelor de configurare.* Dezvoltarea unui sistem nou implică de obicei ideea conform căreia etapa de partiționare este menită să creeze subseturi care pot forma baza elementelor de configurare hardware respectiv software. Aceste elemente de configurare formează entități logice (dar nu neapărat entități ideale de proiectare) care pot fi create şi gestionate de către echipe de dezvoltatori.
- *Perspectiva liniei de produse.* Sistemele ce includ linii de produse în cadrul cărora ponderea componentei software este predominantă presupun partiționarea unui set de cerințe sistem în subentități ce pot fi implementate ca şi componente software. Elementele de configurare vor fi definite mai târziu, atunci când un anumit sistem este dezvoltat pentru un anumit beneficiar. Componentele software vor constitui elemente componente pentru arhitectura sistemelor mici, medii şi mari în cadrul aceluiaşi domeniu al problemei. O activitate de partiționare şi configurare formală nu este necesară în cadrul acestei abordări.

3.3.3.1.2 Partiții

O partiție reprezintă un subset al cerințelor unui sistem de timp real şi include următoarele caracteristici [NIE95]:

- este implementată sub formă software, hardware sau o combinație de elemente hardware şi software
- are interfețe robuste, bine definite
- este o entitate reutilizabilă ce poate fi alocată diferitelor noduri fizice în cadrul sistemului distribuit
- poate fi implementată să ruleze concurent cu alte partiții implementate într-o configurație distribuită

BUPT

- comunică cu alte partiţii exclusiv prin mesaje (date partajate pot fi utilizate în cadrul unei partiţii)

Partiţiile sunt create în mod normal în etapa de proiectare a sistemului şi alocate unui element hardware în timpul activităţii de configurare. Partiţiile vor fi considerate în continuare din perspectiva cerinţelor sistem şi a nodurilor virtuale ca elemente software care implementează o anumită porţiune a cerinţelor sistem. Astfel, o partiţie poate fi implementată prin intermediul unuia sau mai multor noduri virtuale.

3.3.3.1.3 Principii de partiţionare

În continuare vor fi enunţate câteva principii de partiţionare a unui sistem distribuit de timp real pentru fiecare din cele trei perspective descrise mai sus:

- *Perspectiva subsistemelor*:

  Punctul de plecare pentru partiţionarea subsistemelor este constituit de specificaţia cerinţelor sistemului care în mod normal include şi o diagramă de context a sistemului. Hatley-Pirbhai recomandă în [HAT87] utilizarea de dreptunghiuri cu colţurile rotunjite pentru a desemna module arhitecturale, astfel încât să existe o diferenţiere faţă de cercurile care reprezintă cerinţele esenţiale în cadrul analizei cerinţelor sistemului.

  Un exemplu de diagramă a contextului arhitectural este prezentată în [NIE95] pentru un sistem de gestiune a zborurilor:

BUPT

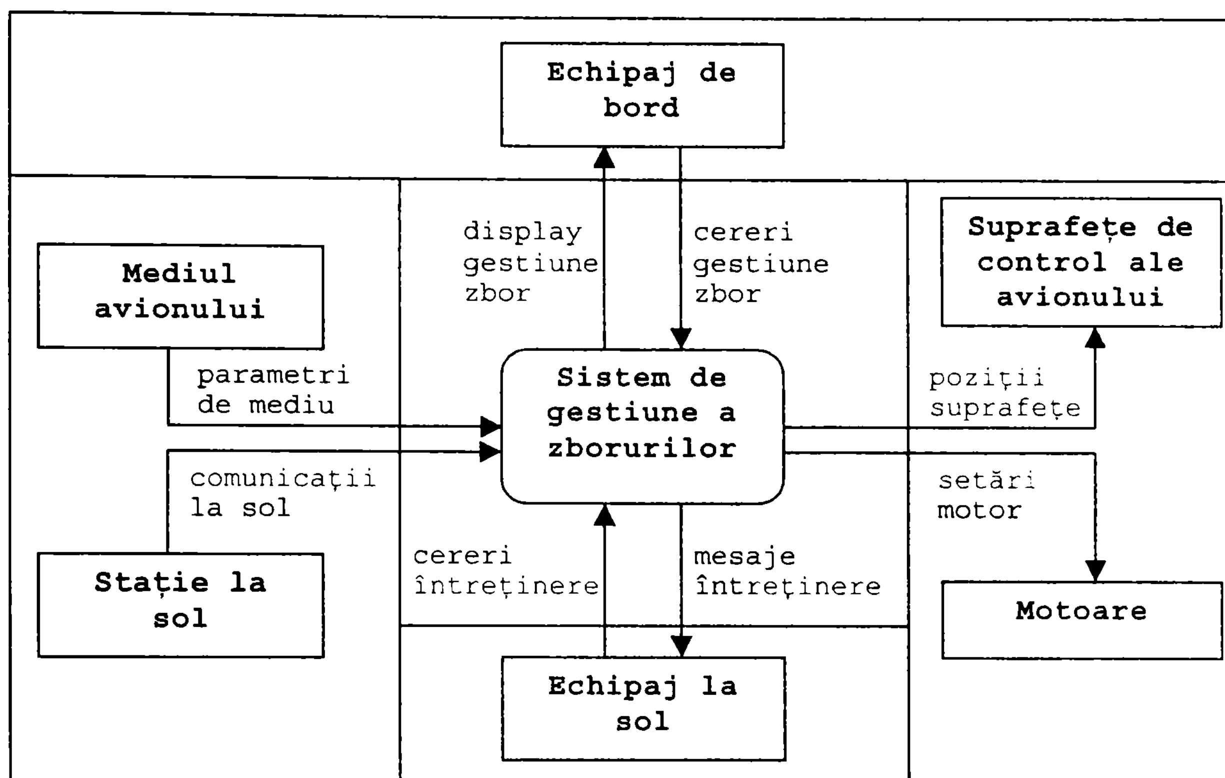


Figura 3.11 Diagrama de context arhitectural pentru sistemul de gestiune a zborurilor

Un subsistem este privit uneori ca şi o cutie neagră şi efectuează o anumită funcție specificată în cadrul cerințelor sistemului. În etapa de analiză a domeniului s-a încercat localizarea subsistemelor existente care pot fi utilizate, respectiv modificate pentru noul sistem. În acest sens este necesară o analiză a performanței pentru a determina dacă hardware-ul existent face față noilor cerințe. În caz afirmativ este necesară doar o intervenție la nivel software, activitatea de partiționare şi configurare formală ne fiind necesară. În acest caz proiectarea sistemului se va concentra asupra perfecționării produselor existente. În caz contrar este necesară reproiectarea arhitecturii hardware.

Dacă nu este disponibil un subsistem reutilizabil trebuie efectuată o partiționare tradițională. Alocarea funcționalităților la subsisteme poate fi făcută utilizând modelul Hatley-Pirbhai, respectiv modelele de control şi de proces.

- *Perspectiva elementelor de configurare.*

Obiectivul activității de partiționare este de a identifica *entități de configurare* care pot fi create de către echipe diferite de dezvoltatori. Pentru a genera un set de elemente de configurare software pot fi utilizate următoarele reguli [NIE95]:

- Elementele de configurare software vor fi determinate pe baza unei analize de alocare a funcționalității hardware şi software. Acest proces este iterativ, deci în

BUPT

urma selecției entităților de configurare software poate avea loc o modificare a alocării.

- Toate activitățile unui element de configurare software vor fi legate de o funcție sau de un grup de funcții.
- Dacă este posibil, fiecare element de configurare software va rula în cadrul unui singur nod hardware.
- Un element de configurare software compus din elemente reutilizabile poate fi legat de mai multe procesoare. Acest fapt este întâlnit în cadrul sistemelor de baze de date distribuite sau în sisteme în cadrul cărora controlul este distribuit mai multor componente software.
- Dimensiunea unui element de configurare software nu trebuie să depășească 100 – 150K linii de cod sursă sau 5 – 10 obiecte.
- Fiecare element de configurare software trebuie să fie testabil ca și entitate de sine stătătoare, precum și în cooperare cu alte elemente de configurare software. În acest sens vor fi explorate scenariile cu perspectivă operațională bazată pe thread-uri și cele ce suportă dezvoltarea incrementală. În cadrul unei abordări bazate pe scenarii un element de configurare software poate suporta unul sau mai multe scenarii.
- Un element de configurare software trebuie să se caracterizeze prin nivel scăzut de cuplare prin intermediul unor interfețe robuste spre interior și spre exterior.
- Numărul de elemente de configurare software trebuie minimizat pentru a minimiza numărul de documente necesare.
- Procesul de identificare a elementelor de configurare software trebuie să preceadă decizia de atribuire a acestora echipelor de dezvoltatori.

Un subsistem este considerat de obicei a fi un nod în cadrul unui sistem distribuit. Fiecare nod este format din unul sau mai multe elemente de configurare software și din entitățile hardware necesare pentru a suporta cerințele de performanță asociate funcționalității elementului de configurare software.

- *Perspectiva liniei de produse.*

Crearea unei linii de produse compusă din sisteme reutilizabile implică o strategie specifică de dezvoltare în ceea ce privește etapa de proiectare a sistemului. Partea de software va fi dezvoltată pentru a fi reutilizată pentru sisteme mici, medii sau mari ce se

BUPT

încadrează în acelaşi domeniu de probleme. O arhitectură hardware unică pentru toate aceste sisteme nu poate fi determinată printr-un proces tradiţional de partiţionare şi configurare. Arhitectura hardware se va dezvolta pentru a fi flexibilă şi expandabilă pentru cerinţele viitoare ale sistemului. Elementele software vor fi dezvoltate ca şi entităţi independente ce pot fi încadrate într-o arhitectură distribuită. Elementul critic în cadrul acestei abordări îl constituie mecanismul de comunicare interproces.

În cadrul acestei abordări nu este nevoie de o etapă formală de partiţionare şi configurare iar softul este dezvoltat independent de arhitectura hardware. Acest fapt se poate traduce prin cerinţe suplimentare legate de proiectarea mecanismului de comunicare interproces, cum ar fi controlul comunicaţiei pentru transferurile interproces şi interprocesor şi comunicaţia în cadrul unui mediu hardware eterogen.

3.3.3.1.4 Produse de lucru ale partiţionării

Produsele de lucru ale etapei de partiţionare enumerate în continuare sunt derivate din utilizarea metodei Hatley-Pirbhai [HAT87], bazată pe arhitectura sistemului:

- diagrama arhitecturală de context
- diagramele arhitecturale de flux
- diagramele arhitecturale de interconectare
- specificaţia arhitecturală de interconectare
- specificaţia modulelor arhitecturale
- dicţionarul arhitectural
- documentul de proiectare a sistemului
- documentul de descriere a mesajelor

3.3.3.2 Configurarea

Etapa de configurare se desfăşoară în strânsă legătură cu cea de partiţionare. În etapa de configurare, cerinţele din cadrul fiecărei partiţii sunt alocate hardware-ului respectiv software-ului. Se anticipează un proces iterativ de partiţionare şi configurare, dat fiind faptul că alocarea cerinţelor poate sugera o partiţionare mai eficientă decât cea iniţială.

Etapa de proiectare a sistemului se încheie odată cu desăvârşirea fazei de configurare. Începând din acest moment, proiectarea şi implementarea componentelor hardware şi software rezultate în urma analizei poate fi făcută de către echipe separate. Dezvoltarea componentelor

BUPT

hardware se va concentra asupra arhitecturii hardware iar dezvoltarea componentelor software continuă cu analiza cerinţelor software pentru fiecare componentă software.

3.3.3.2.1 Alocarea funcţională

Alocarea funcţională a cerinţelor sistem componentelor hardware respectiv software reprezintă o decizie esenţială a activităţii de proiectare pentru sistemele mari. Pe parcursul acestei faze are loc o revizuire a eventualelor decizii software luate în etapa de partiţionare respectiv a deciziilor hardware luate fără a lua în considerare spectrul de probleme software.

Problemele legate de soluţii distribuite în ceea ce priveşte proiectarea componentelor hardware şi software sunt rezolvate în faza de configurare. Modulele arhitecturale determinate în faza de partiţionare, reconfirmate sau realocate în această fază, sunt mapate la noduri fizice, fiecare nod fizic constând din una sau mai multe unităţi centrale de prelucrare, memoria locală aferentă şi mediul de interconectare. Dispozitivele globale, incluzând memoria şi dispozitive hardware fac de asemenea parte din soluţia arhitecturală hardware distribuită. Deciziile legate de proiectarea arhitecturală asociate acestei faze iau în considerare:

- numărul total de procesoare din sistem
- capacităţile de expansiune
- interfeţele procesoarelor
- mecanismul de intercomunicare al procesoarelor
- detecţia, raportarea şi revenirea din erori

Problemele specifice soluţiilor distribuite asociate proiectării software includ:

- mecanismul de comunicare interproces
- reconfigurarea statică
- reconfigurarea dinamică
- date locale şi date partajate
- baze de date distribuite
- detecţia, raportarea şi revenirea din erori

Alocarea cerinţelor sistemului poate fi efectuată doar componentelor hardware, unei combinaţii de componente hardware şi software sau doar componentelor software, caz în care este necesară o singură unitate centrală de prelucrare. Modul de efectuare a acestei alocări depinde în

BUPT

mare măsură de sistemul implementat. Un sistem de tipul "time-critical" va avea o mare parte a cerinţelor alocate componentelor hardware, pe când în cazul unui sistem ce prelucrează o cantitate mare de date va predomina componenta software.

3.3.3.2.2 Utilizarea metodelor structurate în cadrul configurării

Metoda de bază utilizată în etapa de configurare este legată de utilizarea aceloraşi modele arhitecturale Hatley-Pirbhai care au fost utilizate în etapa de partiţionare. Diagramele de flux a datelor respectiv a controlului pentru fiecare modul arhitectural sunt dezvoltate pentru a determina sciziunea între funcţiile software şi cele hardware.

Criteriile utilizate în partajarea alocării între componentele hardware şi cele software se referă la:

- aspectele legate de performanţă
- cerinţele legate de interfaţare
- cerinţele legate de mentenabilitate
- facilitatea testării
- reutilizabilitate
- fiabilitate
- disponibilitate
- cost şi planificarea timpului

3.3.3.2.3 Arhitectura hardware

Arhitectura hardware cuprinde următoarele specificaţii:

- procesoare
- memorie
- interconectări (magistrale, reţele locale şi interconexiuni fizice)
- descriere a nodurilor fizice.

Metoda Hatley-Pirbhai utilizează diagrama de interconexiune arhitecturală pentru descrierea modulelor arhitecturale şi a interfeţelor acestora. Aspectele majore ale arhitecturii hardware legate de dezvoltare, expansiune şi adaptabilitate se referă la:

- un răspuns mai rapid

BUPT

- funcţionalităţi adiţionale
- reconfigurarea elementelor reutilizabile pentru acelaşi domeniu al problemei pentru sisteme mici, medii şi mari
- compromisul între hardware şi software
- posibilităţile de upgradare hardware

Consideraţiile legate de toleranţa la defecţiuni se referă la:

- procesoare sau memorie redundantă
- reţele locale redundante
- reconfigurare
  - detecţia, raportarea şi revenirea din eroare
  - control software master-slave
  - echilibrarea încărcării
  - reconfigurarea manuală faţă de cea automată

Diagrama arhitecturală hardware nu descrie partiţionarea sistemului în module arhitecturale. Modulele arhitecturale prezentate în diagramele arhitecturale de interconexiune trebuie mapate la nodurile fizice descrise în diagrama arhitecturală hardware.

3.3.3.2.4 Produsele de lucru ale configurării

Produsele de lucru primare din faza de proiectare a sistemului includ:

- specificaţia arhitecturală hardware
  - noduri hardware (unităţi centrale de prelucrare, memorie, interfeţe)
  - arhitectura reţelei
  - dispozitivele de intrare / ieşire
  - diagramele arhitecturale hardware
- alocarea finală a cerinţelor
  - diagrame de flux al datelor
  - diagrame de flux al controlului
  - diagrame şi specificaţii ale modulelor arhitecturale
- maparea modulelor arhitecturale la nodurile fizice
- specificarea cerinţelor derivate

BUPT

- constrângeri de proiectare hardware şi software
- matricea de urmărire a cerinţelor sistem
- versiunea finală a documentului de proiectare a sistemului

3.3.3.2.5 Metode orientate spre obiecte pentru crearea subsistemelor

Aproape toate abordările de analiză orientată spre obiecte creează subsisteme prin gruparea obiectelor. Această activitate de grupare reprezintă un proces de tipul "bottom-up" care are loc după ce toate obiectele de analiză au fost identificate. Nu există o tratare explicită a alocării cerinţelor spre componente hardware respectiv software aşa cum este cazul în cadrul abordării Hatley-Pirbhai.

Metodele de analiză orientată spre obiecte care tratează crearea de subsisteme utilizează notaţiile de clasă-obiect pentru a descrie module arhitecturale. Avantajul constă în utilizarea aceluiaşi instrument de dezvoltare, dezavantajul – în faptul că aceeaşi notaţie este utilizată pentru concepte fundamental diferite, iar instrumentul nu face distincţia între entităţi arhitecturale şi obiecte de analiză. Metoda Hatley-Pirbhai, utilizând notaţii distincte (dreptunghiuri cu colţurile rotunjite pentru module arhitecturale), face distincţie clară între elementele arhitecturale cu interfeţele lor şi structura procesului. Metodele de analiză orientată spre obiecte utilizează propriile notaţii pentru asociaţii şi legături ce descriu conectivitatea arhitecturală.

Utilizarea aceloraşi notaţii OOA pentru analiza cerinţelor sistemului, proiectarea sistemului şi analiza cerinţelor software poate conduce la confuzii, fapt pentru care notaţia arhitecturală a metodei Hatley-Pirbhai este preferată faţă de oricare dintre notaţiile OOA.

Instrumentele CASE dedicate metodelor OOA sunt mai puţin mature în comparaţie cu cele dedicate metodelor structurate şi nu suportă procesul integral de dezvoltare. Instrumentele se concentrează doar asupra notaţiei şi a descrierii claselor, obiectelor, atributelor, operaţiilor şi asocierilor valabile pentru respectiva metodă.

3.3.3.2.6 Abordare bazată pe scenarii pentru crearea subsistemelor

Abordarea bazată pe scenarii caracteristică metodei OOSE [JAC92] [NIE95] descrisă în cadrul etapei de analiză a cerinţelor sistemului (vezi cap. 3.1.2) conţine câteva indicaţii pentru asocierea între scenarii, obiecte şi subsisteme:

- *Schimbările în cadrul sistemului*: cel mai important criteriu de formare a subsistemelor este legat de *predicţia comportamentului la schimbări*. Este de dorit ca *un subsistem să fie*

BUPT

legat de un singur actor, dat fiind faptul că schimbările sunt cauzate de obicei de către actori.

- *Funcționalitatea*: divizarea în subsisteme trebuie să se bazeze şi pe funcționalitatea sistemului. Obiecte strâns cuplate funcțional trebuie să facă parte din acelaşi subsistem. Următoarea euristică facilitează determinarea cuplajului funcțional între obiecte:
    - Modificările la nivelul unui obiect conduc la modificări la nivelul altor obiecte?
    - Obiectele înrudite comunică cu acelaşi actor?
    - Depind două obiecte înrudite de un al treilea obiect cum ar fi un obiect entitate sau un obiect de interfață?
    - Efectuează un obiect operații asupra celuilalt obiect?
- *Comunicarea*: în procesul de creare a subsistemelor trebuie avută în vedere minimizarea comunicației necesare între acestea. Aceasta se traduce printr-o cuplare redusă între subsisteme. Una dintre abordările posibile constă în a plasa un obiect de control în cadrul unui subsistem şi după aceea adăugarea de obiecte entitate şi de interfață strâns cuplate între ele. Subsistemul va fi constituit din această colecție de obiecte de control, entitate şi interfață.
- *Perspectiva produsului*: una dintre caracteristicile unui subsistem este legată de faptul că reprezintă cea mai mică entitate care poate fi achiziționată în cadrul unui sistem. Beneficiarului îi este oferit fie întreg subsistemul fie nici un element al acestuia ca şi entitate opțională. Astfel subsistemul va fi identificat cu un număr de produs într-un catalog de vânzări şi tratat ca şi componentă conectabilă în cadrul sistemului.

Euristica descrisă mai sus pentru identificarea subsistemelor utilizând abordarea orientată spre scenarii poate constitui o completare a metodei Hatley-Pirbhai şi constituie un test suplimentar pentru divizarea în subsisteme. Aceste elemente de euristică pot fi utilizate de asemenea ca elemente de sine stătătoare, caz în care subsistemele vor fi ilustrate utilizând dreptunghiuri cu colțurile rotunjite, analog notației Hatley-Pirbhai. Cerințele vor fi identificate în acest caz cu obiectele de control, de interfață şi entitate în locul proceselor Hatley-Pirbhai.

3.3.4 Analiza cerințelor software

Metoda prezentată în cadrul capitolului actual pentru procesul de dezvoltare software marchează debutul *analizei cerințelor software* ca moment în care *proiectarea sistemului* a fost încheiată, deci sistemul global a fost descompus într-un număr de subsisteme. Cerințele sistem au

BUPT

fost alocate componentelor hardware şi software, urmând analiza cerinţelor asociate fiecăreia dintre componentele software majore respectiv fiecărui subsistem.

Această etapă poate urma direct după etapa de analiză a cerinţelor sistemului în cazul în care pentru sistemul care se dezvoltă predomină componenta software. Acest caz poate fi întâlnit la dezvoltarea de componente software reutilizabile care vor fi individualizate pentru o anumită configuraţie hardware. Pentru sistemele exclusiv software, cum ar fi o bibliotecă de clase C++, etapa de analiză a cerinţelor software urmează imediat după analiza domeniului. Analiza cerinţelor sistemului şi proiectarea sistemului pot fi omise în acest caz.

Etapa de analiză a cerinţelor software se concentrează asupra cerinţelor fiecărei componente software în parte, şi nu asupra sistemului în globalitatea sa. Utilizarea modelelor OOA în această etapă permite rafinarea entităţilor lumii reale determinate în fazele de analiză a domeniului respectiv a cerinţelor sistem, descompunerea şi rafinarea de subsisteme sau definirea de clase şi obiecte abstracte dacă cele două etape precedente au fost omise. Clasele şi obiectele OOA determinate în actuala etapă reprezintă baza pentru transpunerea acestor elemente în entităţi de proiectare.

3.3.4.1 Modele

Etapa de analiză a cerinţelor software are rolul de a crea modelele care reprezintă instrumentul de tranziţie de la analiză la proiectarea software.

O *abordare structurată* a analizei cerinţelor software poate să se bazeze pe *analiza structurată de timp real* (RTSA), descrisă în Hatley-Pirbhai [HAT87], Ward-Mellor[WAR85] şi Shumate-Keller [SHU92]. Această metodă generează un *model comportamental* care poate fi utilizat ca instrument de tranziţie pentru abstractizarea procesului. Modelele de diagrame de flux de date respectiv de flux al controlului sunt combinate cu un set de reguli de selecţie a proceselor pentru a determina elementele concurente ale subsistemului. Modelul de concurenţă rezultat reprezintă arhitectura software care poate fi implementată cu ajutorul unui anumit limbaj de programare.

Un *model informaţional* poate fi adăugat pentru a obţine o mai bună perspectivă asupra domeniilor de date utilizând abordarea bazată pe relaţia între entităţi. Utilizarea diagramelor de relaţii între entităţi poate fi deosebit de utilă în cazul sistemelor ce prelucrează o cantitate mare de date. În cazul acestora, cerinţele derivate devin elemente importante ale modelului obiectelor. Diagrama de relaţii între entităţi poate fi utilizată pentru a determina obiecte. Dacă entităţilor din

BUPT

cadrul diagramelor de relații între entități le pot fi atașate operații și atribute, acestea pot fi adăugate celorlalte obiecte ale lumii reale obținute din documentul cerințelor sistemului.

Modelele orientate spre obiecte pot fi utilizate pentru a produce componente reutilizabile prin crearea de clase și obiecte ale lumii reale. Modelele RTSA respectiv OOA pot fi utilizate împreună și se pot completa reciproc. Modelele RTSA pun la dispoziție legăturile necesare între cerințele de timp real și proiectare iar modelele OOA suportă conceptele de reutilizare.

Pentru implementarea unei structuri de proiectare utilizând metode de proiectare orientate spre obiecte nu este strict necesară utilizarea analizei orientate spre obiecte. O structură validă de proiectare orientată spre obiecte poate fi obținută și pe baza metodei RTSA. Crearea de entități OOD depinde în primă măsură de paradigmele OOD suportate de limbajul de programare ales, nu de faptul că analiza orientată spre obiecte a fost sau nu folosită.

3.3.4.2 Analiza cerințelor software orientată spre obiecte

Modelele OOA utilizate în etapa de analiză a cerințelor software sunt aceleași cu cele utilizate în etapa de analiză a cerințelor sistemului. Singura diferență în utilizarea acestor modele constă în faptul că acum ne concentrăm asupra cerințelor componentelor software individuale și nu asupra întregului sistem. În acest moment un sistem mare este deja descompus în subsisteme.

Una dintre activitățile actualei etape de analiză este legată de identificarea claselor abstracte și a obiectelor care pot forma baza pentru obiectele de proiectare.

Avantajul primar al analizei software orientate spre obiecte este focalizarea de la bun început asupra entităților funcționale care mai târziu pot fi implementate ca și entități software reutilizabile în cadrul unui anumit limbaj de programare.

Pașii parcurși în etapa de analiză a cerințelor software orientată spre obiecte reprezintă extinderea și rafinarea analizei domeniului, a analizei cerințelor sistemului și a proiectării sistemului (etape care pot fi omise) și constau din:

- crearea de scenarii
- identificarea de clase și obiecte
- identificarea atributelor și operațiilor
- pregătirea perspectivelor asupra obiectelor
- modelarea datelor
- evaluarea claselor și a obiectelor

BUPT

3.3.4.3 Crearea scenariilor

Crearea scenariilor analizei cerințelor software reprezintă o rafinare a scenariilor identificate în etapa analizei cerințelor sistemului. Sunt adăugate detalii suplimentare pe măsură ce setul de cerințe este perceput şi înțeles mai bine. Pentru crearea scenariilor etapei actuale sunt valabile aceleaşi linii directoare ca şi în cazul analizei cerințelor sistemului.

3.3.4.4 Identificarea claselor şi a obiectelor

Crearea de clase şi obiecte de analiză a cerințelor software reprezintă activitatea de identificare a abstractizărilor de nivel înalt a entităților lumii reale. După ce aceste entități au fost identificate, lor li se asociază operații şi atribute. Punctul de plecare pentru această identificare este reprezentat de *specificația cerințelor software* ce a fost generată în faza de analiză a cerințelor sistemului pentru subsistemul curent. Dacă încă nu este disponibilă o specificație a cerințelor software se poate utiliza documentația legată de specificațiile sistem alocate subsistemului curent.

Clasele create şi rafinate în această etapă vor fi transpuse în entități de proiectare în etapa OOD pentru a fi implementate într-un limbaj de programare orientat spre obiecte în etapa OOP.

Setul de categorii de clase detaliat pe parcursul analizei domeniului poate fi utilizat ca şi element inițial pentru crearea de clase pornind de la cerințele funcționale ale subsistemului curent:

- clasele interfață dispozitiv
- clasele sistem extern
- clasele interfață utilizator
- clasele computaționale
- clasele rol
- clasele abstractizare a datelor

Clasificarea Jacobson [JAC92] poate genera obiecte adiționale în actuala etapă de analiză:

- obiecte entitate
- obiecte interfață
- obiecte de control

Utilizarea schemei de clasificare Jacobson este mai potrivită pentru subsisteme care conțin interfețe utilizator sau interfețe cu sisteme externe (inclusiv alte subsisteme). Obiectele interfață pot fi utilizate pentru a implementa un mecanism de comunicare interproces între subsisteme iar

BUPT

obiectele de control pot fi modelate ca şi agenţi ce se ocupă de procesarea mesajelor incidente şi de pregătirea mesajelor transmise de diferite procesoare.

3.3.4.5 Identificarea atributelor şi a operaţiilor

Atributele pot fi identificate prin determinarea datelor necesare descrierii proprietăţilor statice şi dinamice ale instanţelor unei clase. Pentru clasa „cont bancar" reprezentată în figura următoare valorile statice sunt reprezentate de „Nume" şi „Număr de cont" iar datele dinamice de adresă, balanţă şi beneficiar.

Cont
Nume Număr de cont Adresă Balanţă Beneficiar
Deschidere Închidere Adăugare Retragere Transfer

Figura 3.12 Atribute şi operaţii ale unei clase

Atributele trebuie atent verificate în ceea ce priveşte conţinutul lor. Un atribut complex care conţine mai multe elemente de date reprezintă mai degrabă o instanţă a unei clase ce nu a fost încă identificată.

Operaţiile pot fi identificate prin examinarea setului de servicii necesar pentru a afecta comportamentul unei instanţe a unei clase. Pentru clasa „Cont bancar" din figura de mai sus operaţiile sunt „Deschiderea", „Închiderea", „Adăugarea", „Retragerea" şi „Transferul".

Figura de mai sus reprezintă o modalitate de reprezentare grafică a atributelor şi operaţiilor asociate unei clase în etapa de analiză a cerinţelor software. Determinare justă a atributelor şi a operaţiilor presupune experienţă în acest domeniu. Unele dintre operaţii pot fi determinate pe baza verbelor din cadrul documentului de cerinţe software, dar un set complet poate fi determinat doar cunoscând domeniul problemei. Acest lucru este valabil mai ales pentru subsistemele mari unde multe dintre cerinţe sunt ascunse în abstractizări de mari dimensiuni.

BUPT

3.3.4.6 Reprezentarea obiectelor

Un model al obiectelor rezultă din determinarea asocierilor între diferitele obiecte ale lumii reale ce reprezintă abstractizări cheie ale domeniului problemei. Figura următoare prezintă un exemplu de model al obiectelor pentru un automat bancar. Sunt prezentate abstractizările principale şi relaţiile dintre ele. Una sau mai multe dintre aceste abstractizări pot reprezenta elemente de configurare software, de exemplu cartela împreună cu automatul bancar formează elementul de configurare software ATM.

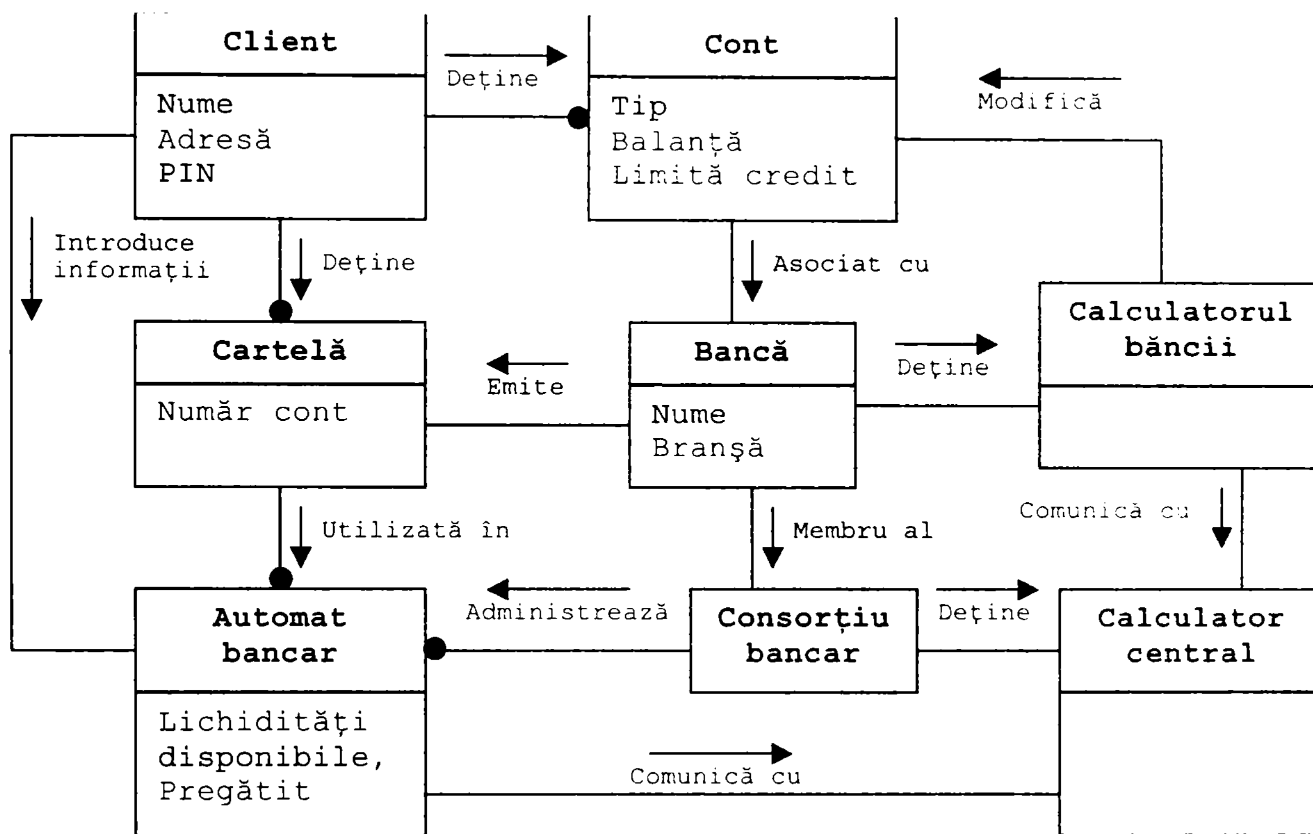


Figura 3.13 Modelul obiectelor pentru aplicaţia „automat bancar"

Relaţiile de moştenire nu sunt ilustrate în cadrul figurii anterioare. Acestea vor fi dezvoltate pe parcursul etapei OOD. Moştenirea nu reprezintă o perspectivă a abstractizărilor lumii reale şi aparţine activităţii de proiectare. Relaţiile de agregare vor fi de asemenea abordate doar în etapa OOD, dar uneori aceste relaţii sunt disponibile în specificaţia cerinţelor şi poate fi astfel inclusă în modelul obiectelor determinat în etapa de analiză a cerinţelor software.

BUPT

3.3.4.7 Modelarea datelor

Există o relaţie strânsă între entităţile determinate în faza de modelare a informaţiei şi crearea de clase abstracte şi de obiecte. Entităţile ilustrate în diagrama de relaţii între entităţi corespund în mod direct obiectelor lumii reale. Aceste entităţi pot avea asociate atribute dar în mod normal nu au operaţii asociate. Adăugând un set de operaţii entităţilor din diagrama de relaţii între entităţi se obţine echivalentul obiectelor OOA, deci o corespondenţă unu la unu cu obiectele identificate în faza de analiză orientată spre obiecte. Modelarea datelor poate fi astfel utilizată ca şi complement pentru determinarea abstractizărilor lumii reale prin adăugarea de reguli de identificare a operaţiilor.

3.3.4.8 Evaluarea claselor şi a obiectelor

Aspectele urmărite în ceea ce priveşte evaluarea claselor şi obiectelor determinate în actuala etapă sunt:

- completitudinea
- dimensiunea
- nivelul de detaliu
- atributele şi operaţiile asociate

Fişa de evaluare pentru etapa de analiză a cerinţelor software este similară fişei de evaluare utilizată în etapa de analiză a cerinţelor sistemului:

	Element revizuit		**Comentarii**
1.	Categoria obiectului		
2.	Completitudine		
3.	Dimensiune		
4.	Atribute		
5.	Operaţii		
6.	Identificator scenariu		
7.	Nivel de detaliu		
8.	Volatilitate		
9.	Dependenţa de		

BUPT

	Element revizuit		Comentarii
	implementare		
	- hardware		
	- software		
	- DBMS		
10.	Subsistem candidat		
11.	Agregare		
12.	Asociere		

3.3.4.9 Produse de lucru

Produsele de lucru principale generate în etapa de analiză a cerințelor software sunt:

- modelele de obiecte
- diagramele de relații între entități (dacă sunt utilizate)
- descrierile claselor şi obiectelor
- matricea de corespondență între obiecte şi entitățile din diagrama de relații între entități
- diagrame de trasare a evenimentelor pentru fiecare scenariu
- cartele CRC (dacă sunt utilizate)
- dicționarul de date

3.3.5 Proiectarea software

Proiectarea software urmează în mod normal după analiza cerințelor software. Scopul acesteia constă în generarea structurii software a aplicației prin crearea de componente arhitecturale software şi a interfețelor acestora.

În cadrul metodei propuse pentru procesul de dezvoltare software, un set inițial de entități de proiectare software este obținut prin transpunerea entităților de analiză în entități de proiectare. Acest set inițial va fi completat cu clase şi obiecte adiționale generate în cadrul etapei de proiectare software.

Pentru sisteme mari de timp real tranziția de la etapa OOA la etapa OOD are ca rezultat determinarea unui set de elemente concurente care reprezintă un model arhitectural de procese

BUPT

capabile de execuţie paralelă. Activitatea de transpunere include generarea unor scheme adecvate de transmitere de mesaje pentru arhitecturi distribuite.

Un aspect important abordat în etapa de proiectare software este *toleranţa le erori.* Asigurarea toleranţei la erori va fi realizată în cadrul metodei de dezvoltare software propuse cu ajutorul unei strategii de tratare a excepţiilor.

Obiectivele proiectării software sunt:

- descrierea transpunerii entităţilor OOA în entităţi OOD
- descrierea abstractizărilor proceselor pentru elemente concurente
- sumarizarea activităţii OOD şi a produselor asociate
- descrierea strategiei de tratare a excepţiilor
- generarea unei liste de criterii de evaluare

Subpaşii etapei de proiectare software sunt prezentaţi în cadru figurii următoare:

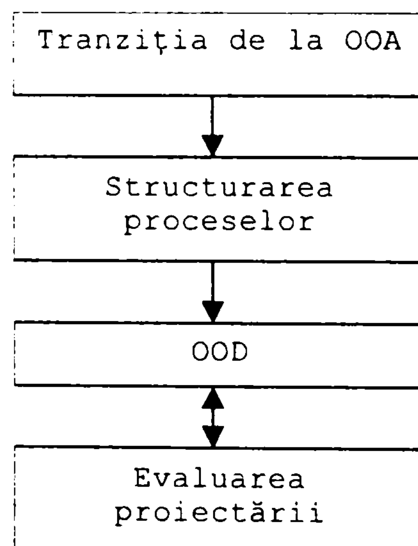


Figura 3.14 Subpaşii etapei de proiectare software

3.3.5.1 Tranziţia de la etapa de analiză

Etapa de proiectare software urmăreşte realizarea unei corelaţii între cerinţe şi entităţile de proiectare care pot fi implementate sub forma unuia sau mai multor programe ce pot rula ca task-uri distincte. Prima activitate din cadrul proiectării software este cea de transpunere a entităţilor de analiză în entităţi de proiectare.

BUPT

Următorul set de reguli de transpunere poate fi aplicat în cazul unui set de obiecte de analiză care cuprind cerințele unui program de tip single-thread fără a ține cont de aspecte legate de concurență [NIE95]:

- clase şi obiecte → clase
- atribute → date membre
- operații → funcții membre (metode)
- relații → moştenire, agregare şi utilizare
- diagrame de tranziție a stărilor → ierarhii de clase
- scenarii şi obiecte entitate → obiecte entitate
- modelul de flux al obiectelor → obiecte în cadrul subsistemelor
- diagrama de trasare a evenimentelor → mesaje transmise între obiecte

Una dintre cele mai importante controverse în ceea ce priveşte delimitarea între analiză şi proiectare se referă la *momentul abordării moştenirii*. Metoda de dezvoltare propusă aici pledează împotriva abordării moştenirii în etapa de analiză. Relațiile clasă-obiect determinate în etapa de analiză vor fi transpuse în relații de moştenire şi agregare în etapa de proiectare. Automatul bancar al cărui model al obiectelor a fost prezentat anterior va fi structurat utilizând moştenirea şi agregarea conform figurii următoare.

BUPT

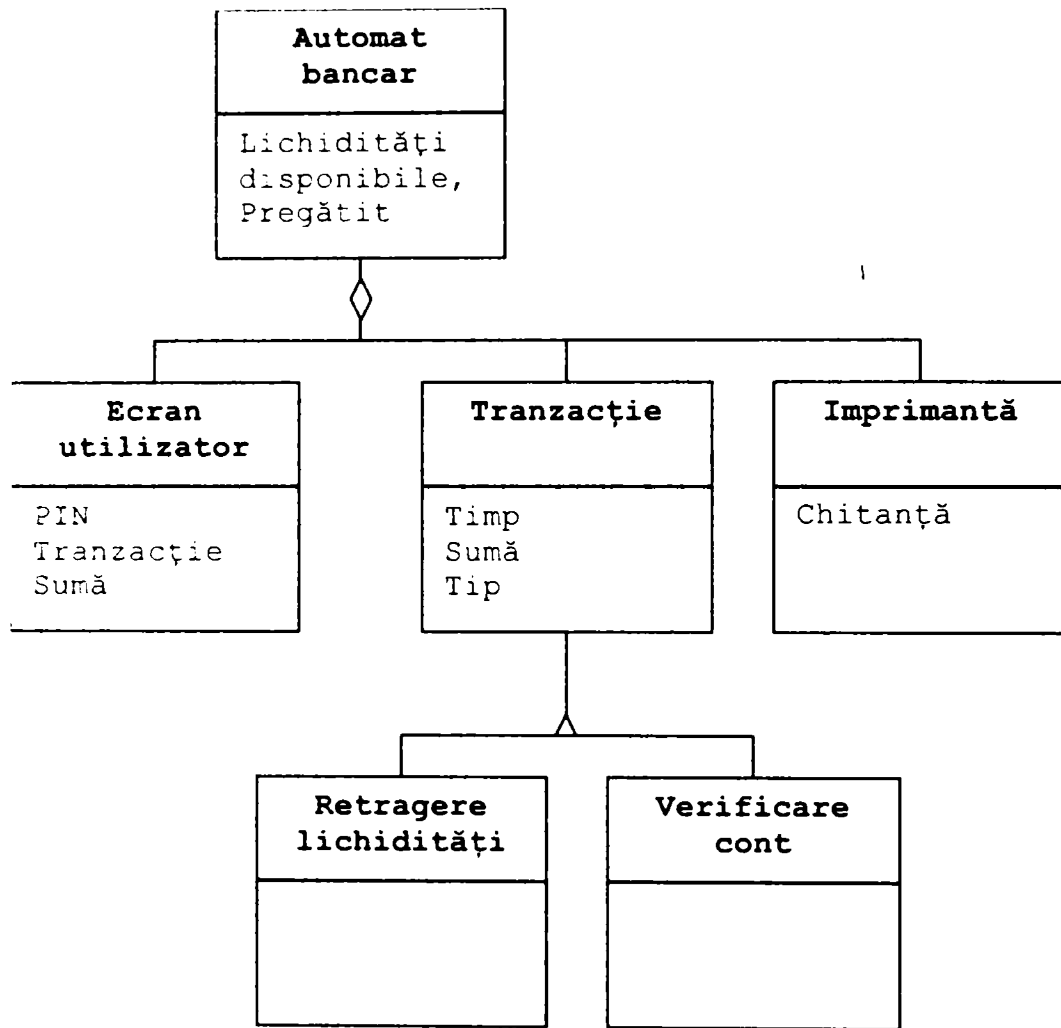


Figura 3.15 Model de moştenire şi de agregare

Comportamentul dinamic al sistemului este descris în etapa de analiză orientată spre obiecte prin *diagrama de tranziție a stărilor*. O metodă de tranziție a acestei diagrame spre etapa de proiectare a fost descrisă în [FAI93]. Figura următoare prezintă această soluție.

BUPT

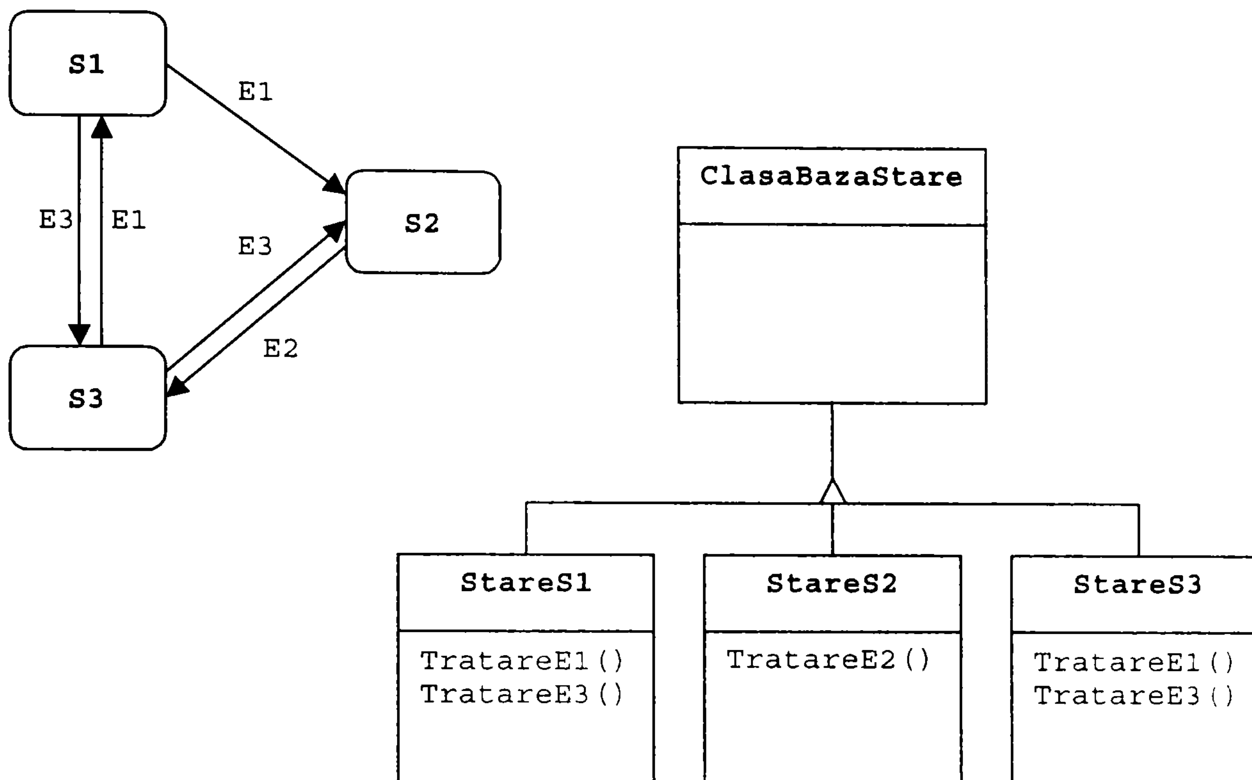


Figura 3.16 Diagramă de tranziție a stărilor și ierarhia de clase asociată

Ierarhia asociată diagramei de tranziție a stărilor are la bază clasa `ClasaBazaStare` care conține toate variabilele de stare. Pentru fiecare dintre stările prezente în cadrul diagramei de tranziție a stărilor se derivează câte o clasă `StareS`*`i`* care conține metode de tratare a evenimentelor care determină tranziția din starea curentă în altă stare.

3.3.5.2 Structurarea proceselor

Activitatea de structurare a proceselor creează un cadru pentru un model concurent iar paradigmele proiectării orientate spre obiecte sunt aplicate pentru a genera arhitectura software.

Termenul de proces este utilizat în actualul context ca și abstractizare a unei entități ce va fi implementată sub forma unui modul software care va fi executat concurent cu alte procese. Un exemplu în acest sens ar fi un proces Unix. Are loc astfel o supraîncărcare a semnificației utilizate pentru "procesul" de dezvoltare software.

Abstractizarea „proces" se utilizează în descompunerea unui sistem de timp real pentru a modela procesele concurente. Procesul creat poate să conțină procese concurente care comunică prin date partajate sau prin mesaje și procese distribuite care comunică prin mesaje.

Utilizarea abstractizării „proces" în proiectare implică satisfacerea a două condiții de către setul de procese secvențiale cooperante [NIE95]:

BUPT

- *Securitatea*: Fiecare proces trebuie să poată rula corect, independent de celelalte procese într-un domeniu static. O ilustrare pentru această condiție implică excluderea mutuală pentru datele și resursele partajate.
- *Viabilitatea*: Această condiție caracterizează proprietățile dinamice ale procesului. Proprietatea de viabilitate caracterizează un proces ce rulează fără a intra în stări de blocare („starvation") [DIJ68, BEN82] și reprezintă echivalentul „corectitudinii totale" pentru sistemele secvențiale.

Proprietatea de viabilitate este mai dificil de pus în evidență decât cea de securitate pentru că se referă la evenimente dinamice. Erorile de tip blocare sunt de obicei foarte dificil de depistat pentru că nu sunt reproductibile într-un mediu dinamic în continuă schimbare.

Fiecare proces constituie o mașină virtuală reprezentând un element paralel al soluției modelului. Procedura de descompunere a unui sistem de timp real într-un set de procese abstracte nu reprezintă un proces simplu și intuitiv. Aspectele care trebuie avute în vedere sunt legate de:

- elementele concurente
- selecția proceselor
- transmiterea mesajelor
- sistemele distribuite
- mecanismele de comunicare interproces

3.3.5.3 Proiectarea orientată spre obiecte

Subpasul OOD al proiectării software se ocupă de crearea componentelor arhitecturale și a interfețelor acestora astfel încât acestea să poată fi implementate cu ajutorul unui limbaj de programare orientat spre obiecte. Proiectarea interfețelor poate avea ca efect generarea de noi clase și obiecte, pe lângă cele care au fost deja identificate în etapele anterioare ale metodei de dezvoltare software propuse.

Clasele transpuse din faza de analiză sunt examinate pentru a verifica dacă atributele și operațiile satisfac liniile generale impuse în proiectare, adică pentru fiecare clasă a fost definit un set complet de atribute și operații. Aspectele care sunt stabilite în actualul subpas se referă la:

- vizibilitate
- nivelul de încapsulare și de ascundere a datelor.

BUPT

Arhitectura software descrie relațiile dintre clase şi este prezentată în diagrame de arhitectură a modulelor software.

3.3.5.4 Produse de lucru

Produsul principal al subpasului de structurare a proceselor este diagrama de structură a proceselor însoțită de o descriere a proceselor şi a interfețelor acestora. Este necesară şi includerea descrierii mecanismului de comunicare interproces însoțită de o justificare a soluției adoptate. Documentația mecanismului de comunicare interproces trebuie să includă toate restricțiile impuse, cum ar fi modul de conexiune, lungimea mesajelor, utilizarea datelor partajate, alocarea de spațiu tampon pentru coada de mesaje, identificarea mesajelor etc.

Produsele OOD cuprind toată documentația legată de structurarea proiectului:

- modelele obiectelor rafinate
- modelele de agregare şi moştenire
- diagrama de arhitectură a modulelor
- diagrame de trasare a evenimentelor pentru fiecare scenariu
- diagrame de tranziție a stărilor la nivelul fiecărui obiect
- descrierea claselor şi a obiectelor
- cartele CRC
- dicționarul extins de date

3.3.5.5 Tratarea excepțiilor

Una dintre cerințele cele mai importante în ceea ce priveşte un produs software este toleranța la erori. Considerarea toleranței la erori are loc în etapa de proiectare software, subpasul OOD.

Tratarea erorilor poate fi abordată în cadrul proiectării utilizând diferite soluții [NIE95]:

- simpla ignorare a erorii
- returnarea unei stări de eroare
- setarea unei variabile de eroare partajate
- utilizarea rutinelor de bibliotecă (de ex. assert())
- crearea de rutine de tratare a erorilor, definite de utilizator

BUPT

Tratarea excepțiilor se extinde dincolo de simpla tratare a erorilor şi constituie un element cheie pentru implementarea toleranței la defecțiuni a produsului software.

Aspecte legate de tratarea excepțiilor abordată din punct de vedere al programării structurate sunt prezentate în [FAR95]. Lucrarea de față se concentrează asupra tratării excepțiilor în cadrul sistemelor orientate spre obiecte. Tematica tratării excepțiilor în cadrul sistemelor orientate spre obiecte este abordată în [FAR2000].

3.3.5.5.1 Categorii de excepții

Următoarele categorii de excepții se recomandă a fi tratate în cadrul mecanismului general de toleranță la defecțiuni [NIE95]:

1. *Condiții excepționale anticipate*: reprezintă excepții care pot rezulta din condiții (rare) care nu ar trebui să întrerupă programul. Dacă de exemplu se asignează un set de numere consecutive pe post de identificatori, se poate obține o excepție de depăşire a capacității unui registru sau a unei locații de memorie în funcție de durata rulării programului. Rutina de tratare a excepției poate să reinițializeze setul de identificatori.
2. *Date de intrare eronate sau inconsistente*: pot fi detectate în cadrul unui modul software (server) care este apelat pentru a efectua un anumit serviciu. Serverul nu poate remedia datele introduse şi generează o excepție care este returnată apelantului. Remedierea situației este lăsată în seama apelantului.
3. *Erori hardware*: funcționarea dispozitivelor hardware este legată de obicei de o anumită perioadă de viață. Detecția unui eşec la nivelul unui dispozitiv hardware poate fi efectuată prin generarea unei excepții dacă respectivul dispozitiv nu răspunde în cadrul unei perioade de timp stabilite prin program.
4. *Erori neanticipate (erori de programare)*: erorile de programare sunt frecvente în cadrul produselor software, fiind necesară detecția şi raportarea acestora. În timp ce excepțiile din primele trei categorii pot avea nume explicite, excepțiilor rezultate din erori de programare nu li se poate asocia un nume. Acestea sunt generate de către sistem în timpul rulării şi ar trebui captate cât mai aproape de punctul în care au fost generate.

3.3.5.5.2 Strategia de proiectare a mecanismului de tratare a excepțiilor

În general, mecanismul de tratare a excepțiilor este proiectat ca parte a sistemului global. Aceasta este considerată a fi o *strategie de programare defensivă* care suportă toleranța la

BUPT

defecţiuni. Durata în care sistemul trebuie să continue funcţionarea după detecţia unei excepţii depinde de natura excepţiei. În unele cazuri excepţia poate fi doar raportată fără a întrerupe execuţia, altă dată excepţia are ca efect întreruperea execuţiei.

Una dintre problemele importante la nivel de proiectare este stabilirea locului unde urmează a fi tratată excepţia. De obicei tratarea excepţiei trebuie să aibă loc cât de aproape de locul unde a fost detectată. În cazul în care se urmăreşte protecţia software-ului server, de obicei tratarea excepţiei se face de către client iar excepţia este transmisă apelantului.

Mecanismul de tratare a excepţiilor selectat în faza de proiectare trebuie să poată fi suportat de facilităţile limbajului de programare utilizat pentru implementare. Un alt aspect important este ca mecanismul de tratare a excepţiilor să nu ducă la un consum exagerat de resurse în timpul rulării. Un astfel de mecanism este implementat în cadrul limbajului C++, fiind actualmente suportat de mai multe limbaje de programare.

3.3.5.6 Evaluarea proiectării

Cadrul în care se desfăşoară evaluarea proiectării pentru un produs software este reprezentat de revizuirile proiectului. Există două tipuri de revizuiri:

- *Revizuirile formale* sunt privite de obicei ca parte a interacţiunii cu beneficiarul produsului software. Scopul acestora este de a convinge beneficiarul că proiectul poate fi implementat respectând cerinţele de fiabilitate şi performanţă solicitate. O singură revizuire a proiectului poate să acopere întregul set de cerinţe sau se pot planifica mai multe revizuiri, fiecare acoperind cerinţele legate de un anumit stadiu.
- *Revizuirile informale* ale proiectului vor avea loc periodic şi continuu pe parcursul dezvoltării produsului. Aceste revizuiri au loc intern, fără asistenţa beneficiarului. Analiza materialului de proiectare are ca scop asigurarea că activitatea de proiectare este îndreptată în direcţia corectă.

Evaluare rezultatelor etapei de proiectare software pentru metoda de dezvoltare software analizată în cadrul actualului capitol are două componente:

- *evaluarea calitativă*, bazată pe urmărirea îndeplinirii unor criterii de calitate în ceea ce priveşte:
  - structura proceselor
  - structura de clase
  - interfeţele claselor

BUPT

- utilizarea moştenirii
- utilizarea excepţiilor

- *evaluarea cantitativă*, bazată pe utilizarea unor metrici software.

Aceste două componente vor fi prezentate în cele ce urmează.

3.3.5.6.1 Structura proceselor

Următoarele caracteristici vor fi evaluate pentru a constata dacă configuraţia de procese a fost aleasă în mod corespunzător [NIE95]:

- *Numărul proceselor*: utilizarea de procese multiple implică un efort suplimentar legat de fiecare dintre procesele din cadrul sistemului. Consumul de resurse creşte cu numărul de procese. Având în vedere aceste aspecte, numărul de procese din cadrul sistemului trebuie să fie cât mai mic.
- *Interacţiunile între procese*: interacţiunile între procese sunt evaluate pe baza deciziilor luate la nivel de apelant, respectiv apelat şi a transmiterii parametrilor.
- *Elemente de care depind procesele ciclice*: toate elementele de care depind procesele ciclice trebuie analizate cu grijă. Dacă sunt în concordanţă cu specificaţiile problemei iar soluţia este corectă, atunci proiectul este acceptabil, altfel va trebui reluat procesul de proiectare pe baza capacităţilor de planificare puse la dispoziţie de către mediul în care rulează aplicaţia.
- *Polling*: dacă procesul de polling este asociat cu "busy wait" [GEH84] atunci probabil că nu poate fi acceptat în proiect iar acesta va trebui modificat.
- *Date partajate şi excluziune mutuală*: dacă sunt utilizate date partajate, atunci proiectul trebuie să prevadă şi excludere mutuală pentru a păstra integritatea datelor. Datele partajate între procese vor fi de preferinţă protejate de un proces monitor.

3.3.5.6.2 Structura de clase

Evaluările unei clase cuprind aspecte legate de modul în care se încadrează o clasă într-o ierarhie, de setul său de operaţii şi de atribute:

- *Categoria clasei*: fiecare clasă ar trebui să fie un membru al clasificărilor bazate pe funcţionalitate sau pe scenarii. În caz contrar clasa este inadecvată sau trebuie adăugată o nouă clasificare.

BUPT

- *Încapsularea şi nivelul de abstractizare*: fiecare clasă trebuie să încapsuleze o singură abstractizare. Dacă nivelul de abstractizare este prea ridicat clasa trebuie descompusă în două sau mai multe clase mai mici.
- *Moştenirea*: o clasă derivată dintr-o clasă de bază trebuie să satisfacă o relaţie de tipul „is-a". În caz contrar clasa trebuie reproiectată ca şi clasă independentă sau conform unei relaţii de tipul „has-a".
- *Agregarea*: dacă în cadrul unei clase se utilizează agregarea atunci componentele trebuie să satisfacă relaţia „has-a" (sau parte întreagă)
- *Operaţii*: operaţiile asociate unei clase trebuie să formeze un set complet pentru abstractizarea încapsulată. Acolo unde este necesar se vor adăuga şi operaţiile complementare.
- *Atribute*: atributele selectate trebuie să reprezinte elemente de date asociate instanţelor clasei şi trebuie să fie în relaţie directă cu abstractizarea încapsulată. Atribute de mari dimensiuni sau atribute a căror cuplare cu clasa este redusă vor trebui reproiectate sub forma de clase adiţionale.

3.3.5.6.3 Interfeţele claselor

Privind clasa ca server pentru potenţiali clienţi, interfaţa acesteia este evaluată din perspectiva clientului. Evaluarea are loc prin prisma operaţiilor disponibile şi a nivelul de ascundere a informaţiei. Un alt aspect al evaluării se referă la posibilitatea de a utiliza clasa pe post de clasă de bază pentru o clasă derivată.

Criteriile de evaluare a interfeţei unei clase sunt următoarele:

- *Ascunderea informaţiei*: operaţiile disponibile pentru clienţi trebuie să fie vizibile (implementate în C++ ca şi *public*) iar elementele de date trebuie să fie ascunse (implementate în C++ ca şi *private*).
- *Operaţiile*: operaţiile pot fi evaluate conform următoarelor categorii:
  - *Funcţii manager*: sunt funcţii membre care realizează gestiunea iniţializărilor, a atribuirilor, a conversiilor de tip şi a gestiunii memoriei alocate dinamic (din această categorie fac parte constructorii, destructorii, copy-constructorii şi operatorii de atribuire [MIC97], ultimii doi pentru clasele ce conţin date alocate dinamic). Funcţiile manager sunt de obicei vizibile pentru clienţi.

BUPT

- *Funcții de implementare*: funcțiile din această categorie manipulează obiectele create ca instanțe ale clasei și reprezintă o porțiune a interfeței "contractuale" a clasei, deci sunt vizibile pentru clienți.
- *Funcții auxiliare*: funcțiile din această categorie efectuează operații auxiliare necesare celorlalte funcții membre. Având un rol strict de suport, sunt de obicei ascunse și nu sunt disponibile clienților.
- *Funcții de acces*: funcțiile de acces pun la dispoziție informații referitoare la datele membre ascunse din cadrul clasei. Diferența față de funcțiile de implementare este că ele doar citesc valorile datelor membre, fără a le putea modifica.

- *Trăsăturile claselor de bază*: la proiectarea unei clase se va ține cont dacă aceasta se va afla sau nu la baza unei ierarhii. Operațiile incluse trebuie să constituie un set de operații comune care pot fi utilizate de către clasele derivate care reprezintă specializări ale clasei de bază.

3.3.5.6.4 Utilizarea moștenirii

Următoarele aspecte sunt urmărite în ceea ce privește evaluarea proiectării prin prisma utilizării moștenirii:

- Numărul de nivele în structura de moștenire se recomandă a nu depăși valoarea 3 [NIE95]. În caz contrar, eforturile pe care le implică depanarea și testarea trebuie să fie justificate de avantaje evidente pe care le-ar oferi o adâncime mai mare a structurii de moștenire.
- Trebuie evitată situația în care toate subclasele sunt derivate dintr-o clasă de bază.
- Relația „is-a" trebuie să fie satisfăcută pentru toate subclasele unei clase de bază. Clasele părinte trebuie să fie stabile, având un set bine definit de operații care vor putea fi reutilizate sau suprascrise.
- Moștenirea multiplă se va utiliza doar atunci când este pe deplin justificată, date fiind dificultățile ce le implică în faza de implementare și cantitatea mare de modificări la nivelul proiectului în cazul în care oricare dintre clasele părinte este modificată.

BUPT

3.3.5.6.5 Utilizarea excepţiilor

Modul de utilizare a excepţiilor trebuie evaluat pe baza taxonomiei excepţiilor prezentată anterior. Utilizarea excepţiilor va fi specificată uzual în faza de proiectare de nivel înalt şi evaluată în faza de proiectare detaliată. Sunt acceptate următoarele utilizări:

- *Condiţii excepţionale anticipate*: se aplică condiţiilor ce pot apărea în timpul execuţiei unui program.
- *Protecţia softului server*: excepţiile sunt generate pentru a preveni utilizarea eronată a softului de uz general.
- *Raportarea eşecurilor hardware*: excepţiile sunt utilizate pentru a implementa un sistem tolerant la defecţiuni care să poată trata şi erori hardware.
- *Raportarea erorilor neanticipate (erori de programare)*: excepţiile din această categorie sunt utilizate pentru a detecta şi pentru a raporta erori de programare.

Faza de evaluare trebuie să conţină şi asigurarea că excepţiile nu au fost utilizate pentru controlul programului şi că excepţiile care nu sunt necesare au fost eliminate. O excepţie trebuie tratată cât mai aproape de locul unde a fost detectată.

3.3.5.6.6 Metrici software pentru evaluarea structurii de proiectare

Metrica software aleasă pentru evaluarea proiectului în cadrul metodei de dezvoltare software propuse este metrica LCOM, aparţinând setului definit de Chidamber şi Kemerer [CHI91]. Acest set cuprinde următoarele şase metrici software orientate spre obiecte, bazate pe teoria măsurării:

- DIT (*Depth of Inheritance Tree*): adâncimea arborelui de moştenire
- NOC (*Number of Children*): numărul de copii
- CBO (*Coupling between Objects*): cuplajul între obiecte
- RFC (*Response for a Class*): răspunsul pentru o clasă
- WMC (*Weighted Methods per Class*): pondere a metodelor pe clasă
- LCOM (*Lack of Cohesion of Methods*): lipsa de coeziune a metodelor

Definiţia metricii LCOM

Există diverse definiţii ale metricii LCOM, dintre care vor fi enumerate următoarele [ETZ98]:

BUPT

1. Definiția originală Chidamber şi Kemerer

Fie o clasă C_1 ce conține metodele $M_1, M_2, ..., M_n$. Fie $\{I_i\}$ setul de variabile ale instanței utilizate de către metoda M_i. Există „n" asemenea seturi, $\{I_1\}, ..., \{I_n\}$.

LCOM = numărul de seturi disjuncte rezultate prin intersecția celor „n" seturi.

Următoarele observații sunt valabile în ceea ce priveşte această metrică:

- Încapsularea obiectelor este cu atât mai bună cu cât coeziunea metodelor din cadrul unei clase este mai puternică.
- Lipsa de coeziune se interpretează de obicei ca necesitate de divizare a clasei în două sau mai multe subclase.
- Coeziunea redusă contribuie la mărirea complexității, sporind probabilitatea de apariție a erorilor în timpul procesului de dezvoltare.

2. Definiția Li şi Henry

Contribuția definiției lui Li şi Henry este legată de clarificarea noțiunii de disjuncție ce apare în definiția originală Chidamber şi Kemerer. Astfel:

LCOM = numărul de seturi disjuncte de metode locale; nu există două seturi care să se intersecteze; oricare două metode din acelaşi set au în comun cel puțin o variabilă locală de instanță, luând valori de la 0 la N, unde N reprezintă un întreg pozitiv.

3. Redefiniția Chidamber şi Kemerer

Chidamber şi Kemerer redefinesc metrica LCOM [HIT96, BAS96, CHI94] după cum urmează:

Fie clasa C_1 având n metode $M_1, M_2, ..., M_n$. Fie $\{I_j\}$ = setul de variabile ale instanței utilizate de către metoda M_j. Există n asemenea seturi $\{I_1\}, ..., \{I_n\}$.

Fie:

$$P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}$$

şi

$$Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$$

dacă toate cele n seturi $\{I_1\}, ..., \{I_n\}$ sunt vide atunci P = mulțimea vidă

$$LCOM = \begin{cases} |P| - |Q|, & \text{dacă } |P| > |Q| \\ 0 & \text{altfel} \end{cases}$$

BUPT

Metrica este definită deci ca fiind diferența dintre numărul de perechi de metode care nu au în comun nici o variabilă a instanței și numărul de perechi de metode care au în comun cel puțin o variabilă a instanței.

Dezavantajele acestei definiții sunt:

- metrica ia valoarea zero atunci când $| P | \leq | Q |$, deci LCOM = 0 nu reprezintă coeziune maximă, pentru că în mulțimea claselor cu LCOM = 0 unele pot prezenta un grad mai ridicat de coeziune decât altele.
- clase având nivele de coeziune foarte diferite pot prezenta aceeași valoare LCOM.

4. Definiția Hitz și Montazeri

Hitz și Montazeri propun o formulare diferită a versiunii Li și Henry a metricii LCOM, bazată pe teoria grafurilor.

Fie X o clasă, I_X setul de variabile ale instanței și M_X setul de metode ale acesteia. Considerăm un graf simplu unidirecțional $G_X(V, E)$ cu $V = M_X$ și E = {<m,n> este un element al V x V | pentru fiecare i care este un element al lui I_X : (m accesează pe i) intersectează (n accesează pe i)}, deci exact acei vertecşi sunt conectați care reprezintă metode care au cel puțin o variabilă a instanței comună. LCOM(X) este definită ca fiind numărul de componente conectate ale lui G, adică numărul de „clustere" ce operează pe seturi disjuncte de variabile ale instanței.

Utilizarea unei metrici LCOM pentru evaluarea gradului de coeziune al unei clase

Calculul valorii LCOM pentru o clasă va trebui să țină cont de următoarele aspecte:

- setul de variabile membre poate fi extins și la variabilele membre moștenite
- în cadrul setului de metode ale clasei se poate include și constructorul și destructorul clasei

Nici una din definițiile metricii LCOM enumerate mai sus nu conțin specificații referitoare la cele două aspecte enunțate mai sus. Un studiu de caz în acest sens a fost efectuat de Letha Etzkorn, Carl Davis și Wei Li în [ETZ98] asupra definiției Chidamber și Kemerer revizuită și a definiției Li și Henry. Cazurile în care a fost calculată valoarea LCOM sunt:

1. cu considerarea constructorului; cu considerarea moștenirii
2. cu considerarea constructorului; fără considerarea moștenirii
3. fără considerarea constructorului; cu considerarea moștenirii

BUPT

4. fără considerarea constructorului; fără considerarea moştenirii

Utilizând ca reper expertiza unor autorităţi recunoscute în domeniu pentru analiza bazată pe regresia liniară, concluzia a fost că valoarea LCOM care reflectă cel mai fidel gradul de coeziune al unei clase este cea obţinută conform cazului 2 (definiţiei Li şi Henry, pentru calcul ignorând variabilele moştenite şi incluzând constructorul).

Un exemplu de calcul al valorii LCOM conform definiţiei Li şi Henry va fi prezentat în cadrul capitolului următor pentru clasele din cadrul bibliotecii de ferestre.

3.3.6 Implementarea

Implementarea reprezintă ultimul pas în procesul de dezvoltare. Arhitectura software derivată dintr-un set de cerinţe software este creată utilizând facilităţile limbajului de programare ales. Implementarea poate avea loc pentru întreaga aplicaţie, dar de obicei se efectuează incremental pentru o anumită versiune ce corespunde unei porţiuni bine definite a cerinţelor.

În etapa de implementare, activitatea de programare este susţinută de o strategie de testare pentru detectarea erorilor. Unul dintre instrumentele foarte utile în această fază este debugger-ul simbolic care poate fi adaptat unui anumit şablon de testare, punctele de întrerupere facilitând examinarea porţiunii staţionare a programului aflat în rulare.

3.3.6.1 Tranziţia de la etapa de proiectare

Entităţile de proiectare create în etapa OOD vor fi implementate utilizând facilităţile limbajului C++. Deciziile luate în etapa de proiectare vor fi implementate direct utilizând structuri adecvate sau făcând uz de caracteristici ale limbajului de programare. Tranziţia de la faza de entităţi de proiectare la faza de entităţi de programare cuprinde următoarele transformări:

- clase → clase C++
- operaţii → funcţii membre
- atribute → date membre
- ascunderea informaţiei → segmente *private* şi *protected* în interfaţa clasei
- relaxarea ascunderii informaţiei → utilizarea noţiunii *friend*
- ierarhia de moştenire → utilizarea noţiunilor de moştenire

BUPT

- agregarea → declararea datelor membre ca şi obiecte ale altor clase
- modularitatea → separarea fişierelor header de fişierele sursă
- tratarea excepţiilor → noţiuni de excepţii C++ sau *assert()*

Aceste transformări pun la dispoziţie un set iniţial de elemente care vor fi programate. În timpul activităţii de programare se descoperă de obicei necesitatea existenţei unor entităţilor de programare adiţionale. Astfel are loc un proces iterativ între etapele de proiectare şi implementare.

3.3.6.2 Programarea

Activitatea de programare conţine implementarea claselor, a obiectelor şi a structurii software ierarhice.

Limbajul selectat pentru etapa de implementare în cadrul metodei de dezvoltare descrise în cadrul capitolului actual este C++. Acesta reprezintă un limbaj de programare orientat spre obiecte ce prezintă suport nativ pentru moştenire şi suport run-time pentru polimorfism. Dat fiind faptul că C++ a fost derivat ca şi superset al limbajului C, se va utiliza o strategie hibridă ca şi combinaţie de programare procedurală, modulară şi orientată spre obiecte.

Clasele transpuse din faza de proiectare vor fi implementate ca şi tipuri de date abstracte încapsulate sub formă de clase C++. Nivelul de ascundere al informaţiei a fost determinat în faza de proiectare şi va fi implementat sub formă de segmente *public*, *private* şi *protected*. Unele dintre tipurile de date abstracte cum ar fi stivele, cozile, listele vor fi implementate utilizând metaclasele (*template*) pentru a maximiza reutilizarea acestora în cadrul structurilor care diferă doar prin tip. Ierarhiile de moştenire vor fi implementate utilizând trăsăturile OOP ale limbajului C++, cum ar fi structuri pentru crearea de clase de bază şi clase derivate. Relaţiile de agregare sunt implementate declarând clasele agregate ca obiecte membre în cadrul clasei care le conţine.

Funcţiile membre asociate entităţilor de proiectare transpuse trebuie verificate pentru completitudine. Crearea de constructori şi destructori reprezintă o activitate de programare şi va fi efectuată în etapa actuală.

Programarea modulară este utilizată pentru a facilita reutilizarea şi întreţinerea. O structură de fişiere va fi pusă la punct pentru a minimiza durata compilării în timpul dezvoltării, precum şi în cazul inevitabilelor modificări ale cerinţelor. Acest aspect este rezolvat în C++ de obicei prin utilizarea fişierelor header pentru definirea interfeţelor. Codul sursă C++ este plasat în fişiere diferite. Această separare a specificaţiilor de implementare facilitează o strategie de compilare incrementală. Dacă interfeţele nu se modifică softul client nu trebuie recompilat.

BUPT

3.3.6.3 Tratarea excepţiilor

Detecţia şi tratarea evenimentelor excepţionale în faza de rulare reprezintă cheia realizării unui sistem tolerant la defecţiuni. Strategia de proiectare pentru toleranţa la defecţiuni se stabileşte în etapa de proiectare software. Pentru implementarea mecanismului de tratare a excepţiilor pot fi utilizate următoarele metode:

- utilizare funcţiei C *assert()*
- utilizarea mecanismului intern C++ de tratare a excepţiilor
- combinarea celor două metode

Mecanismul C++ de tratare a erorilor este disponibil acum în majoritatea mediilor de dezvoltare. Efortul implicat de includerea mecanismului de tratare a excepţiilor se va compensa prin evitarea propagării erorilor nedetectate care pot avea efecte fatale asupra aplicaţiei.

În cele ce urmează se vor prezenta două soluţii pentru implementarea unui mecanism de tratare a excepţiilor în C++.

3.3.6.3.1 Utilizarea mecanismului intern C++ de tratare a excepţiilor

Excepţiile pot fi considerate obiecte şi declarate ca obiecte agregate în specificaţia clasei. Acest aspect va fi exemplificat în cele ce urmează în contextul unei clase ce implementează operaţii cu vectori. Clasa este utilizată în cadrul editorului 2D a cărui dezvoltare se urmăreşte în cadrul capitolului următor.

```
class CVector
  {
  public:
    CVector(//constructor
            double x,
            double y,
            double z
           );

    double VectLength();    //returnează lungimea vectorului
    void Versor( //determină cosinuşii directori
                double &cosX,
                double &cosY,
                double &cosZ
               );
//...

    class ZeroDivide{};      //clasă excepţie
    class OutofRange{};      //clasă excepţie
```

BUPT

```
      class Overflow{};          //clasă excepţie
      class InvalidData          //clasă excepţie cu date membre
        {
        public:
          int      index;        //date membre asociate
          double   valoare;      //excepţiei InvalidData
        };

    private:
      double vX,    //componentele vectorului
             vY,
             vZ;
//...
    };
```

Excepţia, fiind un obiect, poate fi de orice tip şi nu trebuie specificată ca şi clasă. Un set de excepţii poate fi grupat în cadrul unui tip enumerare [STR91, p.302]:

```
enum MathException {OutofRange, Overflow, Underflow, Zerodivide}
```

Excepţii înrudite pot fi declarate ca parte a unei ierarhii de clase:

```
class MathException {};
class OutofRange : public MathException {};
class Overflow :   public MathException {};
class Underflow :  public MathException {};
class Zerodivide : public MathException {};
```

O excepţie C++ este generată prin intermediul expresiei `throw`. Acest aspect va fi exemplificat în cadrul implementării clasei CVector:

```
void CVector::Versor(
                     double &cosX,
                     double &cosY,
                     double &cosZ
                    )
  {
  double vLen = VectLength();
  if(vLen > PRECISION)
    {
    cosX = vX / vLen;
    cosY = vY / vLen;
    cosZ = vZ / vLen;
    }
  else
    throw ZeroDivide;
  }
```

BUPT

Excepţia generată în cadrul metodei Versor poate fi interceptată în cadrul codului ce apelează această metodă. În funcţie de context se poate încerca o revenire din eroare sau, mai frecvent în cazul concret de mai sus informaţia va fi utilă în faza de depanare pentru a preveni apariţia excepţiei la utilizarea clasei CVector de către client.

3.3.6.3.2 Mecanism de aserţiuni bazat pe excepţii implementat pentru C++

O importanţă deosebită în dezvoltarea unei aplicaţii o are acurateţea transpunerii în cod a deciziilor luate în faza de proiectare. Un mecanism menit să asigure acest deziderat este oferit de către macrodefiniţia assert(), pe care majoritatea compilatoarelor o implementează sub forma următoare:

```
#if defined DEBUG
#define assert(expr)  (expr)?(void)0:_assert(#expr,__FILE__,__LINE__);
#else
#define assert(expr)
#endif
```

Utilizarea acestui macro reprezintă o soluţie foarte bună în contextul programării structurate. Utilizarea lui assert() în cadrul unui sistem orientat spre obiecte întâmpină următoarele neajunsuri:

- assert() are ca efect terminarea imediată a programului, fără a-i da posibilitatea de a încerca o revenire din eroare. Acest comportament nu este acceptabil pentru sisteme tolerante la defecţiuni.
- Utilitatea lui assert() este legată doar de faza de compilare, dat fiind faptul că în versiunea de distribuţie a aplicaţiei constanta DEBUG nu este definită, deci întregul mecanism bazat pe aserţiuni nu este funcţional în varianta finală a produsului.
- Utilizarea lui assert() în cadrul unei biblioteci de obiecte furnizată în formă compilată relevă doar erori de logică internă; este de dorit ca mecanismul de tratare a excepţiilor să intercepteze şi erori de utilizare „externă" a componentelor din cadrul bibliotecii.
- Un alt neajuns al lui assert() este legat de faptul că mesajul de eroare este direcţionat spre dispozitivul „standard error", fără a da programatorului posibilitatea de a modifica acest lucru. În unele medii grafice dispozitivul „standard error" este mapat la dispozitivul NULL, suprimând astfel mesajele de eroare generate de către assert().
- Unele condiţii supuse testului cu macrodefiniţia assert() pot să conţină la rândul lor apeluri de funcţii. Aceste apeluri nu vor fi efectuate în versiunea de distribuţie a

BUPT

programului, rezultând un comportament diferit a versiunii DEBUG față de cel al versiunii RELEASE.

Pe baza observațiilor de mai sus se va dezvolta în continuare un mecanism de aserțiuni bazat pe excepții. Acesta este descris în cadrul lucrării *Exception-based Assertion Mechanism for Object-Oriented Systems used in the Implementation of a 2d-Editor for a CAD/CAM System*, prezentată de către autor în cadrul Conferinței internaționale CONTI'2000, Timișoara, 12-13 oct.2000 [FAR2000].

Un prim pas în dezvoltarea mecanismului de aserțiune constă în definirea clasei Assertion:

```
class Assertion
{
    Assertion(
                const char *expr,
                const char *file,
                int line
               )
               :
               theExpr(expr),
               theFile(file),
               theLine(line)
       {
       }

    char *theExpr;
    char *theFile;
    int  theLine;
};
```

În continuare se redefinește funcția _assert():

```
inline void _assert(
                     char *expr,
                     char *file,
                     int line
                    )
           {
           throw Assertion(expr, file, line);
           }
```

Aceasta este apelată în cadrul macrodefiniției assert() prezentate mai sus:

```
#if defined DEBUG
#define assert(expr)  (expr)?(void)0:_assert(#expr,__FILE__,__LINE__);
#else
#define assert(expr)
```

BUPT

```
#endif
```

Avantajele abordării de mai sus sunt:

- Faza de detecţie a erorii este separată de faza de raportare a ei şi de cea de tratare, eventual revenire.
- Aserţiunea poate fi propagată în afara frontierelor componentei software în cadrul căreia a apărut eroarea.

Dezavantajele care persistă în ceea ce priveşte această abordare sunt următoarele:

- Macrodefiniţia este controlată de variabila booleană DEBUG. Nu există posibilitatea de validare selectivă a diverselor tipuri de verificări.
- Aserţiunile care conţin acţiuni vor modifica logica programului în varianta RELEASE.
- Clienţii componentei software în cauză partajează responsabilitatea pentru tratarea erorilor. Acest aspect poate deveni sursă de confuzie sau de ineficienţă: dacă atât componenta software cât şi clientul verifică condiţiile ce ar putea genera erori rezultă un program ineficient, dacă nici una din părţi nu efectuează acest lucru rezultatul ar putea fi dezastruos.

Înlăturarea ultimului dintre dezavantajele menţionate mai sus va fi efectuată recurgând la o abordare a mecanismului de aserţiuni bazat pe excepţii după modelul programării bazate pe contract [MEY88]. Conceptul de programare bazată pe contract este suportat în mod nativ de limbajul Eiffel şi este descris în cele ce urmează.

Programarea bazată pe contract

Programarea bazată pe contract (dezvoltarea bazată pe contract) reprezintă o paradigmă de dezvoltare introdusă de Bertrand Meyer, creatorul limbajului Eiffel. Premisa principală a programării bazate pe contract este reprezentată de faptul că specificaţia unui element de program cum ar fi o funcţie sau o clasă poate fi tratată ca şi un contract între utilizatorul elementului şi elementul însuşi.

Paradigma programării pe bază de contract este următoarea [MEY88]: *„Dacă clientul (apelantul) îndeplineşte anumite cerinţe atunci când utilizează furnizorul, atunci furnizorul (funcţia apelată) garantează producerea unui rezultat pertinent. Beneficiul din partea clientului este de a şti că furnizorul produce întotdeauna un rezultat valid atât timp cât acesta respectă partea sa de contract. Furnizorul beneficiază de asemenea ştiind că trebuie să furnizeze un*

BUPT

rezultat pertinent atunci când clientul îl utilizează în mod corect (în termenii contractului). Atunci când termenii contractului sunt expliciți atunci obligațiile de detecție a erorilor devin clare."

Un contract este specificat pe baza unor aserțiuni și este formulat în contextul furnizorului. Aceste aserțiuni se încadrează în următoarele tipuri:

- *Precondiții*: sunt condițiile pe care clientul trebuie să le respecte pentru ca funcția sau clasa furnizorului să își îndeplinească cu succes sarcina menită. Instrucțiunea „require" definește precondiții.
- *Postcondiții*: sunt condiții pe care trebuie să le îndeplinească furnizorul după execuția unei metode care a fost utilizată în mod corect de către client. Instrucțiunea „ensure" este utilizată pentru a defini postcondiții.
- *Condiții in-line*: în timpul execuției unei funcții puse la dispoziție de un furnizor este utilă verificarea existenței unor condiții necesare rulării corecte, condiții ce depind de date externe furnizorului. Un exemplu tipic ar fi apelurile externe spre rutine scrise în C sau în limbaj de asamblare sau apelul funcțiilor API ale sistemului de operare. Astfel de verificări sunt efectuate prin intermediul instrucțiunii „check".
- *Invarianți*: acest tip de aserțiune se referă la clasa furnizor ca ansamblu și specifică ce condiții trebuie îndeplinite pentru ca o instanță a clasei să fie într-o stare validă. Invariantul descrie, cel puțin parțial, semantica clasei. Clauza „invariant" a unei clase este utilizată pentru a defini invarianții clasei.

Spre exemplificarea paradigmei programării bazate pe contract urmează un fragment al specificației Eiffel al unei clase Stiva prevăzută cu aserțiunile corespunzătoare:

```
class Stack[T]
export push, pop, top, full, empty, size, capacity
push(item:T)
  -- adăugarea unui element pe vârful stivei
require
  not_full:size < capacity
ensure
  top=item
  size=old size + 1
do ... end -- push

pop
  -- extragerea elementului din vârf
require
  not_empty: size > 0
ensure
```

BUPT

```
    size = old size - 1
  do ... end -- pop

  top:T
    -- returnează elementul din vârf
  require
    not_empty: size > 0
  ensure
    no_change
  do ... end -- top

  size:INTEGER
    -- numărul curent de elemente din stiva

  capacity:INTEGER
    -- capacitatea maximă a stivei

  invariant
    -- numărul curent al elem. din stivă în interval valid
    (size >= 0) and
    (size <= capacity)

    -- definirea comportamentului de tip LIFO
    (size > 0) and then
    top = push(top)

  end -- Stack
```

Eiffel pune la dispoziția programatorului un mecanism foarte disciplinat de tratare a excepțiilor. Atunci când o aserțiune eşuează excepția este propagată apelantului. Dacă apelantul dispune de o rutină de tratare a excepției, denumită în Eiffel clauză „rescue" atunci controlul este transferat acesteia. În caz contrar, excepția este propagată în sus de-a lungul stivei până când se întâlneşte o rutină de tratare a excepției. În cazul în care nu există nici o asemenea rutină, controlul este predat unei rutine implicite de tratare a excepțiilor ce aparține mediului, care informează utilizatorul despre apariția excepției şi determină părăsirea programului.

Fiecare dintre tipurile de aserțiuni pot fi validate individual în Eiffel. În versiunea de livrare a programului rămân de obicei validate doar precondițiile, celelalte aserțiuni fiind utilizate doar în faza de depanare pentru a verifica consistența codului scris.

Abordare prezentată mai sus reprezintă fără îndoială o soluție mai bună decât cea bazată pe clasa „Assertion" împreună cu redefinirea funcția _assert() prezentată anterior. Modelul de programare bazată pe contract implementat în cadrul limbajului Eiffel va fi transpus în cele ce urmează în C++.

Implementarea C++ a programării pe bază de contract suportă următoarele tipuri de aserțiuni:

BUPT

- REQUIRE: enunță precondițiile.
- ENSURE: enunță postcondițiile.
- CHECK: verifică apelurile externe.
- IMPLIES: verifică o condiție care trebuie să fie îndeplinită dacă o altă condiție este îndeplinită.
- NEVER_GET_HERE: verifică faptul că o anumită cale nu este abordată niciodată. În mod tipic asigură ca opțiunea „default" a unei instrucțiuni „case" să nu primească niciodată controlul.
- INVARIANT: enunță condițiile care trebuie să fie îndeplinite pentru ca un obiect al clasei să se afle într-o stare validă.
- ASSERT: enunță o singură proprietate din cadrul aserțiunii INVARIANT.
- CHECK_INVARIANT: testează toate proprietățile aserțiunii INVARIANT. În C++, verificările de INVARIANT trebuie apelate explicit în cadrul funcțiilor membre ale clasei.
- BASE_INVARIANT: testează secțiunea INVARIANT a unei clase moștenite. Poate fi utilizat doar cu un bloc INVARIANT.
- FOR_ALL: aplică un test ce trebuie să furnizeze un rezultat adevărat pentru toți membrii unei colecții.
- THERE_EXISTS: aplică un test care trebuie să dea rezultat adevărat pentru cel puțin un membru al unei condiții.
- USES_OLD: creează o copie a obiectului curent denumită „old"

Aserțiunile de mai sus vor fi implementate sub forma unor macrodefiniții, dat fiind faptul că limbajul C++ nu oferă suport pentru aceste aserțiuni. O limitare a limbajului C++ constă în faptul că aserțiunile pentru funcțiile membre trebuie să apară la nivelul implementării și nu a specificației clasei. Acest neajuns va fi înlăturat prin repetarea contractului în cadrul specificației clasei.

Codul sursă C++ pentru mecanismul de excepții bazat pe aserțiuni este prezentat în anexa lucrării. Se observă plasarea în cadrul funcțiilor membre ale clasei a clauzelor REQUIRE și ENSURE. REQUIRE reprezintă partea de contract ce trebuie îndeplinit de către client atunci când apelează metoda, așa că această clauză apare imediat la începutul fiecărei metode, înaintea codului propriu-zis al funcției. ENSURE reprezintă partea de contract ce se referă la obligațiile server-ului în cazul în care a fost apelat în termenii contractului, deci clauzele ENSURE vor apare la sfârșitul fiecărei metode, înainte de instrucțiunea „return". CHECK_INVARIANT este plasat

BUPT

de asemenea înaintea instrucțiunii „return" a fiecărei metode, dat fiind faptul că în urma execuției codului respectivei metode obiectul trebuie să se găsească într-o stare validă.

3.3.6.4 Testarea

Procesul de testare este legat de scenariul care se implementează. Procedurile de testare rezultă pe baza analizei thread-urilor, incluzând stimulii de intrare şi rezultatele aşteptate. Testele efectuate vor acoperi doar o parte din numărul total al căilor posibile prin toate modulele software. Nu este fezabilă verificarea tuturor căilor posibile.

Procedurile de testare trebuie scrise pentru toate interfețele sistemului cu utilizatorul. Testarea claselor individuale (*testul de componente*) este deosebit de importantă înainte de a le utiliza împreună în cadrul unei aplicații (*testul de integrare*), ducând la reducerea timpului global de testare.

Testarea softului, deşi este efectuată în etapa de programare, va fi planificată deja în etapa de proiectare. Procedurile de testare pot fi scrise imediat ce au fost stabilite scenariile în etapa de analiză.

Mecanismul de tratare a excepțiilor stabilit în etapa de proiectare va fi verificat în cadrul procesului de testare. În cadrul fiecărei noi funcții poate fi plasat cod care să controleze generarea de excepții. Acesta va fi eliminat din versiunea finală a produsului.

Activitatea de testarea devine dificilă atunci când sunt implicate ierarhiile de clase şi polimorfismul [JAC92, PER90]. O funcție membră a unei clase situate la un anumit nivel al ierarhiei de moştenire poate să se comporte diferit față de aceeaşi funcție în clasa de bază. Testarea funcțiilor membre doar în clasa de bază poate să facă imposibilă descoperirea unor probleme neaşteptate legate de polimorfism.

3.3.6.5 Depanarea

Activitatea de depanare este descrisă în [CLA93] ca fiind activitatea de direcționare a atenției unui programator spre o eroare astfel încât aceasta să poată fi remediată. Debugger-ul este în mod uzual un instrument care trebuie să însoțească procesul de depanare facilitând examinarea unei porțiuni staționare a programului aflat în rulare.

Activitatea de depanare reprezintă deci o secvență de paşi care includ testarea, inspectarea, şi remedierea segmentului de cod care conține o eroare şi restartarea după recompilare şi

BUPT

linkeditare. Faza de depanare are loc în cadrul procesului de testare şi este necesară pentru a descoperi un număr cât mai mare de erori software.

Utilizarea debuggerelor nu reprezintă o noutate pentru programarea în C, dar programele scrise în C++ prezintă o serie de facilităţi rezultate din caracteristicile noi ale limbajului şi care sunt dificil de identificat în faza de rulare, cum ar fi supraîncărcarea funcţiilor, crearea de obiecte de către constructori în ierarhii de moştenire şi de agregare respectiv valori şi obiecte temporare asociate tipurilor de date abstracte.

3.4 Concluzii. Contribuţii

Programarea orientată spre obiecte reprezintă soluţia de dezvoltare software ce oferă cel mai puternic instrument de modelare a realităţii. Implementarea este focalizată asupra claselor şi a obiectelor, entităţi ce încapsulează informaţie şi comportament, completând universul modelat cu un set complex de relaţii ce se stabilesc între aceste entităţi.

Metoda de dezvoltare software propusă în cadrul actualului capitol reprezintă o abordare orientată spre obiecte ce permite o continuare naturală a procesului de analiză cu cel de proiectare respectiv de implementare. Paradigmele orientării spre obiecte analizate în cadrul capitolului anterior sunt urmărite pe întreg parcursul acestui proces.

Contribuţiile prezente în cadrul capitolului actual sunt legate de dezvoltarea acestei metode de analiză şi proiectare software:

- *Abordarea procesului de proiectare prin rafinări succesive.* Se creează astfel posibilitatea de a urmări deciziile de proiectare din faza de analiză până în faza de implementare, asigurând flexibilitatea procesului de dezvoltare şi posibilitatea de reconfigurare cu efort minim.
- *Urmărirea conceptelor orientării spre obiecte pe parcursul întregului proces de dezvoltare software.* Astfel se urmăreşte evoluţia entităţilor utilizate în cadrul implementării aplicaţiei pornind de la clase şi obiecte ale etapei de analiză până la clase şi obiecte ale etapei de implementare într-un limbaj de programare orientat spre obiecte cum ar fi C++.
- *Completarea procesului de dezvoltare cu elemente specifice proiectării structurate.* Astfel se compensează lipsa unei abordări suficient de riguroase în cadrul metodelor consacrate de dezvoltare orientate spre obiecte. O atenţie deosebită în acest context a fost acordată soluţiilor distribuite şi aspectelor legate de operarea în timp real.

BUPT

- *Generarea unor entităţi reutilizabile.* Acestea satisfac conceptele orientării spre obiecte legate de abstractizarea datelor, încapsulare şi ascundere a informaţiei.
- *Evaluarea riscului* pe care îl implică abordarea orientată spre obiecte în cadrul fiecărei etape de dezvoltare ale aplicaţiei.
- *Unificarea tratării deciziilor de proiectare la nivel hardware şi software* pentru a obţine o abordare unitară în conformitate cu similitudinea existentă între programarea orientată spre obiecte şi structura modulară a sistemelor de calcul.
- *Urmărirea transpunerii corecte în cod a deciziilor de analiză şi proiectare* prin utilizarea unui mecanism puternic de tratare a excepţiilor ce implementează un mecanism bazat pe aserţiuni şi pe programarea bazată pe contract.
- *Aplicarea unor criterii de evaluare a claselor şi obiectelor* generate în fiecare etapă a procesului de dezvoltare software. Criteriile calitative sunt aplicate în etapele de analiză şi proiectare, calitatea procesului de implementare fiind evaluată prin intermediul unor metrici software.

O importanţă deosebită pentru stabilirea structurii metodei propuse în acest capitol a avut-o studiul bibliografic critic al unora dintre metodele existente de dezvoltare software orientate spre obiecte. Viabilitatea metodei de dezvoltare software propusă va fi demonstrată în cadrul capitolului următor, unde aceasta va fi aplicată în dezvoltarea unor componente din cadrul unui sistem de proiectare CAD/CAM.

BUPT

Soluție orientată spre obiecte aplicată în dezvoltarea componentelor unui sistem CAD/CAM

4.1 Introducere

Capitolul actual prezintă aplicarea în practică a metodei de dezvoltare software orientată spre obiecte propusă în cadrul capitolului anterior. Este prezentat procesul de dezvoltare a două componente software, *bibliotecă de ferestre* și *editor 2D*, pe care autorul le-a realizat în cadrul produsului CAD/CAM *DHP-Bauwerk* al firmei Dietrich's DHP AG.

Una dintre trăsăturile caracteristice unei aplicații CAD/CAM este dată de necesitatea existenței unui potențial considerabil de extensibilitate a setului de funcționalități. Un mare număr de astfel de funcționalități este de obicei definit după stabilirea și implementarea nucleului aplicației. Aplicația *DHP-Bauwerk*, abordată în cadrul prezentului capitol face parte din categoria sistemelor de proiectare și execuție a construcțiilor din lemn. Nucleul funcțional al acesteia este reprezentat de modulul *DICAM: Freie Konstruktion*, ce conține un modelator de solide, un modul de transformări geometrice 2D și 3D și de un sistem de reprezentare cu eliminarea liniilor și suprafețelor ascunse, prezentate în cadrul lucrării [MUR96]. Funcționalitățile specifice unui sistem CAD/CAM pentru prelucrarea lemnului adăugate acestui nucleu conferă aplicației o deosebită putere, aprecierea de care aceasta se bucură în rândul utilizatorilor fiind subliniată în diverse ocazii, dintre care amintim târgurile *Holzbau + Ausbau* Friedrichshafen, 04 – 07 mai 2000 și *Holz + Handwerk* Nürnberg, 23 – 26 martie 2000. O prezentare a elementelor de tehnologie orientată spre obiecte utilizate în dezvoltarea aplicației amintite a fost făcută de către autor în cadrul lucrărilor:

- *Abordare orientată spre obiecte pentru implementarea unui sistem integrat de proiectare a construcțiilor din lemn*, prezentată în cadrul Sesiunii de comunicări științifice „Zilele academice timișene, 1999 [MUR99]
- *Metode de generare automată a hașurilor în cadrul documentației de execuție generate de un sistem CAD/CAM.*, prezentată în cadrul Sesiunii de comunicări științifice „Zilele academice timișene, 1999 [MUR99a]
- *Object-oriented Development Process based upon Use Cases for a 2D-Editor in a CAD/CAM System,* prezentată în cadrul conferinței internaționale CONTI'2000: 4th

BUPT

International Conference on Technical Informatics, 12-13 oct. 2000, Timişoara, România [SAV2000]

- *Exception-based Assertion Mechanism for Object-oriented Systems used in the Implementation of a 2D-Editor for a CAD/CAM System*, prezentată în cadrul conferinței internaționale CONTI'2000: 4th International Conference on Technical Informatics, 12-13 oct. 2000, Timişoara, România [FAR2000]

Dimensiunea şi dinamica aplicației justifică utilizarea unui proces de dezvoltare software orientat spre obiecte. Metoda de dezvoltare orientată spre obiecte definită în cadrul capitolului anterior este utilizată în procesul de dezvoltare a acestei aplicații.

4.2 Descrierea sistemului de proiectare CAD/CAM

Aplicația *DHP-Bauwerk* se înscrie în familia produselor CAD/CAM şi reprezintă un mediu de proiectare şi prelucrare în domeniul construcțiilor în lemn. Referințe, descrieri şi aprecieri referitoare la această aplicație se regăsesc în [DIE99], [SCH2000], [QUA99], [BAU2000], [HOL2000].

Aplicația este formată din patru module ce pot funcționa atât împreună în cadrul oferit de aplicația *DHP-Bauwerk* cât şi individual:

- *DiWand*: modul pentru proiectarea şi prelucrarea pereților cu componentele:
  - *Grundriss*: subsistem pentru generarea şi prelucrarea pereților exteriori şi interiori în plan orizontal.
  - *Wandkonstruktion*: subsistem pentru prelucrarea componentelor unui perete individual.
- *Dachausmittlung*: modul pentru generarea şi prelucrarea de acoperişuri.
- *Deckenkonstruktion*: modul pentru generarea şi prelucrarea casetelor de plafon.
- *DICAM: Freie Konstruktion*: modul pentru generare şi prelucrare liberă de entități 3D ce conține:
  - *Modelator de solide*
  - *Modul de rendering*
  - *Modul pentru transformări geometrice 3D*
  - *Modul pentru gestiunea prelucrărilor speciale şi a îmbinărilor specifice construcțiilor în lemn*

BUPT

- *Modul de generare a documentaţiei de execuţie şi a comenzilor pentru maşini automate de prelucrare a lemnului*

Aplicaţia are rolul de a asista procesul de proiectare şi execuţie a unei construcţii din lemn parcurgând următorii paşi:

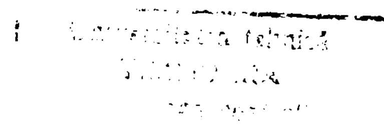
- Definirea în plan orizontal a conturului exterior al clădirii, a pereţilor exteriori şi interiori pentru fiecare etaj.
- Plasarea structurii de rezistenţă formate din elemente de construcţie din lemn şi a elementelor de izolaţie şi finisare în cadrul fiecărui perete.
- Definirea casetelor de plafon pentru fiecare etaj.
- Plasarea structurii de rezistenţă şi a elementelor de izolaţie şi de finisare în cadrul fiecărei casete de plafon.
- Definirea formei geometrice a acoperişului.
- Plasarea structurii de rezistenţă şi a elementelor de izolaţie şi de finisare în cadrul acoperişului
- Prelucrarea elementelor de construcţie din cadrul pereţilor, casetelor de plafon şi a acoperişului şi generarea de elemente de construcţie suplimentare în cadrul modulului *DICAM: Freie Konstruktion*.

Ordinea prelucrării nu este restricţionată la ordinea enumerării de mai sus. Pot avea loc iteraţii între diversele etape şi între cele patru domenii. Modificările efectuate asupra proiectului în oricare dintre domenii sunt vizibile în toate celelalte domeniile aplicaţiei *DHP-Bauwerk*.

4.3 Procesul de dezvoltare orientat spre obiecte

Procesul de dezvoltare al aplicaţiei *DHP-Bauwerk* este bazat pe analiza, proiectarea şi programarea orientată spre obiecte. Pe parcursul capitolului actual va fi urmărit procesul de dezvoltare a elementelor:

- *sistemul de ferestre*, organizat sub forma unei biblioteci de clase, denumit în continuare *bibliotecă de ferestre*.
- *editorul 2D*



BUPT

4.3.1 Locul şi rolul bibliotecii de ferestre şi a editorului 2D în cadrul aplicației

Atât biblioteca de ferestre cât şi editorul 2D sunt utilizate în cadrul fiecăruia dintre cele patru module ale aplicației *DHP-Bauwerk*. Rolul bibliotecii de ferestre este legat de:

- reprezentarea grafică a entităților geometrice 2D şi 3D în cadrul ferestrelor definite în modulele *DiWand*, *Dachausmittlung*, *Deckenkonstruktion* şi *DIĆAM: Freie Konstruktion* ale aplicației *DHP-Bauwerk*.
- transformări geometrice referitoare la reprezentarea grafică a entităților geometrice în ferestre (transformări proiective, transformări de coordonate etc.)

Rolul editorului 2D este legat de activitatea de generare şi de identificare de entități geometrice 2D (puncte, drepte, segmente de dreaptă, etc.) ca elemente auxiliare de construcție în cadrul celor patru module ale aplicației *DHP-Bauwerk*.

4.3.2 Procesul de dezvoltare software pentru biblioteca de ferestre şi editorul 2D

Procesul de dezvoltare software al celor două componente reprezintă aplicarea în practică a metodei de dezvoltare analizată în cadrul capitolului anterior. Parcursul urmat în cadrul procesului de dezvoltare reprezintă o versiune simplificată a schemei generale a procesului de dezvoltare software, prezentată în cadrul capitolului anterior. Paşii efectuați în cadrul procesului de dezvoltare a celor două componente sunt evidențiați prin chenar şi caractere îngroşate, după cum rezultă din figura următoare:

BUPT

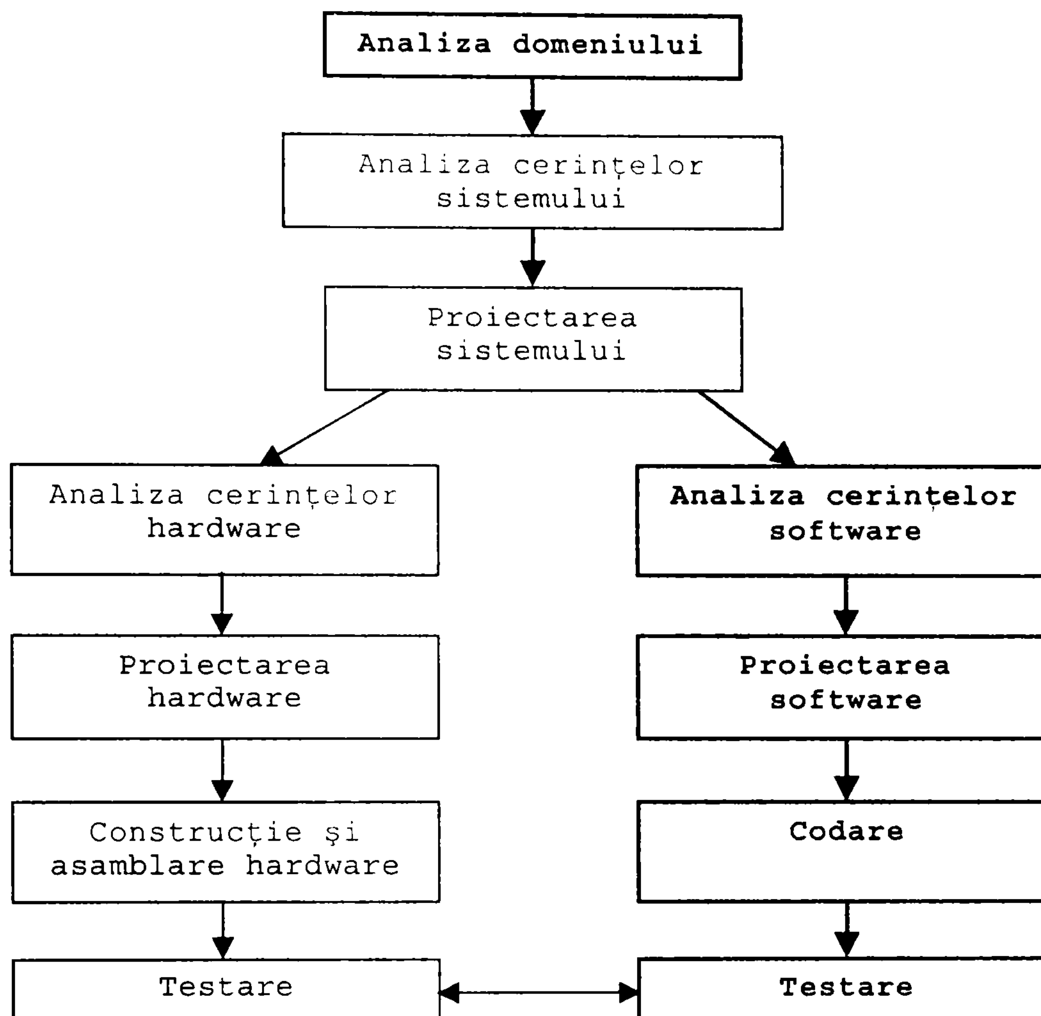


Figura 4.1 Pașii procesului de dezvoltare a aplicației

4.3.3 Sistemul de ferestre

Soluția de implementare aleasă pentru sistemul de ferestre a fost cea de bibliotecă de clase care să conțină instrumentele de bază pentru reprezentarea grafică a entităților geometrice 2D și 3D. Biblioteca conține o parte de gestiune a informației geometrice a ferestrelor precum și aparatul matematic necesar reprezentării și transformărilor efectuate asupra entităților geometrice 2D și 3D. Biblioteca nu include nici un fel de operații legate de desenare sau de lucrul cu dispozitivele de intrare, asigurându-se astfel independența față de aplicația client și de platforma utilizată.

Clasele din cadrul bibliotecii au fost gândite pentru moștenire. Aplicația care utilizează biblioteca va trebui să deriveze propriile clase din clasele prezente în cadrul bibliotecii pentru a adăuga elementele specifice legate de reprezentarea grafică și de interacțiunea cu dispozitivele de

BUPT

intrare. Astfel sunt asigurate dezideratele de încapsulare, generalitate şi reutilizabilitate în cadrul acestei biblioteci.

4.3.3.1 Analiza domeniului

Prima etapă a analizei domeniului constă în precizarea domeniului problemei. Domeniul general al problemei pentru biblioteca de ferestre este legat de vizualizarea entităţilor geometrice bidimensionale şi tridimensionale. Biblioteca este construită în jurul unui aparat matematic ce implementează un set de transformări necesare vizualizării entităţilor geometrice. Metoda de dezvoltare software descrisă în cadrul capitolului anterior se concentrează asupra următoarelor elemente în ceea ce priveşte analiza domeniului:

1. *Trăsăturile sistemului*

Biblioteca de ferestre reprezintă un server pentru aplicaţii dezvoltate într-un limbaj orientat spre obiecte. Ea se înscrie în clasa aplicaţiilor de grafică pe calculator.

2. *Cerinţele sistemului*

Serviciile pe care trebuie să le pună la dispoziţie biblioteca de ferestre sunt:

- *operaţii de gestiune a ferestrelor* în cadrul unui spaţiu de lucru definit
- *vizualizarea* în cadrul ferestrelor a entităţilor geometrice 2D şi 3D
- *transformări proiective* pentru trecerea din spaţiul 3D al modelului în spaţiul 2D al reprezentării pe ecran.
- *transformări geometrice 2D* destinate prelucrării imaginii reprezentate în cadrul ferestrelor (decupare, zoom, defilare etc.)
- *repere auxiliare* pentru entităţile geometrice reprezentate în ferestre: sisteme de coordonate libere şi rastru ortogonal

3. *Arhitectura sistemului*

Biblioteca reprezintă o aplicaţie de dimensiuni restrânse în cadrul căreia predomină componenta software. Ea va fi compilată în cadrul unui mediu de dezvoltare orientat spre obiecte bazat pe limbajul C++. Biblioteca va fi independentă de platformă, astfel că ea va putea fi utilizată în cadrul oricărui sistem de operare care pune la dispoziţie un mediu de dezvoltare pentru un limbaj de programare orientat spre obiecte.

BUPT

4. *Clase şi obiecte ale lumii reale*

Determinarea abstractizărilor esențiale ce vor constitui clase şi obiecte ale lumii reale va fi efectuată pe baza specificației funcționale a sistemului.

Specificația funcțională a sistemului

Specificația funcțională a sistemului defineşte *entitățile de bază* care compun biblioteca de ferestre, *relațiile* dintre acestea şi *serviciile* puse la dispoziție aplicațiilor ce utilizează biblioteca. Entitățile geometrice tridimensionale reprezentate în spațiul $2D_{discret}$ (măsurat în pixeli) al ferestrelor sunt definite într-un spațiu $3D_{continuu}$ (măsurat în metri) iar cele bidimensionale în cadrul unui spațiu $2D_{continuu}$ (măsurat în metri).

- *Dispozitivul de afişare*: reprezintă spațiul discret în care va avea loc reprezentarea imaginii cu ajutorul primitivelor grafice. În cazul aplicațiilor care gestionează întregul ecran în mod grafic (full screen), dispozitivul de afişare este reprezentat de întregul ecran. Sistemele de operare care rulează o aplicație într-o fereastră pe ecran (Windows 9x, Windows NT, Linux, etc.), dispozitivul de afişare va fi considerat a fi zona client (*client area*) [HOR98] a ferestrei în care rulează aplicația.

- *Spațiul de lucru*: reprezintă o zonă dreptunghiulară în cadrul dispozitivului de afişare cu rol de container pentru ferestre. Spațiul de lucru se defineşte în coordonatele dispozitivului de lucru. Spațiul de lucru conține o listă de ferestre.

- *Fereastra 2D*: reprezintă o zonă dreptunghiulară în cadrul spațiului de lucru, utilizată pentru reprezentarea prin intermediul primitivelor grafice a modelului 2D gestionat în cadrul aplicației ce utilizează biblioteca de ferestre. Fereastra se defineşte în sistemul de coordonate al spațiului de lucru.

- *Fereastra 3D*: reprezintă o zonă dreptunghiulară în cadrul spațiului de lucru, utilizată pentru reprezentarea prin intermediul primitivelor grafice a modelului 3D gestionat în cadrul aplicației ce utilizează biblioteca de ferestre. Fereastra se defineşte în sistemul de coordonate al spațiului de lucru.

- *Sistemul de coordonate liber 2D*: reprezintă un reper cartezian în spațiul $2D_{continuu}$ în cadrul căruia sunt definite entitățile geometrice ce se reprezintă în cadrul ferestrei.

BUPT

- *Sistemul de coordonate liber 3D*: reprezintă un reper cartezian în spațiul $3D_{continuu}$ în cadrul căruia sunt definite entitățile geometrice ce se reprezintă în cadrul ferestrei.

- *Rastrul*: reprezintă o rețea ortogonală de puncte echidistante pe cele două direcții principale în spațiul $2D_{continuu}$, definită în cadrul sistemului de coordonate 2D activ al ferestrei.

Din punct de vedere funcțional biblioteca trebuie să pună la dispoziția aplicației client un set de clase care să implementeze următoarele operații:

- controlul parametrilor geometrici ce definesc spațiul de lucru
- definirea și gestiunea informațiilor geometrice pentru un număr oarecare de ferestre în cadrul unui spațiu de lucru în formă dreptunghiulară definit în cadrul unui dispozitiv de afișare
- definirea și gestiunea unui rastru ortogonal în spațiul $2D_{continuu}$
- definirea și gestiunea unui număr oarecare de sisteme de coordonate libere în spațiile $3D_{continuu}$ și $2D_{continuu}$
- transformări proiective din spațiul $3D_{continuu}$ în spațiul $2D_{continuu}$
- transformări de coordonate din spațiul $2D_{continuu}$ în spațiul $2D_{discret}$ al ferestrei pentru maparea spațiului $2D_{continuu}$ la spațiul ferestrei
- operații de decupare în spațiul $2D_{continuu}$ pentru încadrarea imaginii în fereastră
- operații de zoom a imaginii în spațiul $2D_{discret}$
- operații de defilare a imaginii în spațiul $2D_{discret}$

Figura următoare ilustrează entitățile de bază prezente în cadrul bibliotecii de ferestre:

BUPT

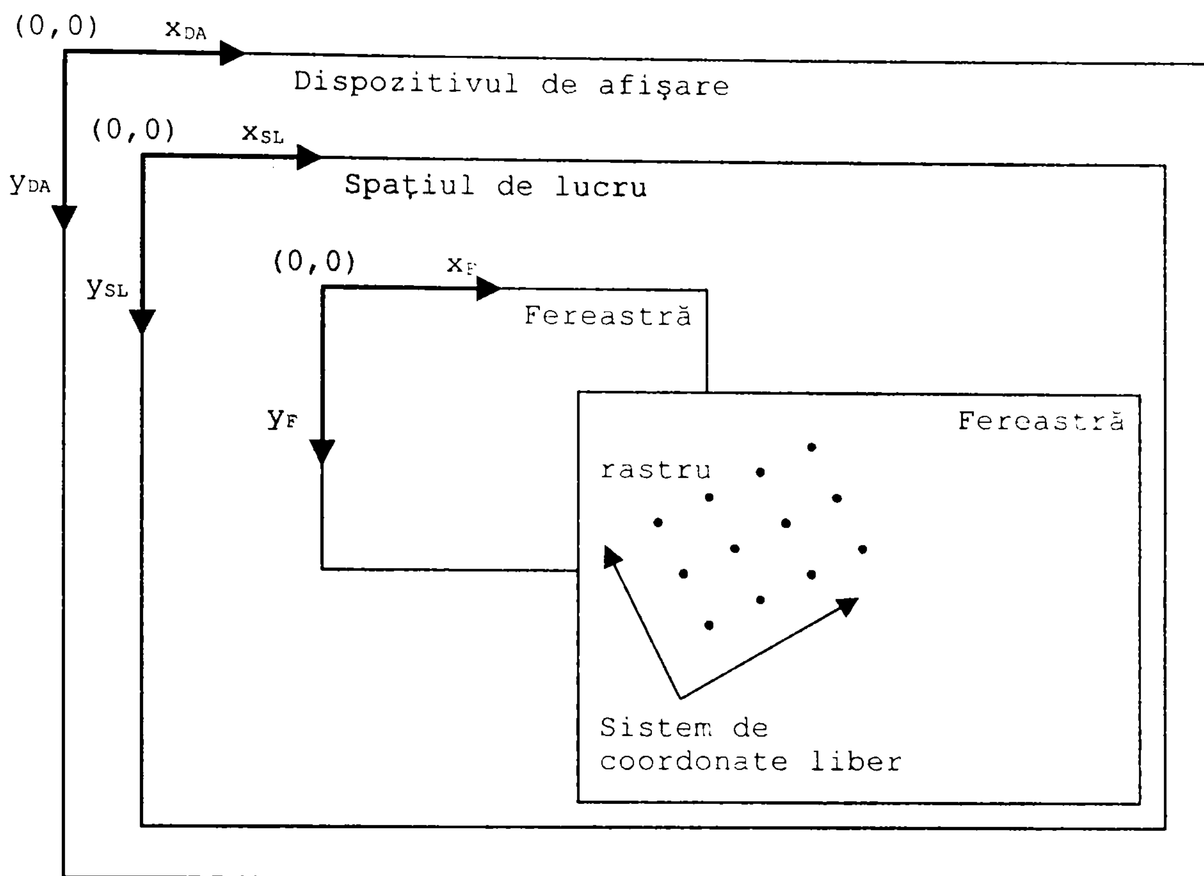


Figura 4.2 Definirea spațiului de lucru și a ferestrei

Ca produs de lucru al analizei domeniului rezultă următoarele fișe pentru obiectele lumii reale identificate:

Identificator obiect: `Display`		**Cod sursă:**	
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Dispozitivul de afişare	Obiectul realizează interfața între spațiul de lucru și dispozitivul fizic de afişare.
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs nou dezvoltat	
5.	**Clasificarea obiectului**	Clasă interfață dispozitiv	
6.	**Atribute**	- dimensiunile zonei de afişare	
7.	**Operații**	- metode de acces la atribute	

BUPT

Identificator obiect: `WorkSpace`		**Cod sursă:**	
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Spațiul de lucru	- definit în cadrul dispozitivului de afişare - reprezintă spațiul suport pentru un număr oarecare de ferestre
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs nou dezvoltat	
5.	**Clasificarea obiectului**	Clasă computațională	
6.	**Atribute**	- informații geometrice referitoare la poziție şi dimensiune - date interne auxiliare	datele interne auxiliare sunt utilizate în cadrul nucleului matematic necesar transformărilor geometrice implementate la nivelul ferestrelor 2D şi 3D
7.	**Operații**	- accesul la atributele ce reprezintă informațiile geometrice	

Identificator obiect: `Window2D`		**Cod sursă:**	
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Fereastra 2D	- definită în cadrul spațiului de lucru
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs nou dezvoltat	
5.	**Clasificarea obiectului**	Clasă computațională	
6.	**Atribute**	- informații geometrice legate de poziție în spațiul de lucru şi dimensiuni - lista de sisteme de coordonate libere 2D asociate - rastrul	
7.	**Operații**	- calcul parametri pentru: o maparea $2D_{continuu} \rightarrow 2D_{discret}$ o zoom o defilare	

BUPT

Identificator obiect: `Window3D`			**Cod sursă:**
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Fereastra 3D	- definită în cadrul spațiului de lucru
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs nou dezvoltat	
5.	**Clasificarea obiectului**	Clasă computațională	
6.	**Atribute**	- informații geometrice legate de poziție în spațiul de lucru și dimensiuni - lista de sisteme de coordonate libere 3D asociate	
7.	**Operații**	- calcul parametri pentru: o proiecția $3D_{continuu} \rightarrow 2D_{continuu}$ o maparea $2D_{continuu} \rightarrow 2D_{discret}$ o zoom o defilare	

Identificator obiect: `FreeCoordSys2D`			**Cod sursă:**
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Sistem de coordonate liber 2D (SCL 2D)	- poate deveni sistem de coordonate activ, caz în care toate coordonatele din fereastră se exprimă relativ la acesta
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs reutilizabil intern	- specificația funcțională definită în cadrul produsului *DICAM*.
5.	**Clasificarea obiectului**	Clasă computațională	
6.	**Atribute**	- parametrii ce definesc SCL	- SCL este definit prin următoarele elemente în cadrul sistemului de coordonate absolut:

BUPT

			originea (P_0), un punct pe axa X (P_x), un punct pe axa Y (P_y). Vectorii P_0P_x respectiv P_0P_y sunt versori.
7.	**Operații**	- setare parametri SCL - citire parametri SCL - transformări de coordonate $SC_{global}\leftrightarrow$ SCL 2D	

Identificator obiect: `FreeCoordSys3D`		**Cod sursă:**	
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Sistem de coordonate liber 3D (SCL 3D)	- poate deveni sistem de coordonate activ, caz în care toate coordonatele din fereastră se exprimă relativ la acesta
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs reutilizabil intern	- specificația funcțională este preluată din cadrul produsului DICAM. - Se va adapta la cerințele tehnologiei orientate spre obiecte
5.	**Clasificarea obiectului**	Clasă computațională	
6.	**Atribute**	- parametrii ce definesc SCL	- SCL este definit prin următoarele elemente în cadrul sistemului de coordonate absolut: originea (P_0), un punct pe axa X (P_x), un punct pe axa Y (P_y) şi un punct pe axa Z.(P_z), P_0P_x, P_0P_y respectiv P_0P_z sunt versori
7.	**Operații**	- setare parametri SCL 3D - citire parametri SCL 3D - transformări de coordonate $SC_{global}\leftrightarrow$ SCL 3D	

BUPT

Identificator obiect: `Raster`		**Cod sursă:**	
Element revizuit		**Valoare**	**Observații**
1.	**Entitate a lumii reale**	Rastrul	
2.	**Completitudine**		
3.	**Dimensiune**		
4.	**Categoria de dezvoltare**	Produs nou dezvoltat	O abordare pentru diverse cazuri particulare poate fi regăsită în cadrul produsului DICAM
5.	**Clasificarea obiectului**	Clasă computațională	
6.	**Atribute**	- pasul pe orizontală SCL activ - pasul pe verticala SCL activ - referință spre sistemul de coordonate 2D în care este definit	- deși rastrul va opera în pixeli în cadrul ferestrei, va fi definit în spațiul $2D_{continuu}$ pentru a oferi precizia dorită și pentru că este definit în SCL activ (care poate avea axele neparalele cu laturile ferestrei)
7.	**Operații**	- setarea parametrilor - citirea parametrilor - returnarea punctului de rastru cel mai apropiat de un punct dat	

4.3.3.2 Analiza cerințelor sistemului

Capitolul anterior descrie etapa de analiză a cerințelor sistemului ca moment al structurării sistemului independent de mediul de implementare. Este identificat un set de obiecte independente ce formează o structură ideală logică și stabilă.

Abordarea propusă pentru analiza cerințelor sistemului în cadrul metodei de dezvoltare software prezentată în cadrul capitolului anterior se bazează pe scenarii. O definiție detaliată a termenului de scenariu precum și importanța scenariilor în procesul de identificare a obiectelor au fost expuse pe larg în capitolul anterior.

Biblioteca de ferestre nu reprezintă o aplicație de sine stătătoare ci un server de clase pentru aplicația client care realizează reprezentarea grafică a entităților 2D și 3D. Nu există interacțiune directă între componentele bibliotecii de ferestre și utilizator. Abordarea orientată

BUPT

spre scenarii nu este potrivită pentru un astfel de tip de aplicație [NIE95], astfel că etapa de analiză a cerințelor sistemului va fi omisă în actualul proces de dezvoltare.

4.3.3.3 Proiectarea sistemului

Metoda de dezvoltare software utilizată leagă etapa de proiectare a sistemului de acțiunea de alocare a cerințelor sistemului componentelor hardware şi software, având ca etape de bază partiționarea şi configurarea. Fiind o etapă specifică dezvoltării sistemelor mari, proiectarea sistemului nu va fi parte a procesului de analiză orientată spre obiecte pentru biblioteca de ferestre.

4.3.3.4 Analiza cerințelor software

În cazul bibliotecii de ferestre, sistem exclusiv software, etapa de analiză a cerințelor urmează direct după analiza domeniului (vezi figura 4.1). În acest context, analiza cerințelor software are ca scop definirea de clase şi obiecte abstracte, ce vor reprezenta baza pentru tranziția de la etapa de analiză la etapa de proiectare.

Modelul informațional care stă la baza analizei cerințelor software se bazează pe următoarea diagramă de relații între entități:

BUPT

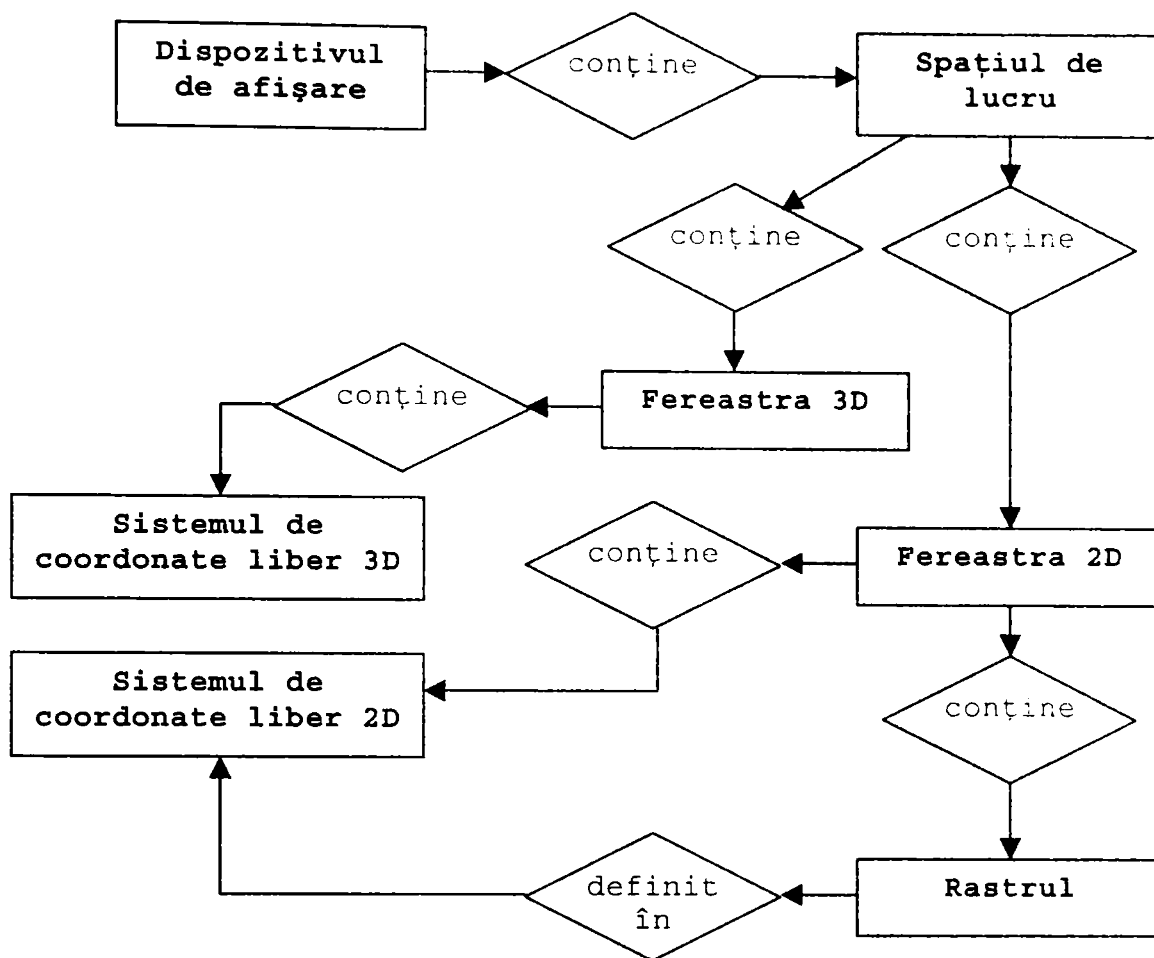


Figura 4.3 Diagrama de relații între entități pentru biblioteca de ferestre

Cerințele software pentru biblioteca de ferestre completează specificația funcțională a sistemului cu următoarele elemente:

1. Funcționalitățile implementate în cadrul bibliotecii de ferestre reprezintă un set independent de sistemul de gestiune a primitivelor grafice utilizate pentru reprezentarea pe ecran. Acestea cuprind doar operațiile matematice necesare manipulării primitivelor geometrice 2D și 3D în vederea reprezentării acestora în ferestre. Operațiile de desenare la nivelul dispozitivului de afişare vor fi implementate la nivelul aplicației client.
2. Clasele din cadrul bibliotecii de ferestre vor permite derivarea de specializări în cadrul aplicației client în vederea implementării operațiilor și trăsăturilor specifice aplicației client.

Clasele și obiectele lumii reale rezultă din diagrama de relații între entități (figura 4.3) și vor fi determinate pe baza modelului de clasificare funcțională prezentat în cadrul capitolului anterior. Perspectiva client-server pe care o urmărește actuala dezvoltare nu este adecvată

BUPT

clasificării bazate pe scenarii (cum ar fi cazul aplicaţiilor care pun la dispoziţie o interfaţă pentru utilizatorul unui sistem extern) ce clasifică obiectele în interfaţă, control şi entitate.

Categoriile în care sunt încadrate obiectele principale ce vor fi reprezentate ca şi abstractizări cheie sunt:

- Abstractizări *dispozitiv de afişare*
- Abstractizări *fereastră*
- Abstractizări *instrumente auxiliare* în cadrul ferestrei

Clasele identificate ca făcând parte din aceste categorii sunt enumerate în continuare. Identificarea atributelor şi a operaţiilor are loc conform definiţiilor prezentate în cadrul capitolului anterior: *atributele*, datele asociate unei clase, includ valorile de stare ale instanţelor acesteia precum şi datele constante ale acestora; *operaţiile* asociate unei clase reprezintă serviciile prin intermediul cărora pot fi modificate valorile atributelor. Rezultatul operaţiei de identificare a atributelor şi operaţiilor conduce la următoarele rezultate:

Din categoria abstractizărilor *dispozitiv de afişare* fac parte:

- *Dispozitivul de afişare*: caracterizează parametrii zonei disponibile pentru afişarea spaţiului de lucru şi a ferestrelor.

Display
rezolutiaX rezolutiaY
getRezolutia setRezolutia

- *Spaţiul de lucru*: reprezintă zona suport pentru ferestre. Acesta conţine o listă de ferestre

WorkSpace
workspaceData windowList internalData
getWorkspaceData setWorkspaceData getInternalData accesWindowList

BUPT

Din categoria abstractizărilor *fereastră* fac parte:

- *Fereastra 2D*: zonă dreptunghiulară în cadrul spațiului de lucru destinată reprezentării entităților geometrice definite în spațiul $2D_{continuu}$

Window2D
windowData Workspace internalData listofFCS2D activeFCS2D raster
getWindowData setWindowData getInternalData calcParameters zoom clip convertPixelToMetric convertMetricToPixel

- *Fereastra 3D*: zonă dreptunghiulară în cadrul spațiului de lucru destinată reprezentării entităților geometrice definite în spațiul $3D_{continuu}$

Window3D
windowData Workspace internalData viewpointData list_of_FCS3D active_FCS3D
getWindowData setWindowData getInternalData calcParameters zoom clip convertPixelToMetric convertMetricToPixel

Din categoria abstractizărilor *instrumente auxiliare* în cadrul ferestrei fac parte:

- *Sistemul de coordonate liber 2D*: reper cartezian definit în cadrul spațiului $2D_{continuu}$

BUPT

FreeCoordSys2D
Name Origin pointX pointY
getName setName getPoints setPoints coordTransformFCS_GCS coordTransformGCS_FCS

- *Sistemul de coordonate liber 3D*: reper cartezian definit în cadrul spaţiului $3D_{continuu}$

FreeCoordSys3D
Name Origin pointX pointY pointZ
setFCS getFCS transformCoordFCS_GCS transformCoordGCS_FCS

- *Rastrul*: reţea ortogonală de puncte echidistante având un pas constant pe direcţia orizontală şi alt pas constant pe direcţia verticală

Raster
Window stepX stepY ctrPoint info
SetStep getStep setParent setCrtPoint getCrtPOint setInfo getInfo attractToGrid

BUPT

Faza de identificare a claselor, a atributelor și a operațiilor acestora fiind încheiată, următorul pas în cadrul analizei cerințelor software îl reprezintă generarea modelului obiectelor pentru biblioteca de ferestre:

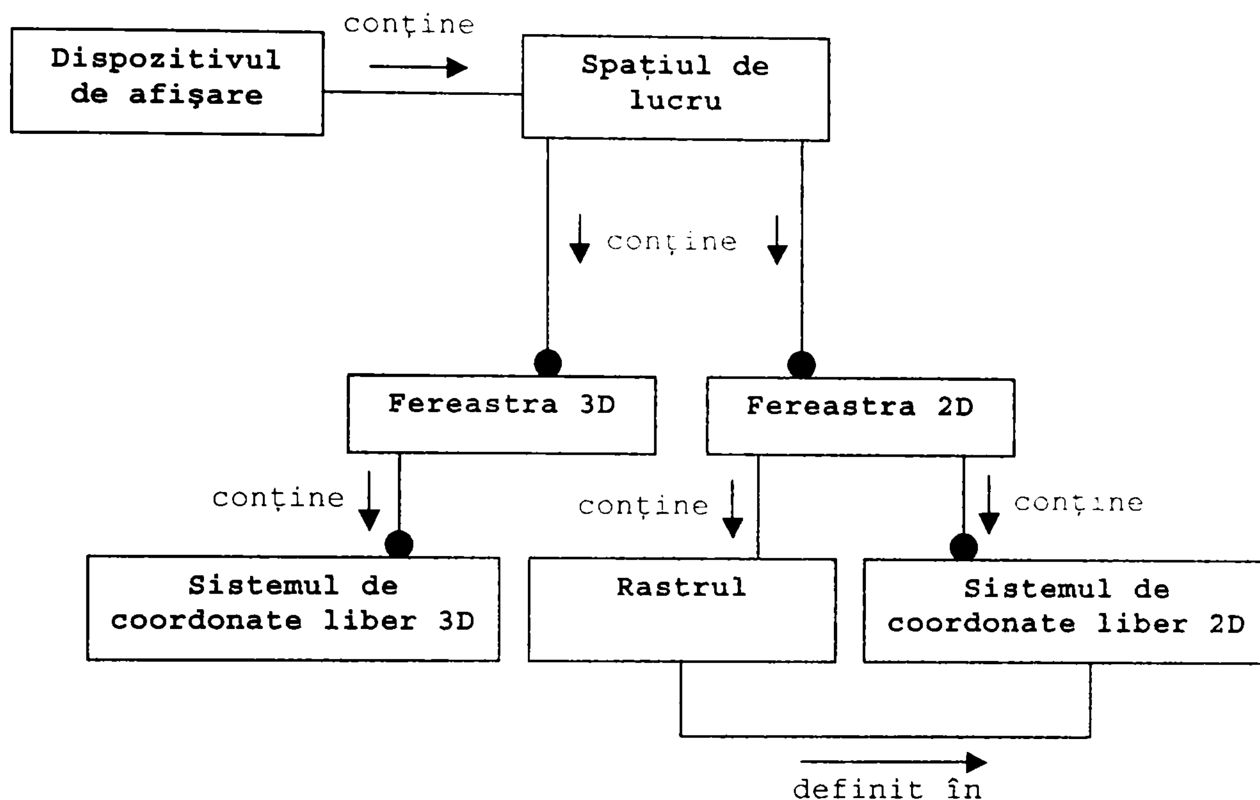


Figura 4.4 Modelul obiectelor pentru biblioteca de ferestre

În cadrul ilustrației de mai sus s-a utilizat pentru cardinalitate notația specifică metodei OMT, având semnificație de „unu la zero sau mai mulți".

Evaluarea claselor și a obiectelor în actuala etapă reprezintă o completare a evaluării efectuate în etapa de analiză a domeniului. Următoarele fișe de control sunt asociate obiectelor identificate până în prezent și reprezentate în cadrul modelului obiectelor.

Identificator obiect: `Display`		**Cod sursă:**	
Element revizuit		**Valoare**	**Observații**
1.	**Categoria obiectului**	Dispozitiv de afişare	
2.	**Completitudine**	obiectul modelează acele trăsăturilor ale dispozitivului de afişare care sunt importante pentru gestiunea spațiului de lucru.	Implementarea setului complet de facilități ale dispozitivului de afişare se realizează în cadrul aplicației client.
3.	**Dimensiune**		

BUPT

4.	**Atribute**	`rezoluţiaX` `rezoluţiaY`	
5.	**Operaţii**	`getRezolutia` `setRezolutia`	
6.	**Identificator scenariu**	–	analiza efectuată nu se bazează pe scenarii
7.	**Nivel de detaliu**		
8.	**Volatilitate**	obiect tranzient	
9.	**Dependenţa de implementare**		
	hardware	independent de platformă	
	sistem de operare	independent de sistemul de operare	
	DBMS	–	
10.	**Subsistem candidat**	–	Biblioteca de ferestre
11.	**Agregare**		
12.	**Asociere**		

Identificator obiect: `WorkSpace`		**Cod sursă:** `CDhpWorkspace`	
Element revizuit		**Valoare**	**Observaţii**
1.	**Categoria obiectului**	Dispozitiv de afişare	
2.	**Completitudine**	Modelează în totalitate facilităţile necesare unui container pentru ferestre	
3.	**Dimensiune**		

4.	**Atribute**	`workspaceData`
		`windowList`
		`internalData`
			- `workspaceData` reprezintă o structură ce conţine parametrii geometrici ai spaţiului de lucru care pot fi modificaţi în cadrul aplicaţiei client
			- `windowList` reprezintă lista de ferestre din cadrul spaţiului de lucru
			- `internalData` reprezintă date interne necesare nucleului matematic pentru transformări geometrice

BUPT

5.	**Operații**	`getWorkspaceData`
		`setWorkspaceData`
		`getInternalData`
		`accesWindowList`
			- `getWorkspaceData` și `setWorkspaceData` facilitează accesul la structura workspaceData
			- `getInternalData` accesează în citire datele interne
			- `accesWindowList` reprezintă un set de metode de acces la lista de ferestre

6.	**Identificator scenariu**	–	analiza efectuată nu se bazează pe scenarii
7.	**Nivel de detaliu**		
8.	**Volatilitate**	obiect tranzient	
9.	**Dependența de implementare**		
	hardware	independent de platformă	
	sistem de operare	independent de sistemul de operare	
	DBMS	–	
10.	**Subsistem candidat**	–	Biblioteca de ferestre
11.	**Agregare**	agregat în cadrul dispozitivului de afișare	
12.	**Asociere**		

Identificator obiect: `Window2D` `Window3D`		**Cod sursă:** `CDhpWindow`	
Element revizuit		**Valoare**	**Observații**
1.	**Categoria obiectului**	Fereastră	
2.	**Completitudine**	Include totalitatea facilităților necesare vizualizării unei scene formate din obiecte 3D, mai puțin funcțiile de desenare a primitivelor grafice în spațiul $2D_{discret}$	
3.	**Dimensiune**		

4.	**Atribute**	`WindowData`
		`WorkSpace`
		`internalData`
		`viewpointData`
			- `WindowData` reprezintă o structură ce conține parametrii geometrici ai ferestrei care pot fi modificați

BUPT

			în cadrul aplicaţiei client - `WorkSpace` reprezintă spaţiul de lucru care conţine fereastra - `internalData` reprezintă date interne necesare nucleului matematic pentru transformări geometrice - `viewpointData` reprezintă parametrii geometrici ai punctului de observaţie pentru fereastra 3D.

5.	**Operaţii**	`getWindowData`
		`setWindowData`
		`getInternalData`
		`calcParameters`
		`zoom`
		`clip`
		`convertPixelToMetric`
		`convertMetricToPixel`
			- `getWindowData` şi `setWindowData` facilitează accesul la structura WindowData
			- `getInternalData` accesează în citire datele interne
			- `calcParameters` realizează transformarea $2D_{continuu} \rightarrow 2D_{discret}$ şi calculează parametrii necesari reprezentării imaginii în fereastră
			- `zoom` calculează parametrii necesari reprezentării în fereastră a unei zone dreptunghiulare din cadrul imaginii
			- `clip` reprezintă implementarea unui algoritm de decupare bidimensională ce se aplică tuturor segmentelor de dreaptă ce se desenează în cadrul ferestrei
			- `convertPixelToMetric` şi `convertMetricToPixel` sunt funcţii de conversie a coordonatelor unui punct între spaţiile $2D_{discret}$ şi $2D_{continuu}$

6.	**Identificator scenariu**	–	analiza efectuată nu se bazează pe scenarii
7.	**Nivel de detaliu**		
8.	**Volatilitate**	Obiect tranzient	

BUPT

9.	**Dependenţa de implementare**		
	hardware	independent de platformă	
	sistem de operare	independent de sistemul de operare	
	DBMS	–	
10.	**Subsistem candidat**	–	Biblioteca de ferestre
11.	**Agregare**	- agregate în cadrul spaţiului de lucru - `Window2D` conţine o listă de obiecte `FreeCoordSys2D` şi obiectul `Rastru` - `Window3D` conţine o listă de obiecte `FreeCoordSys3D`	
12.	**Asociere**		

Identificator obiect: `FreeCoordSys2D` `FreeCoordSys3D`		**Cod sursă:** `CDhpFCS`	
Element revizuit		**Valoare**	**Observaţii**
1.	**Categoria obiectului**	Instrument auxiliar în cadrul ferestrei	
2.	**Completitudine**	Include totalitatea facilităţilor necesare definirii unui sistem de coordonate liber şi a gestiunii informaţiei geometrice în cadrul acestuia	
3.	**Dimensiune**		

4.	**Atribute**	`Name`
		`Origin`
		`pointX`
		`pointY`
		`pointZ`
			- `Name` reprezintă numele sistemului de coordonate (necesar identificării acestuia în cadrul listei de sisteme de coordonate din cadrul ferestrei)
			- `Origin`, `pointX` şi `pointY` reprezintă originea, un punct pe axa OX respectiv un punct pe axa OY a sistemului de coordonate
			- `pointZ` reprezintă un punct pe axa OZ în cazul sistemului de coordonate 3D. Nu este utilizat în

BUPT

			cadrul sistemului de coordonate 2D
5.	**Operații**	getName setName getPoints setPoints coordTransformFCS_GCS coordTransformGCS_FCS	- getName și setName sunt metode de acces la numele sistemului de coordonate - getPoints și setPoints sunt metode de acces la punctele ce definesc sistemul de coordonate
6.	**Identificator scenariu**	–	analiza efectuată nu se bazează pe scenarii
7.	**Nivel de detaliu**		
8.	**Volatilitate**	Obiect tranzient	
9.	**Dependența de implementare**		
	hardware	independent de platformă	
	sistem de operare	independent de sistemul de operare	
	DBMS	–	
10.	**Subsistem candidat**	–	Biblioteca de ferestre
11.	**Agregare**	- Sistemul de coordonate 2D este agregat în cadrul ferestrei 2D - Sistemul de coordonate 3D este agregat în cadrul ferestrei 3D	
12.	**Asociere**	- în cadrul sistemului de coordonate 2D se definește rastrul	

Identificator obiect: Rastrul		**Cod sursă:** CdhpGrid	
Element revizuit		**Valoare**	**Observații**
1.	**Categoria obiectului**	Instrument auxiliar în cadrul ferestrei	
2.	**Completitudine**	Include totalitatea informațiilor necesare implementării și gestiunii unei rețele ortogonale de puncte auxiliare de atracție	
3.	**Dimensiune**		
4.	**Atribute**	Window	- rastrul este definit în cadrul sistemului de

BUPT

		stepX stepY ctrPoint info	cadrul sistemului de coordonate activ al ferestrei Window - stepX şi stepY reprezintă pasul pe axa OX respectiv pasul pe axa OY a rastrului. - crtPoint reprezintă ultimul punct de atracţie furnizat - info reprezintă informaţiile de culoare, stare etc. ale rastrului
5.	**Operaţii**	setStep getStep setParent setCrtPoint getCrtPOint setInfo getInfo attractToGrid	- setStep şi getStep realizează accesul la parametrii geometrici ai rastrului - setParent setează adresa ferestrei în care este reprezentat rastrul - setCrtPoint şi getCrtPoint facilitează accesul la crtPoint - setInfo şi getInfo facilitează accesul la atributele info ale rastrului - attractToGrid furnizează cel mai apropiat punct de atracţie de un punct dat
6.	**Identificator scenariu**	–	analiza efectuată nu se bazează pe scenarii
7.	**Nivel de detaliu**		
8.	**Volatilitate**	Obiect tranzient	
9.	**Dependenţa de implementare**		
	hardware	independent de platformă	
	Sistem de operare	independent de sistemul de operare	
	DBMS	–	
10.	**Subsistem candidat**	–	Biblioteca de ferestre
11.	**Agregare**	- în cadrul ferestrei 2D	
12.	**Asociere**	- rastrul se defineşte faţă de sistemul de	

BUPT

		coordonate activ al ferestrei în cadrul căreia este agregat.	

4.3.3.5 Proiectarea software

Pasul următor în cadrul procesului de dezvoltare al aplicaţiei îl constituie proiectarea software. Paşii parcurşi în etapa de proiectare software sunt descrişi în cadrul capitolului anterior:

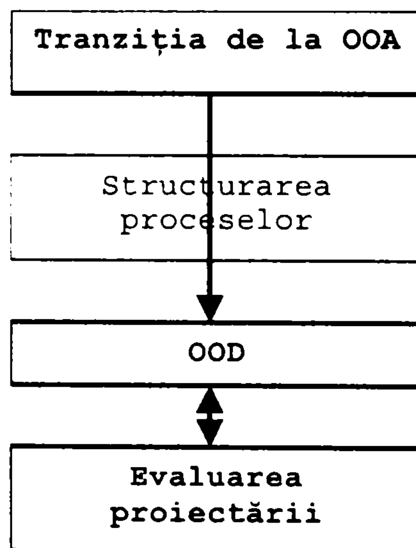


Figura 4.5 Subpaşii fazei de proiectare software

Subpasul de structurare a proceselor nu face parte din etapele parcurse în cadrul prezentului exemplu de aplicare a metodei de dezvoltare software, fapt care rezultă din figura de mai sus.

Tranziţia de la etapa OOA se ghidează după următoarele criterii [NIE95]:

- clase abstracte de analiză → clase C++
- atribute → date membre ale clasei C++ corespunzătoare
- operaţii → funcţii membre ale clasei C++ corespunzătoare
- ierarhii de clase → clase C++, utilizând relaţii de moştenire şi agregare
- relaţii între clase → utilizare, implementată prin referirea altor obiecte din cadrul clasei care au nevoie de acestea
- diagrame de tranziţie a stărilor → fiecare diagramă devine o clasă de bază cu o clasă derivată pentru fiecare stare [FAI93]

BUPT

- scenarii şi obiecte entitate → obiectele abstracte din cadrul fiecărui scenariu devin obiecte entitate
- modelul de flux al obiectelor → fiecare obiect din cadrul modelului de flux al obiectelor devine un subsistem potenţial, reprezentând punctul de plecare pentru proiectarea sistemului
- diagrama de trasare a evenimentelor → evenimentele devin mesaje transmise între obiecte (subsisteme)

Proiectarea software reprezintă momentul optim de abordare a moştenirii în ceea ce priveşte structura de clase. Modelul de moştenire şi agregare generat pentru biblioteca de ferestre este prezentat în continuare:

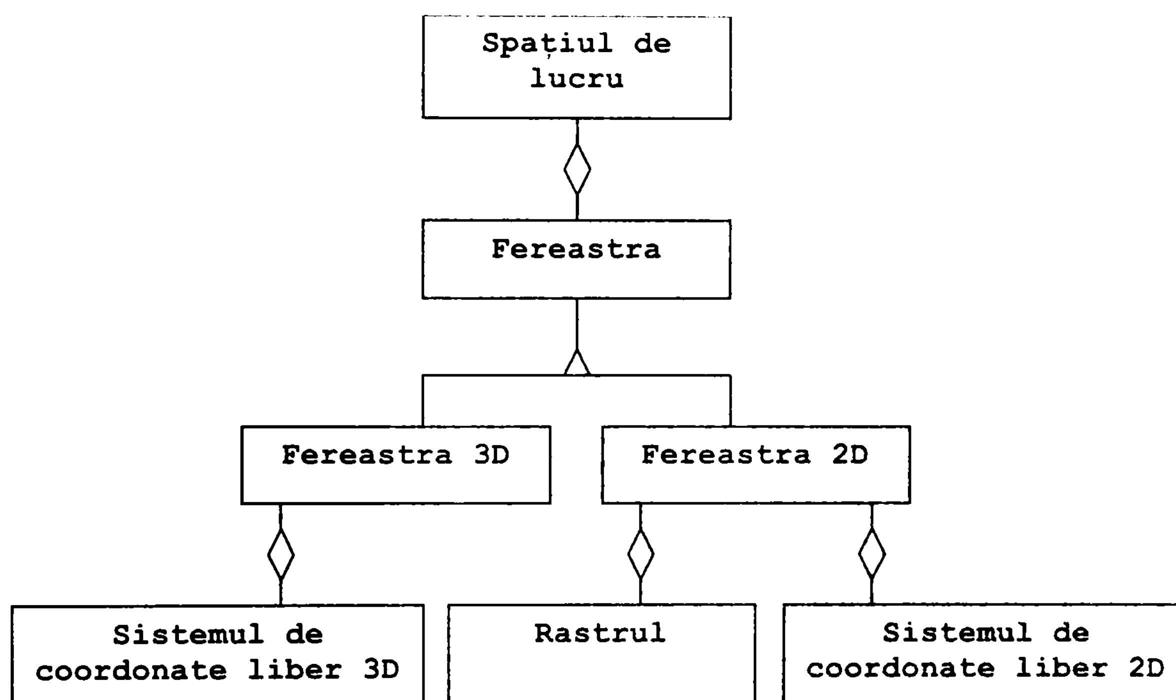
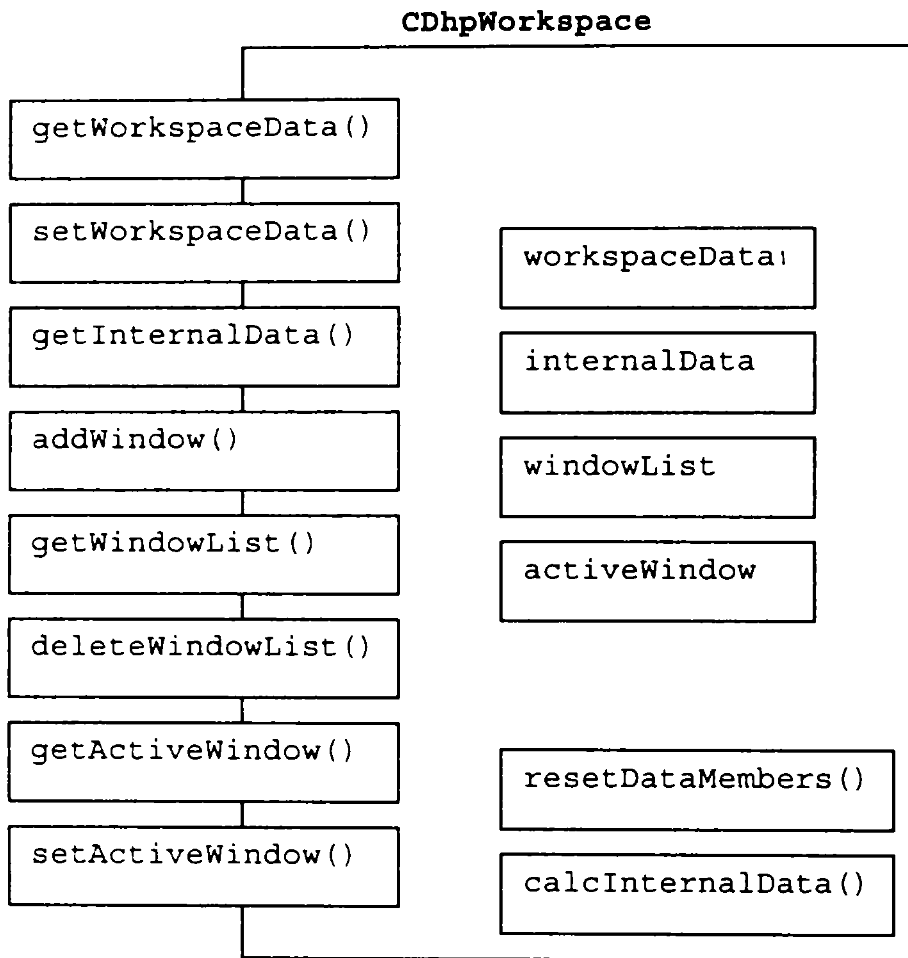


Figura 4.6 Modelul de agregare şi moştenire pentru biblioteca de ferestre

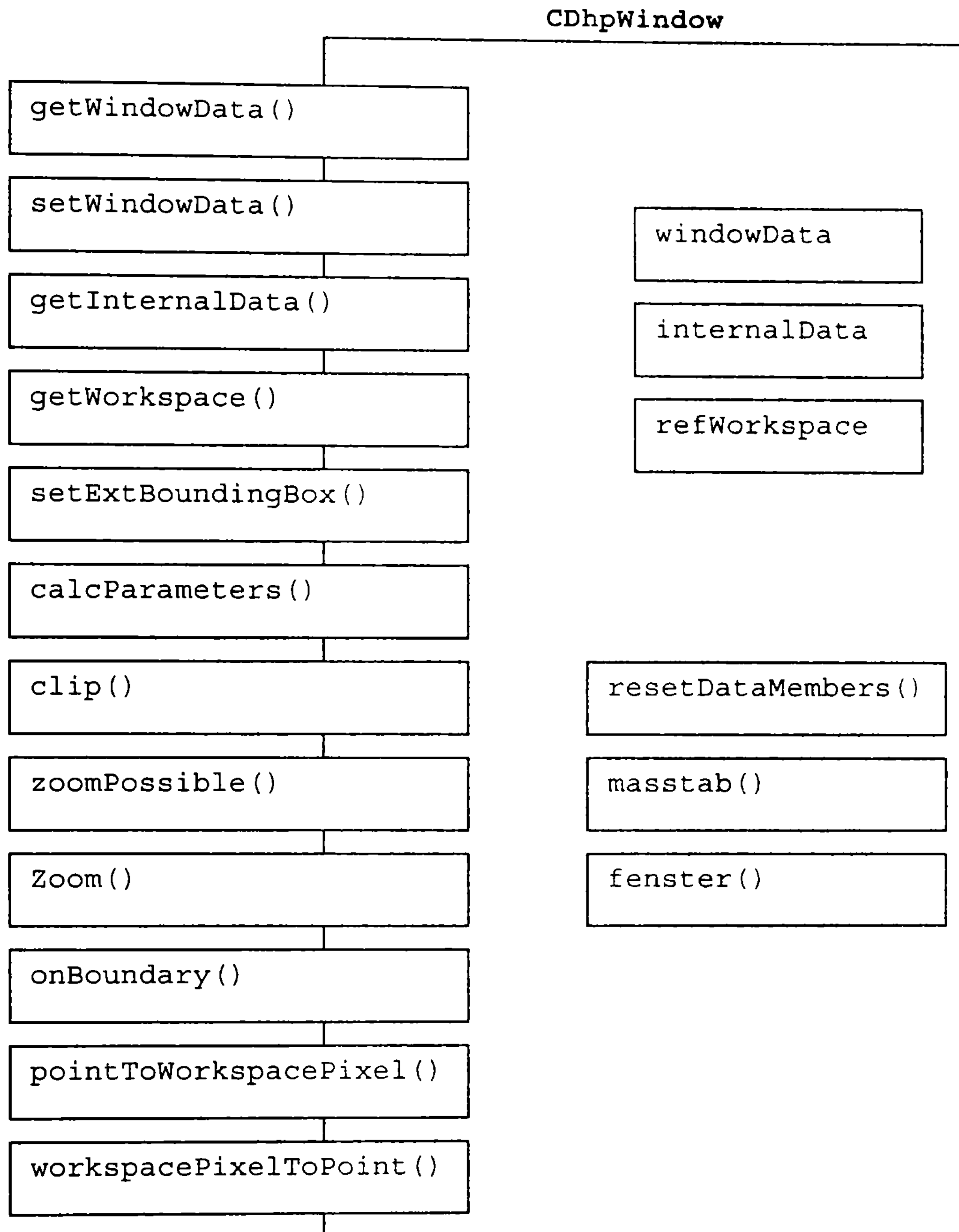
Structurarea proceselor este utilizată pentru descompunerea sistemelor mari de timp real, acest pas fiind omis în cazul bibliotecii de ferestre.

În etapa OOD sunt create componentele arhitecturale şi interfeţele acestora. Clasele din cadrul bibliotecii de ferestre şi interfeţele acestora sunt prezentate în continuare.

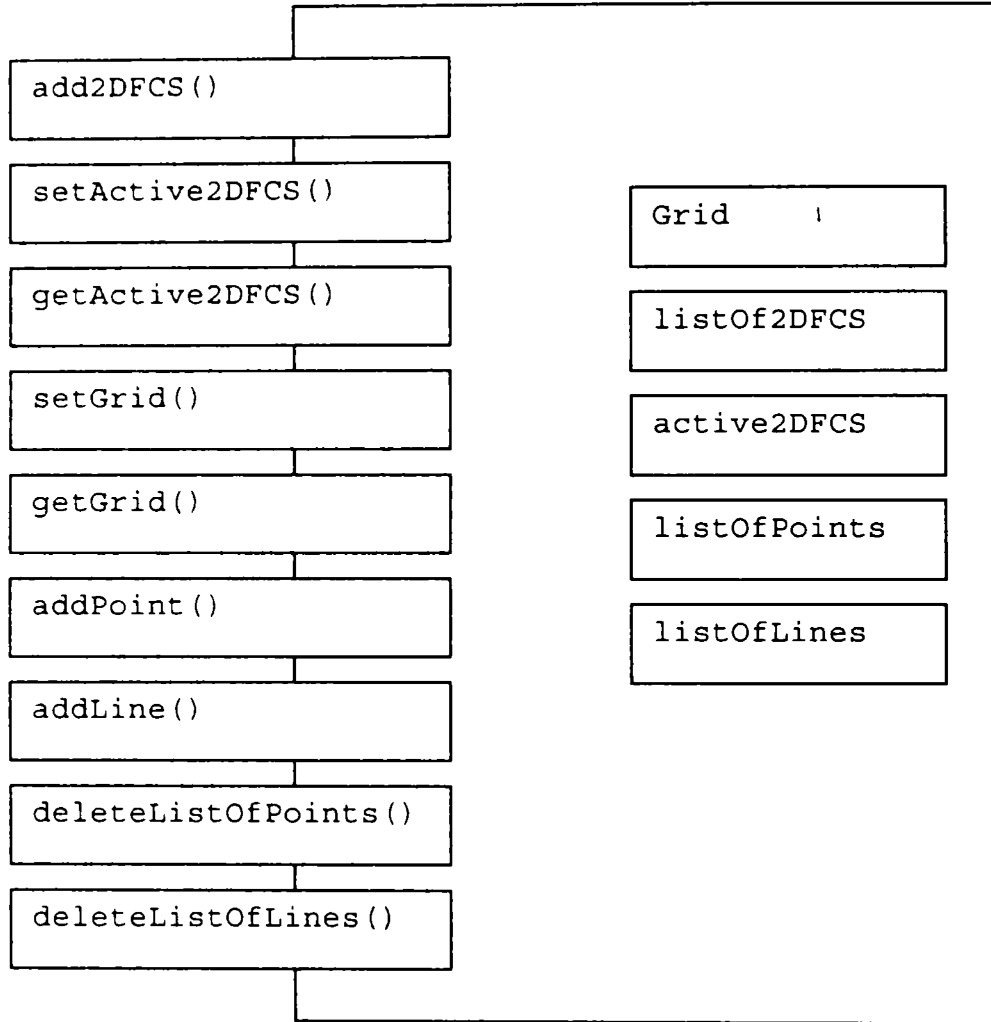
BUPT

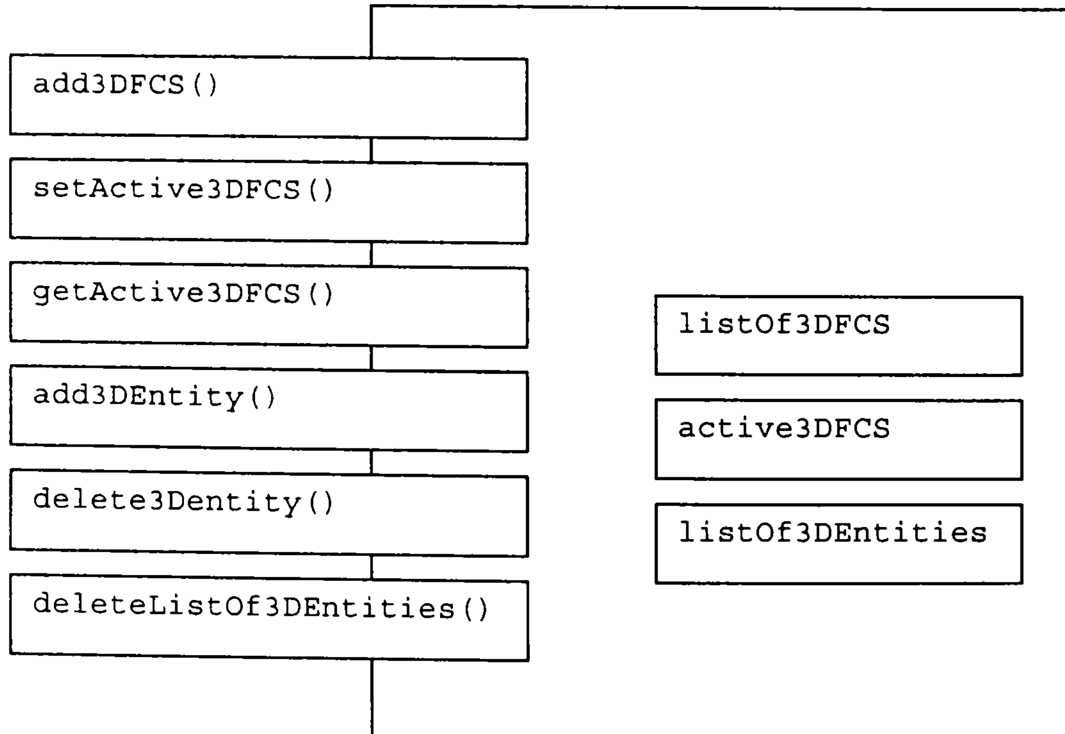


BUPT

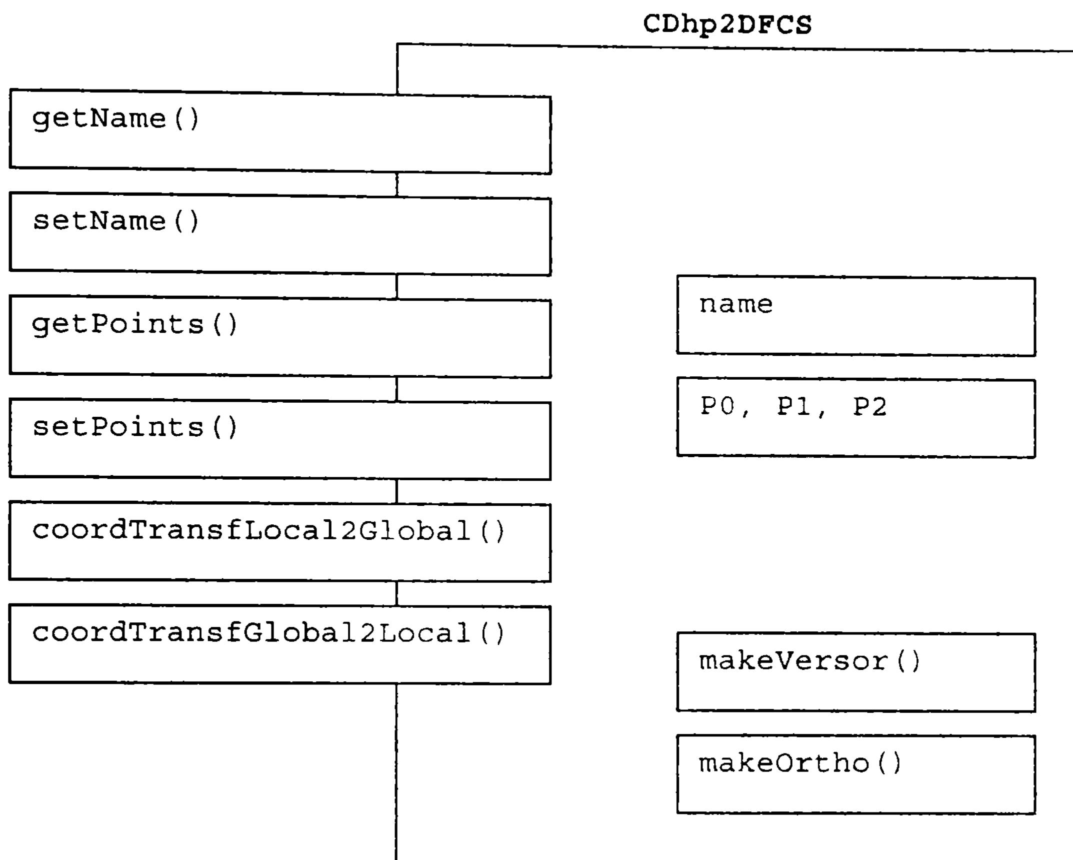


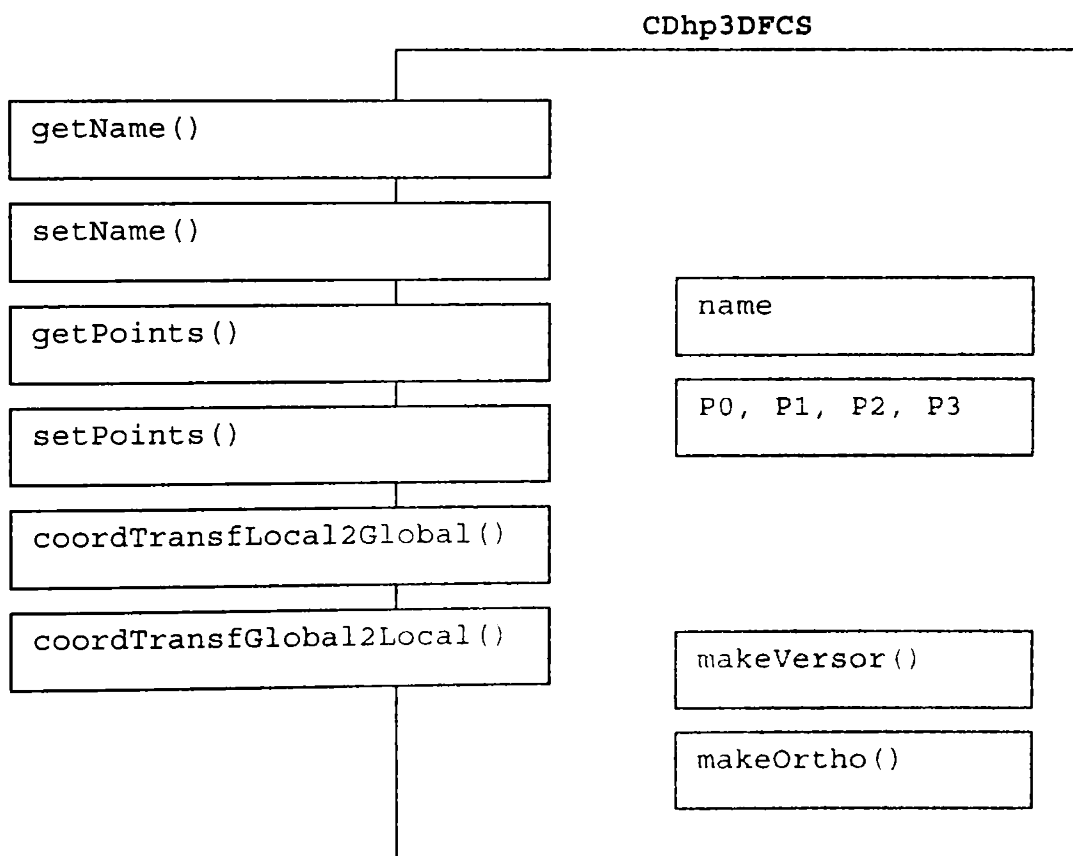
BUPT



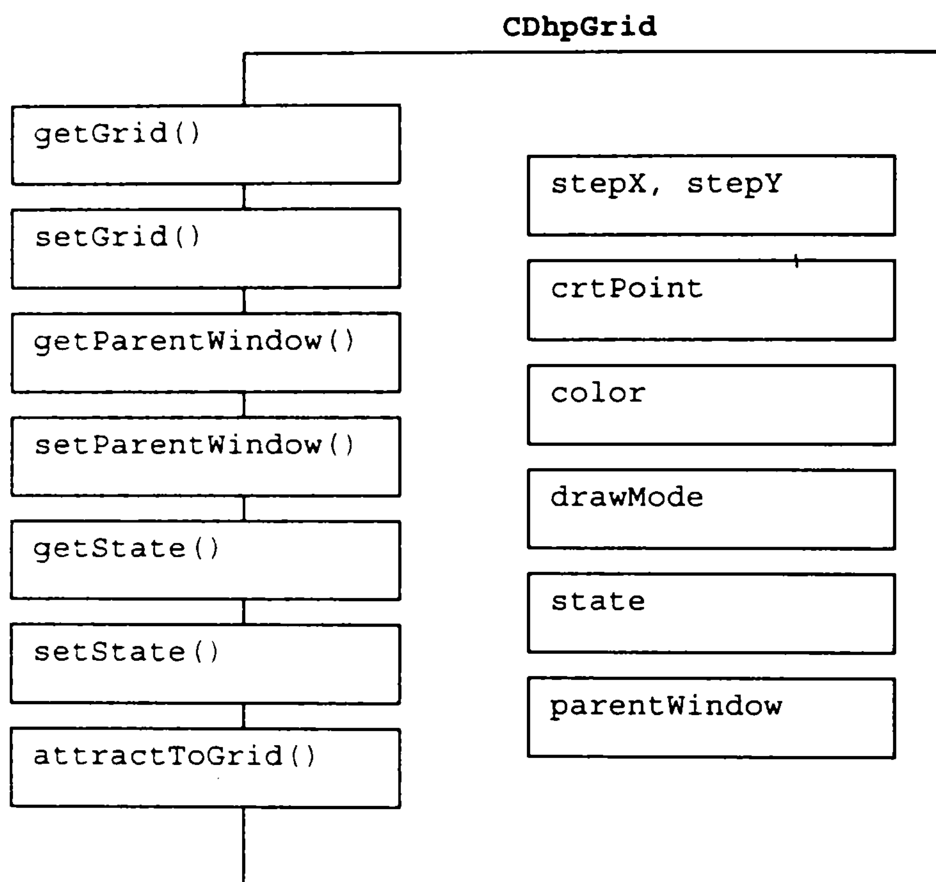


BUPT





BUPT



4.3.3.6 Implementarea

Pasul următor îl constituie implementarea utilizând limbajul de programare C++. Tranziţia de la proiectare la implementare respectă următoarele reguli, bazate pe cele enunţate în acest sens în capitolul anterior:

- clase → clase C++
- operaţii → funcţii membre
- atribute → date membre
- ascunderea informaţiei → segmente *private* şi *protected* în interfaţa clasei
- relaxarea ascunderii informaţiei → utilizarea noţiunii de *friend*
- ierarhia de moştenire → utilizarea moştenirii
- agregarea → declararea de obiecte ca date membre
- modularitatea → separarea fişierelor header de cele sursă
- tratarea excepţiilor → mecanismul C++ de tratare a excepţiilor sau utilizarea sistemului bazat pe aserţiuni

BUPT

Codul sursă C++ ce a rezultat în procesul de dezvoltare a bibliotecii de ferestre este prezentat în anexă. Mecanismul de tratare a erorilor implementat în cadrul bibliotecii a fost prezentat din punct de vedere teoretic în cadrul capitolului anterior.

Pentru evaluarea claselor rezultate se utilizează metrica LCOM, definiția Li şi Henry, descrisă în cadrul capitolului anterior. Calculul valorii LCOM se efectuează cu ignorarea variabilelor moştenite şi cu includerea constructorului.

```
Class CdhpWorkspace
  {
  protected:
    CdhpList<CdhpWindow>   windowList; //lista de ferestre
    CdhpWindow             *activeWindow; //fereastra activa
    DhpWorkspaceData       wsd;        //parametrii de interfata

    long                   confnumxpixels, //date interne
                           confnumypixels;
    double                 pixel_per_xmm,
                           pixel_per_ymm;

  public:
    CdhpWorkspace(void);
    CdhpWorkspace(  //constructor explicit
                  long pxmin, //limitele exprimate în [pixeli]
                  long pymin, //ale spatiului de lucru in
                  long pxmax, // fereastra aplicatiei
                  long pymax,
                  double wxmm = 0.0, //dim. X a sp. De lucru [mm]
                  double wymm = 0.0  //dim. Y a sp. De lucru [mm]
                 );
    CdhpWorkspace(
                  DhpWorkspaceData &wsdata
                 );
    ~CdhpWorkspace();

    void DeleteWindowList(void); //sterge intreaga lista de
                                 //ferestre, inclusiv continutul
    void SetData(
                 DhpWorkspaceData &wsdata
                );
    void GetData(DhpWorkspaceData &wsdata){wsdata = wsd;}
    long GetConfnumxpixels(void){return confnumxpixels;}
    long GetConfnumypixels(void){return confnumypixels;}
    double GetPixel_per_xmm(void){return pixel_per_xmm;}
    double GetPixel_per_ymm(void){return pixel_per_ymm;}

    void AddWindow( //adauga o fereastra la lista de ferestre
                   CdhpWindow *ptrWin
                  );
    CdhpList<CdhpWindow> &GetWindowList(void){return windowList;}
```

BUPT

```
        void SetActiveWindow(
                                    CdhpWindow *newActiveWindow
                                  );
        CdhpWindow *GetActiveWindow(void);
        void ShowDialog2DFCS();
    private:
        void ResetDataMembers(void);
        void CalcInternalData(void);
};
```

Tabelul de mai jos prezintă variabilele membre ale clasei accesate de către fiecare din metodele acesteia:

metoda \ variablia		windowList	activeWindow	wsd	internalData
1	CdhpWorkspace		•		•
2	CdhpWorkspace		•	•	•
3	CdhpWorkspace		•	•	•
4	DeleteWindowList	•	•		
5	SetData			•	•
6	GetData			•	
7	GetConfnumxpixels				•
8	GetConfnumypixels				•
9	GetPixel_per_xmm				•
10	GetPixel_per_ymm				•
11	AddWindow	•			
12	GetWindowList	•			
13	SetActiveWindow		•		
14	GetActiveWindow		•		
15	ResetDataMembers				•
16	CalcInternalData				•

LCOM = {1,2,3,4,13,14}, {5,6}, {7,8,9,10,15,16}, {11,12} = 4

Valoarea LCOM = 1 se obține pentru o clasă în cadrul căreia coeziunea metodelor este maximă. Valoarea 4 obținută pentru metrica LCOM în cadrul clasei CdhpWorkspace denotă o bună coeziune a metodelor în cadrul clasei, dat fiind numărul total de 16 metode prezente în

BUPT

cadrul acesteia. Acest rezultat demonstrează că principiile încapsulării şi al ascunderii informației au fost respectate în procesul de proiectare şi implementare a clasei `CdhpWorkspace`.

4.3.4 Editorul 2D

A doua componentă din cadrul aplicației *DHP-Bauwerk* a cărei proces de dezvoltare este prezentat în cadrul capitolului actual este reprezentată de *editorul 2D*. Procesul de analiză, proiectare şi implementare a avut ca scop realizarea unui modul foarte bine delimitat ce poate funcționa în contextul unui mediu gazdă oferind servicii de generare şi de transformări geometrice asupra entităților geometrice plane.

4.3.4.1 Contextul aplicației client

Utilitatea editorului 2D în cadrul mediului *DHP-Bauwerk* se regăseşte în procesul de generare şi de prelucrare a pereților etajului curent în plan orizontal, în procesul de generare a elementelor de rezistență din cadrul structurii pereților individuali, caz în care planul în care se lucrează este planul XoZ al acestora, precum şi în procesul de generare a conturului exterior şi a celor interioare ale acoperişului. Funcțiile editorului 2D pot fi apelate pentru un plan oarecare în spațiu, astfel că entitățile geometrice plane generate cu ajutorul lui pot servi ca elemente auxiliare în procesul de modelare a solidelor în cadrul componentei *DICAM: Freie Konstruktion* amintite în descrierea generală a aplicației *DHP-Bauwerk.*

4.3.4.2 Definirea cerințelor sistem

Cerințele pe care trebuie să le îndeplinească modulul *editor 2D* vor fi rezumate în cele ce urmează:

Cerințe legate de suportul hardware şi software

- Nucleul funcțional al editorului 2D este independent de platformă, conținând exclusiv aparatul matematic şi componenta de gestiune necesare generării şi prelucrării entităților geometrice plane.
- Conexiunea între editorul 2D şi suportul hardware al maşinii gazdă este realizată de către mediul gazdă în cadrul căruia rulează editorul 2D. Portarea editorului 2D pe alte platforme

BUPT

va ridica doar probleme legate de compilarea codului în cadrul unui mediu capabil să genereze cod executabil pentru respectiva platformă.

Cerinţe legate de comunicaţia client-server

- Editorul 2D reprezintă un server ce pune la dispoziţia mediului gazdă servicii de generare şi prelucrare a entităţilor geometrice plane.
- Mediul gazdă generează comenzile pentru editorul 2D şi preia elementele geometrice 2D generate sau modificate de către acesta. Interfaţa cu utilizatorul prezentă în cadrul editorului 2D conţine:
  - componenta de selecţie a unui element geometric dintr-o mulţime a cărei elemente satisfac o anumită condiţie
  - componenta de gestiune a comenzilor de modificare a stării editorului.

Cerinţe funcţionale

- Funcţiile puse la dispoziţie de către editorul 2D sunt:
  - funcţii de generare:
    - generarea unui punct
      - prin specificarea explicită a coordonatelor (carteziene sau polare)
      - la intersecţia a două drepte
      - piciorul perpendicularei dintr-un punct pe o dreaptă
      - la intersecţia între orizontala dusă dintr-un punct şi o dreaptă
      - la intersecţia între verticala dusă dintr-un punct şi o dreaptă
      - la mijlocul distanţei între două puncte date
      - care împarte un segment dat într-un anumit raport
    - generarea unui segment de dreaptă
    - generarea unui cerc
      - prin specificarea centrului şi a razei
      - prin specificarea a trei puncte necoliniare ce aparţin cercului
    - generarea unui arc de cerc
  - funcţii de transformări geometrice:
    - translaţii
    - rotaţii
    - scalări
    - oglindiri

BUPT

- facilități auxiliare pentru construcție:
  - prindere de obiecte
  - funcții de generare de elemente geometrice auxiliare utilizate în cadrul funcțiilor de generare și de transformări geometrice

Cerințe legate de implementare

- Editorul 2D se va implementa sub forma unei biblioteci dinamice, astfel încât serviciile puse la dispoziție să poată fi accesate de către aplicații client dezvoltate în diverse limbaje de programare.
- Limbajul utilizat pentru implementarea editorului 2D este C++. Această decizie este susținută pe de o parte de suportul pentru orientarea spre obiecte furnizat în mod nativ de acest limbaj, pe de altă parte de popularitatea și larga răspândire a acestui limbaj în majoritatea spectrului platformelor hardware și al sistemelor de operare uzuale.

4.3.4.3 Analiza și proiectarea orientată spre obiecte

4.3.4.3.1 Analiza domeniului

Analiza domeniului are ca punct de plecare specificația cerințelor sistemului. Un prim rezultat al analizei îl constituie determinarea contextului în care va rula subsistemul *editor 2D*. Figura următoare ilustrează interacțiunea între editorul 2D, mediul gazdă și utilizator:

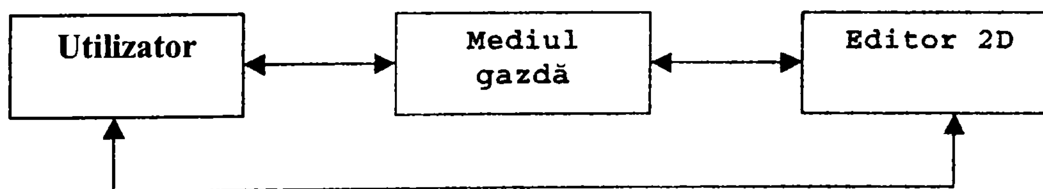


Figura 4.7 Interacțiunea editorului 2d cu mediul gazdă și cu utilizatorul

Un prim set de clase și obiecte ale lumii reale poate fi derivat în acest moment din specificația sistemului prezentată mai sus:

BUPT

- *Editorul 2D*: reprezintă subsistemul responsabil cu gestiunea elementelor geometrice plane. Acesta conţine totalitatea elementelor necesare furnizării serviciilor puse la dispoziţie clienţilor.
- *Spaţiul de lucru*: parte componentă a editorului 2D, reprezintă spaţiul destinat reprezentării entităţilor geometrice plane prelucrate în cadrul acestuia. Spaţiul de lucru conţine:
    - *Lista de ferestre* dotate cu toate facilităţile care au fost menţionate în cadrul bibliotecii de ferestre:
        - *Lista de sisteme de coordonate libere*
        - *rastrul*
- *Interfaţa cu mediul gazdă*: reprezintă suportul pentru schimbul de date şi de informaţii între mediul gazdă şi editorul 2D
- *Interfaţa utilizator*: reprezintă obiectul prin intermediul căruia editorul 2D percepe reacţia utilizatorului în cadrul fiecărei stări în care se găseşte. Interfaţa utilizator este compusă din următoarele obiecte
    - *Linia de comandă*: permite utilizatorului să introducă comenzi pentru editorul 2D utilizând un format predefinit
    - *Mouse-ul*: gestionează comenzile generate de către utilizator prin intermediul dispozitivului periferic mouse şi le transmite obiectului cursor.
    - *Cursorul*: facilitează identificarea anumitor elemente în cadrul spaţiului de lucru (selecţie, punctare etc.)

Relaţiile existente între obiectele lumii reale vor fi ilustrate în cadrul *diagramei de relaţii între entităţi*, reprezentând un model static al editorului 2D:

BUPT

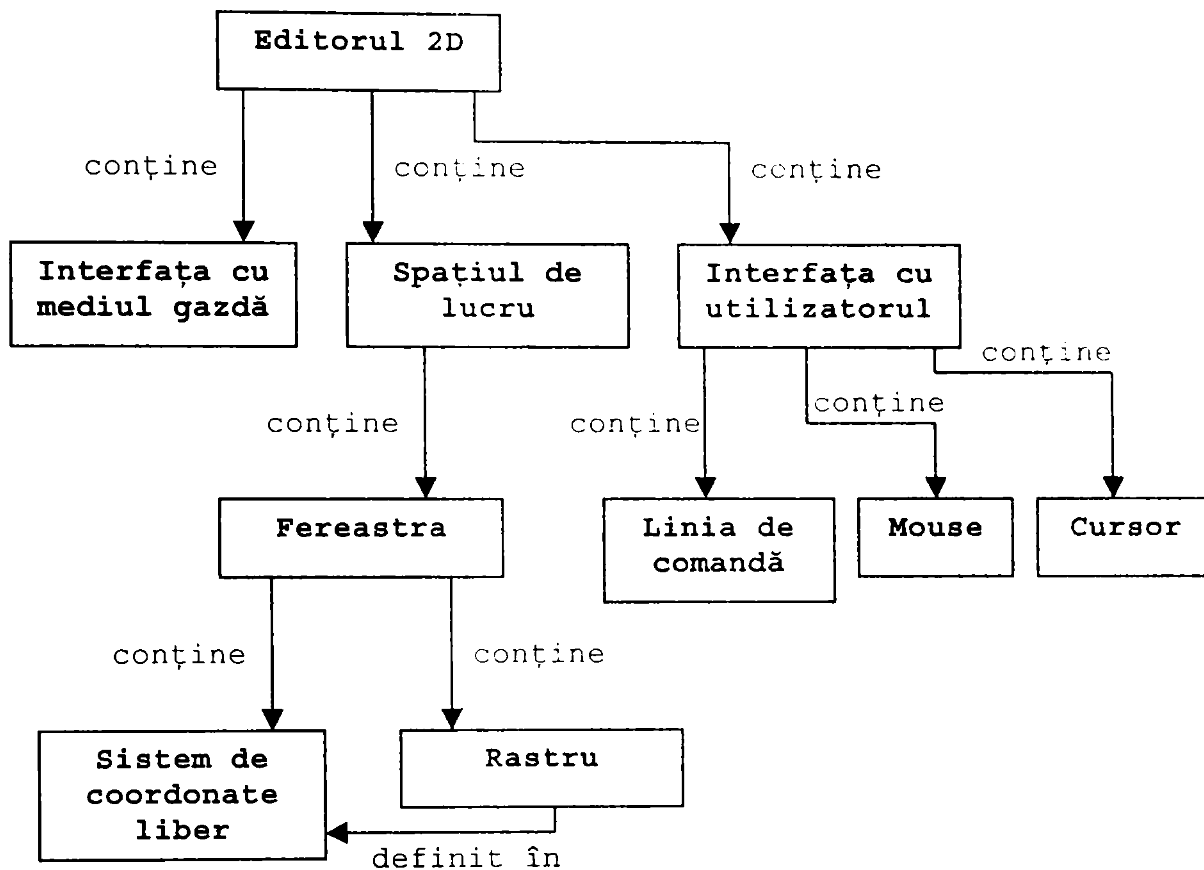


Figura 4.8 Diagrama de relații între entități pentru editorul 2D

4.3.4.3.2 Analiza cerințelor sistemului

Etapa de analiză a cerințelor sistemului are ca scop identificarea modelului dinamic şi a celui funcțional al sistemului. Pentru generarea modelului dinamic se va recurge la o abordare bazată pe cazuri de uz şi scenarii, descrisă în amănunt în capitolul anterior al prezentei lucrări.

4.3.4.3.2.1 Cazuri de uz

Un prim pas în determinarea modelului dinamic al sistemului constă în identificarea tuturor actorilor şi a cazurilor de uz ce descriu funcționalitatea acestuia. Figura următoarea ilustrează actorii şi cazurile de uz pentru editorul 2D. Cazurile de uz etichetează elipsele din cadrul subsistemului „Editor 2D" iar actorii sunt reprezentați în exteriorul acestui subsistem. În cazul de față actorii sunt reprezentați de către subsistemul *mediu gazdă*, şi de către *utilizator*.

BUPT

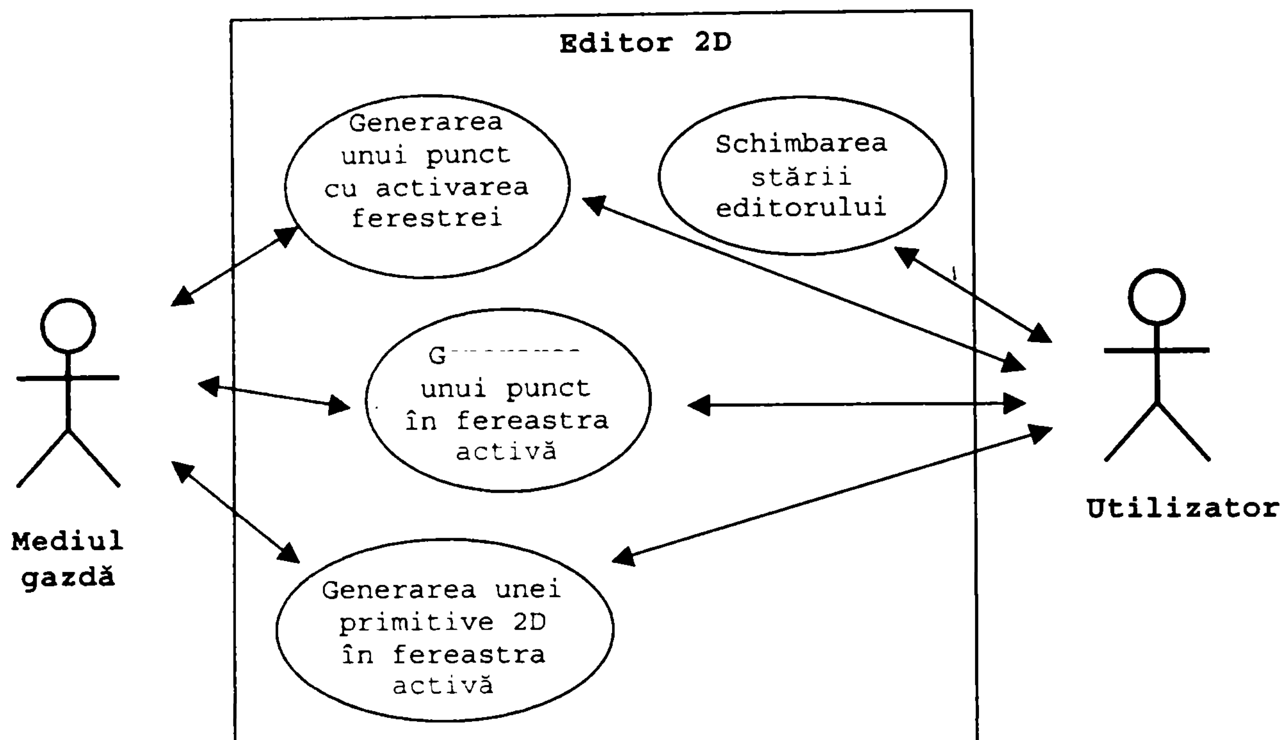


Figura 4.9 Actori şi cazuri de uz în cadrul aplicaţiei Editor 2D

Cazul de uz „Schimbarea stării editorului”

Schimbarea stării editorului se poate efectua prin intermediul obiectului *Linie de comandă.* Apăsarea tastei „F” are ca efect activarea/dezactivarea rastrului iar tasta „O” are ca efect activarea/dezactivarea modului ortogonal al cursorului (modul ortogonal activ are ca efect limitarea tipurilor de segmente de dreaptă generate la segmente verticale şi segmente orizontale). Apăsarea tastelor „+” respectiv „-” are ca efect incrementarea respectiv decrementarea nivelului de zoom în cadrul ferestrei active. Combinaţiile de taste „Ctrl+PgUp” respectiv „Ctrl+PgDown” au ca efect mărirea respectiv micşorarea dimensiunii reticulului cursorului. Următorul tabel decizional surprinde modul de lucru al cazului de uz „schimbarea stării editorului”:

STIMUL	S-a apăsat tasta „F”	DA					
	S-a apăsat tasta „O”		DA				
	S-a apăsat tasta „+”			DA			
	S-a apăsat tasta „-”				DA		
	S-a apăsat tasta „CTRL+PgUp”					DA	
	S-a apăsat tasta „CTRL+PgDown”						DA
RĂSPUNS	Schimbare stare rastru	DA					
	Schimbare stare cursor (mod orto)		DA				
	Incrementare nivel zoom			DA			

BUPT

	Decrementare nivel zoom				DA		
	Mărire domeniu prindere cursor					DA	
	Micşorare domeniu prindere cursor						DA

Diagrama de trasare a evenimentelor corespunzătoare scenariilor din cadrul cazului de uz curent este prezentată în figura următoare:

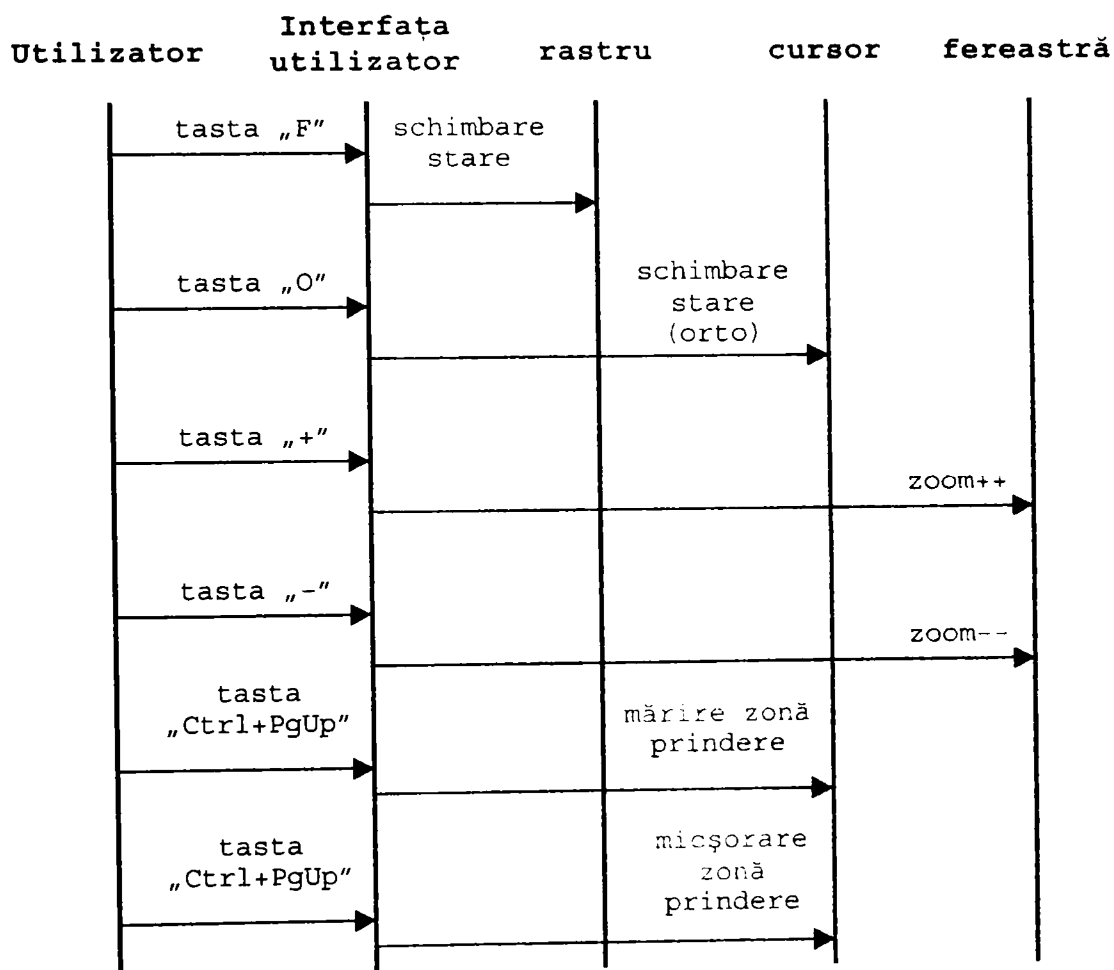


Figura 4.10 Diagrama de trasare a evenimentelor pentru schimbarea stării editorului

Metoda OOSE [JAC92] propune în acest moment clasificarea obiectelor în obiecte de interfaţă, obiecte de control şi obiecte entitate. Pentru cazul de uz curent, obiectul de interfaţă este reprezentat de interfaţa utilizator, obiectul de control este reprezentat de linia de comandă iar din categoria obiectelor entitate fac parte rastrul, cursorul şi fereastra.

Obiectele de analiză determinate până în acest moment sunt grupate în cadrul următoarei diagrame de flux al evenimentelor. Tipul fiecărui obiect apare adnotat în cadrul pictogramei corespunzătoare. Obiectele de interfaţă sunt marcate cu caracterul „I", obiectele de control cu caracterul „C" iar obiectele entitate cu caracterul „E".

BUPT

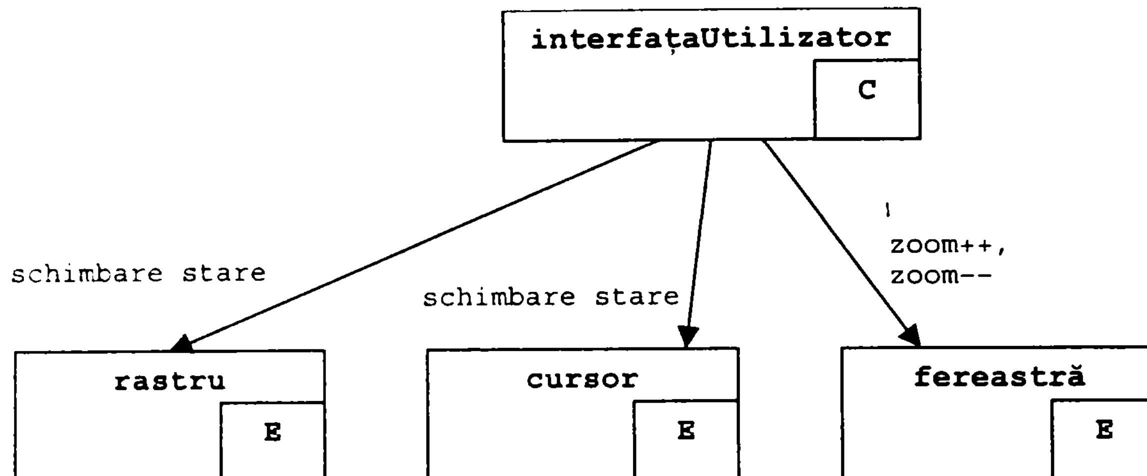


Figura 4.11 Diagrama de flux a evenimentelor pentru schimbare stare editor

Cazul de uz „Generarea unui punct în fereastra activă":

Unul din cazurile de uz esențiale ale editorului 2D este reprezentat de generarea unui punct în fereastra activă. Acest caz de uz va fi detaliat în cele ce urmează. Stimulii ce se aplică sistemului sunt reprezentați subliniat iar răspunsurile sistemului sunt prezentate *cursiv*.

1. Utilizatorul apelează prin intermediul mediului gazdă facilitatea de generare a unui punct
2. *Mediul gazdă transmite editorului 2D prin intermediul obiectului „interfața cu mediul gazdă" informația legată de contextul în care s-a apelat comanda de generare a unui punct.* În urma acestui pas editorul 2D va cunoaște mulțimea tuturor elementelor geometrice ale mediului gazdă accesibile în cadrul ferestrei de lucru.
3. *Controlul este predat obiectului „interfața cu utilizatorul".* Interacțiunea cu utilizatorul poate avea loc în unul din următoarele moduri:
   3.1. comenzi de schimbare a stării editorului, caz în care se apelează cazul de uz „schimbare stare editor"
   3.2. comenzi adresate „Liniei de comandă", caz în care utilizatorul are posibilitatea de a:
      3.2.1. specifica coordonatele punctului dorit prin:
         3.2.1.1.coordonate carteziene absolute
         3.2.1.2.coordonate carteziene relative (cu excepția primului punct)
         3.2.1.3.coordonate polare absolute
         3.2.1.4.coordonate polare relative (cu excepția primului punct)
         3.2.1.5.apăsarea tastelor săgeți (cu excepția primului punct)
   3.3. comenzi transmise prin intermediul obiectului „Mouse" obiectului „Cursor", caz în care:

BUPT

3.3.1. utilizatorul plasează reticulul cursorului în poziţia corespunzătoare punctului ce se doreşte a fi generat şi apasă butonul stâng al mouse-ului.

3.4. utilizatorul apasă butonul din mijloc al mouse-ului pentru a afişa un meniu de funcţii auxiliare care permit generarea unui punct prin următoarele funcţii:

3.4.1. punctul situat la intersecţia a două drepte

3.4.2. punctul situat la mijlocul distanţei între două puncte date

3.4.3. piciorul perpendicularei dintr-un punct pe o dreaptă

3.4.4. intersecţia orizontalei prin punctul dat cu o dreaptă dată

3.4.5. intersecţia verticalei prin punctul dat cu o dreaptă dată

3.5. utilizatorul apasă butonul mouse dreapta pentru a abandona funcţia de generare de punct. Actualul caz de uz se încheie

4. *Editorul 2D procesează comanda* generată de către utilizator. Se disting următoarele situaţii:

4.1. S-au specificat coordonatele punctului dorit prin intermediul liniei de comandă (3.1.1). *Editorul 2D furnizează aceste coordonate mediului gazdă şi cazul de uz curent se încheie*

4.2. S-a executat o comandă de schimbare a stării editorului (3.1.2). *Se apelează cazul de uz „schimbarea stării editorului" şi se revine la pasul 3.*

4.3. S-a apăsat tasta mouse stânga în poziţia dorită. Se efectuează următoarele acţiuni:

4.3.1. *Se formează lista tuturor punctelor* care au fost furnizate de către mediul gazdă la (2), *situate în raza de prindere a cursorului* şi care satisfac prinderile setate.

4.3.2. *Se analizează lista generată anterior*, făcându-se distincţie între următoarele cazuri:

4.3.2.1. Lista nu conţine nici un element: *coordonatele pixelului* corespunzător poziţiei mouse-ului în momentul apăsării pe butonul mouse stânga sunt *convertite în coordonate ale spaţiului* $2D_{continuu}$ reprezentat în fereastră şi *oferite utilizatorului spre confirmare* (cu posibilitatea ca acestea să fie modificate)

4.3.2.1.1. Dacă utilizatorul confirmă coordonatele (iniţiale sau modificate), *punctul este transmis mediului gazdă şi cazul de uz se încheie*

4.3.2.1.2. Dacă utilizatorul renunţă la aceste coordonate *se revine la punctul (3)*

4.3.2.2. Lista conţine un singur element: *coordonatele punctului* conţinut în listă sunt *returnate mediului gazdă şi cazul de uz se încheie.*

4.3.2.3. Lista conţine mai multe elemente: *se oferă fiecare dintre aceste elemente utilizatorului pentru a fi confirmate*

BUPT

4.3.2.3.1. Dacă utilizatorul confirmă vreun element din cadrul listei, *coordonatele punctului respectiv sunt returnate mediului gazdă și cazul de uz se încheie.*

4.3.2.3.2. Dacă utilizatorul nu acceptă nici unul din elementele oferite *se revine la punctul (3).*

4.3.3. S-a apăsat tasta mouse mijloc: *se afișează meniul popup în cadrul căruia se selectează funcția dorită pentru generarea unui punct (recursivitate: aceste funcții vor apela la rândul lor cazul de uz actual).*

4.3.4. S-a apăsat butonul mouse dreapta: *se predă controlul mediului gazdă fără a genera nici un punct*

Pasul următor este reprezentat de identificarea stimulilor și a răspunsurilor în cadrul cazului de uz de mai sus. Rezultatul acestui proces se regăsește în cadrul descrierii de mai sus a cazului de uz, unde stimulii au fost marcați prin text subliniat iar răspunsurile sistemului prin *text cursiv*.

Identificarea stimulilor și a răspunsurilor permite generarea tabelului decizional, tabel în care rândurile de la început corespund stimulilor iar rândurile de la sfârșit corespund răspunsurilor sistemului. Fiecare coloană a acestui tabel reprezintă un scenariu.

STIMULI	**LiniaDeComandă:userInput**: introducere coordonate valide	DA								
	userInput: schimbare stare editor		DA							
	mouse:userInput: buton stânga			DA	DA	DA	DA	DA		
	mouse:userInput: buton mijloc								DA	
	mouse:userInput: buton dreapta									DA
	editor2D: lista de puncte candidate nu conține nici un element			DA	DA					
	editor2D: lista de puncte candidate conține un element					DA				
	editor2D: lista de puncte candidate conține mai multe elemente						DA	DA		

BUPT

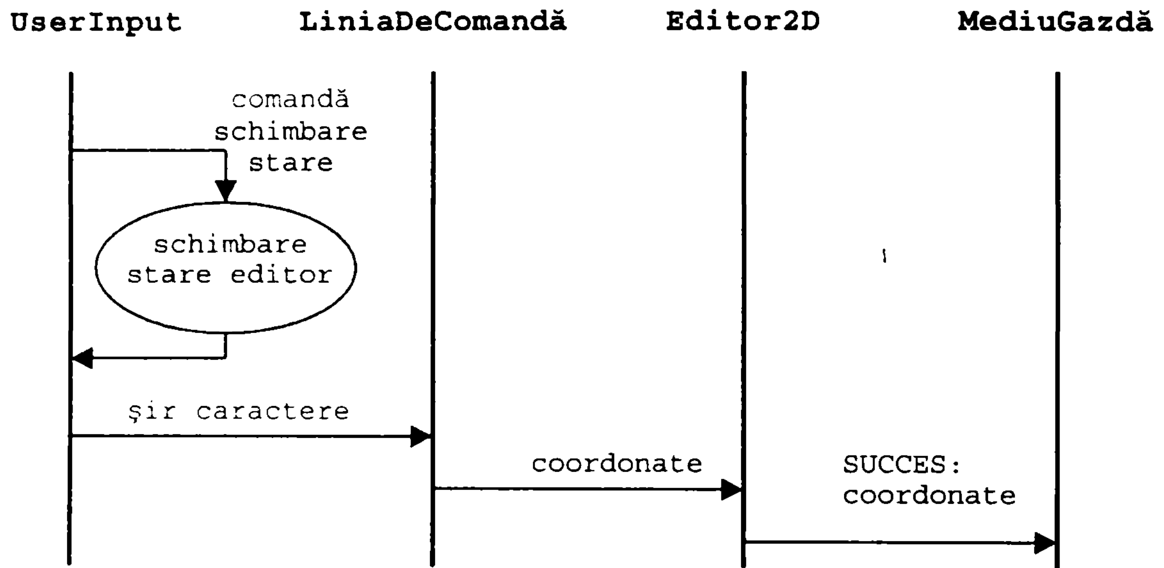
	Utilizator: Confirmare punct oferit						DA	NU		
	Utilizator: Confirmare coordonate			DA	NU					
RĂSPUNSURI	**editor2D**: oferirea listei de puncte candidate spre confirmare						DA	DA		
	editor2D: oferirea coordonatelor punctului spre validare			DA	DA					
	editor2D: apel caz de uz „schimbare stare editor”		DA							
	editor2D: SUCCES: returnează coordonatele punctului spre mediul gazdă	DA		DA		DA	DA			
	editor2D: ABANDON: predă controlul mediului gazdă fără a returna punct									DA
	editor2D: predă controlul obiectului userInput				DA			DA		
	editor2D: apel funcții de generare puncte temporare (apel recursiv al cazului de uz actual)								DA	

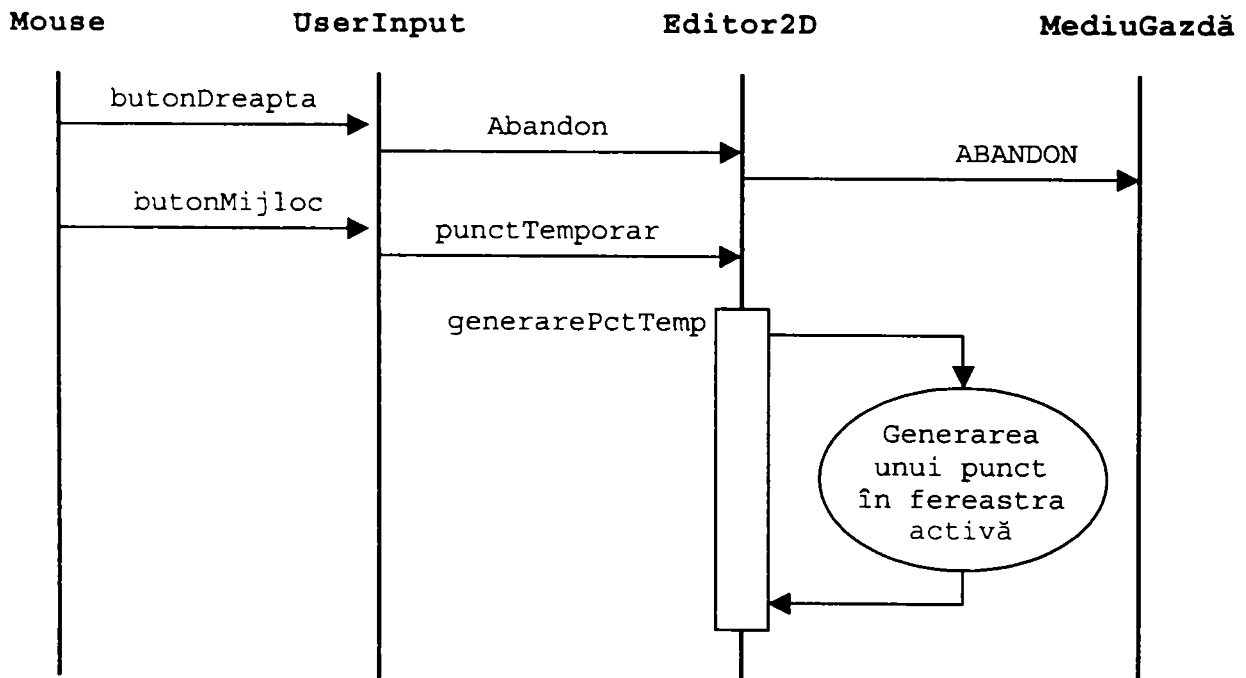
Tabelul decizional pentru cazul de uz „generare de punct în fereastra activă" conține datele de intrare în procesul de generare a diagramei de trasare a evenimentelor. Această diagramă reprezintă fiecare dintre obiectele participante la cazul de uz pe câte o coloană iar mesajele ce se transmit între aceste obiecte prin săgeți orizontale. Cronologia cazului de uz rezultă parcurgând diagrama de sus în jos.

În cazul editorului2D, diagrama de trasare a evenimentelor a fost divizată în următoarele trei subdiagrame pentru a îmbunătăți lizibilitatea, dată fiind importanța acesteia în procesul de generare a documentației. În fiecare dintre cele trei cazuri de uz obiectul de interfață situat pe coloana cea mai din stânga furnizează informațiile transmise de actor (utilizator) obiectului de control situat pe coloana următoare. Actuala abordare corespunde alternativei „FORK" propusă de Jacobson [JAC92], [ROW98] prezentând pentru situația actuală următoarele avantaje:

- comportamentul cazului de uz este concentrat la nivelul unui singur obiect
- responsabilitățile sunt concentrate la nivelul unui singur obiect
- posibilitatea de reutilizare a obiectelor entitate este mai pronunțată

BUPT





BUPT

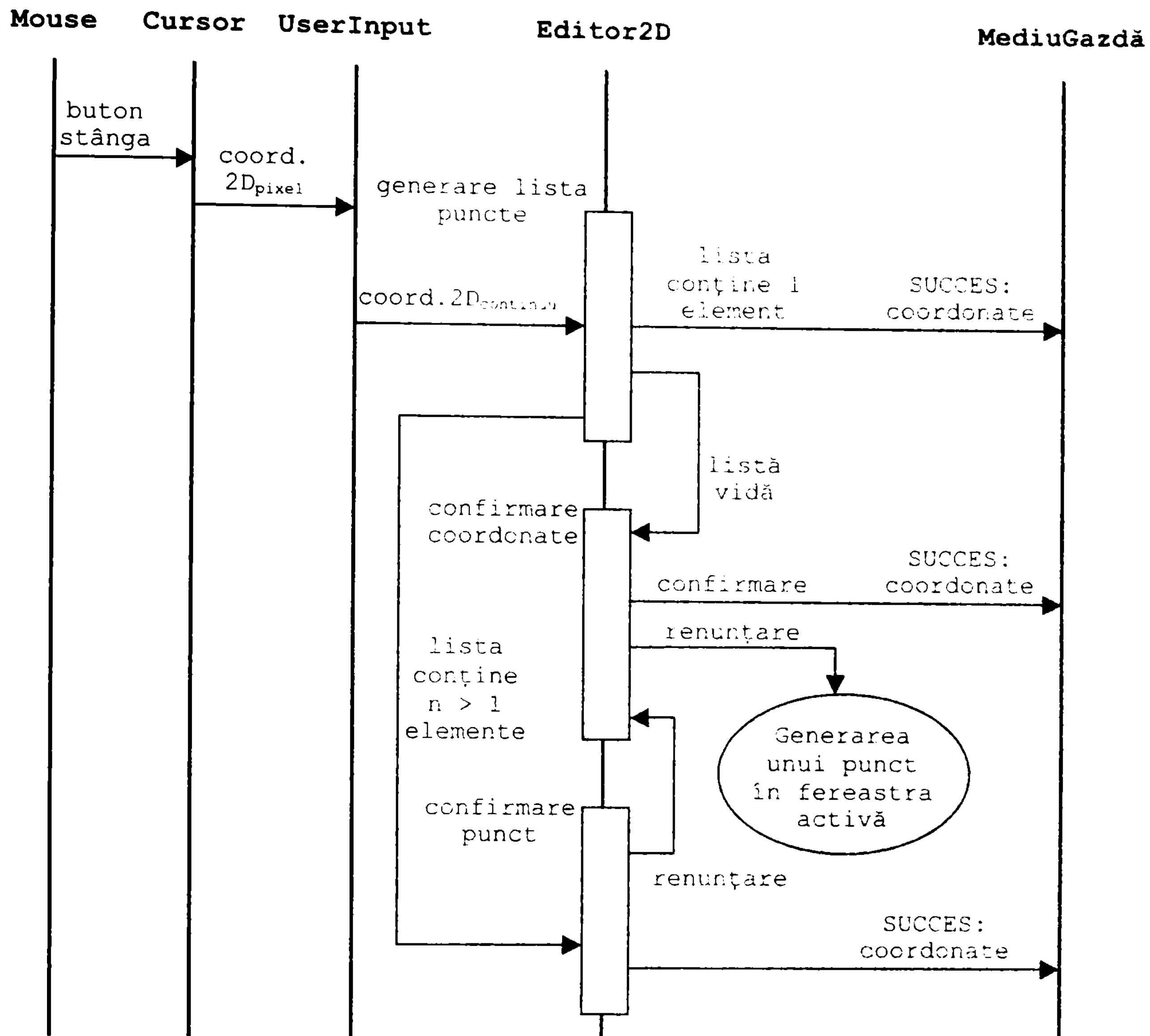


Figura 4.12 Diagrame de trasare a evenimentelor pentru „Generare punct în fereastra activă"

Avantajul major al faptului că un singur obiect concentrează comportamentul cazului de uz constă în faptul că eventualele modificări la nivelul cerințelor sistemului vor fi izolate cu efort minim la nivelul unei singure componente a sistemului. Modificarea codului afectează doar un număr redus de componente, efortul necesar acestui tip de modificări fiind minim.

4.3.4.3.2.2 Analiza comportamentului dinamic și diagrama de tranziție a stărilor

Procesul de analiză în ceea ce privește obiectul „linie de comandă" necesită o abordare bazată pe diagrama de tranziție a stărilor. Cerințele legate de sintaxa comenzilor acceptate de către linia de comandă sunt prezentate în continuare.

Coordonate carteziene absolute: cele două coordonate carteziene ale unui punct în spațiul bidimensional sunt descrise printr-o pereche de numere reale separate printr-un spațiu. Primul număr reprezintă coordonata X iar cel de-al doilea coordonata Y a punctului.

BUPT

Exemplu: șirul „`2.56 5.12`" are ca efect generarea unui punct a cărui abscisă are valoarea 2.56 și a cărui ordonată este 5.12.

Coordonate polare absolute: cele două coordonate polare ale unui punct P în spațiul bidimensional sunt descrise printr-o pereche de numere reale separate prin caracterul ‚<'. Primul număr reprezintă distanța față de originea O a sistemului de coordonate iar cel de-al doilea unghiul, măsurat în grade, între axa OX a sistemului de coordonate activ și semidreapta OP. *Exemplu*: șirul „`2.56<45.5`" are ca efect generarea unui punct a cărui distanță față de origine este de 2.56 și unghiul dintre semidreapta ce pleacă din origine și trece prin acest punct și axa OX este de 45.5°.

Coordonate carteziene relative: dacă șirul care generează un punct în coordonate carteziene absolute este prefixat de caracterul ‚R' atunci coordonatele introduse reprezintă distanțele de-a lungul axei OX respectiv OY a noului punct față de ultimul punct introdus. *Exemplu*: „`R2.56 5.12`"

Coordonate polare relative: dacă șirul care generează un punct în coordonate polare absolute este prefixat de caracterul ‚R' atunci coordonatele introduse reprezintă distanța noului punct față de ultimul punct introdus, respectiv unghiul pe care îl face cu axa OX semidreapte ce unește ultimul punct introdus cu noul punct generat. *Exemplu*: „`R2.56<45.5`"

Introducerea unei valori negative pentru prima coordonată absolută: una dintre facilitățile editorului 2D este „lupa rapidă", ce are ca efect efectuarea operației de „zoom-in" la apăsarea tastei <+> respectiv „zoom-out" la apăsarea tastei <–>. Introducerea unei valori negative pentru prima coordonată absolută ar necesita introducerea caracterului ‚–' pe prima poziție a linie de comandă, caracter care ar fi interceptat de către lupă. Pentru a face posibilă introducerea unei valori negative pentru prima valoare a coordonatelor absolute s-a convenit ca aceasta să fie prefixată de caracterul ‚G'. *Exemplu*: șirul „`G-2.56 5.12`" are ca efect generarea unui punct a cărui abscisă are valoarea –2.56 și a cărui ordonată este 5.12.

Elementul pe care se bazează procesul de realizare a unui sistem care să implementeze cerințele prezentate mai sus îl reprezintă *diagrama de tranziție a stărilor*. Această diagramă reprezintă un automat finit determinist. Generarea acestui automat pe baza unei metode specifice teoriei limbajelor formale [AHO77] este prezentată în cele ce urmează.

Alfabetul limbajului liniei de comandă este reprezentat de mulțimea:

Σ = {`0...9,G,R, ,<,.,-`}

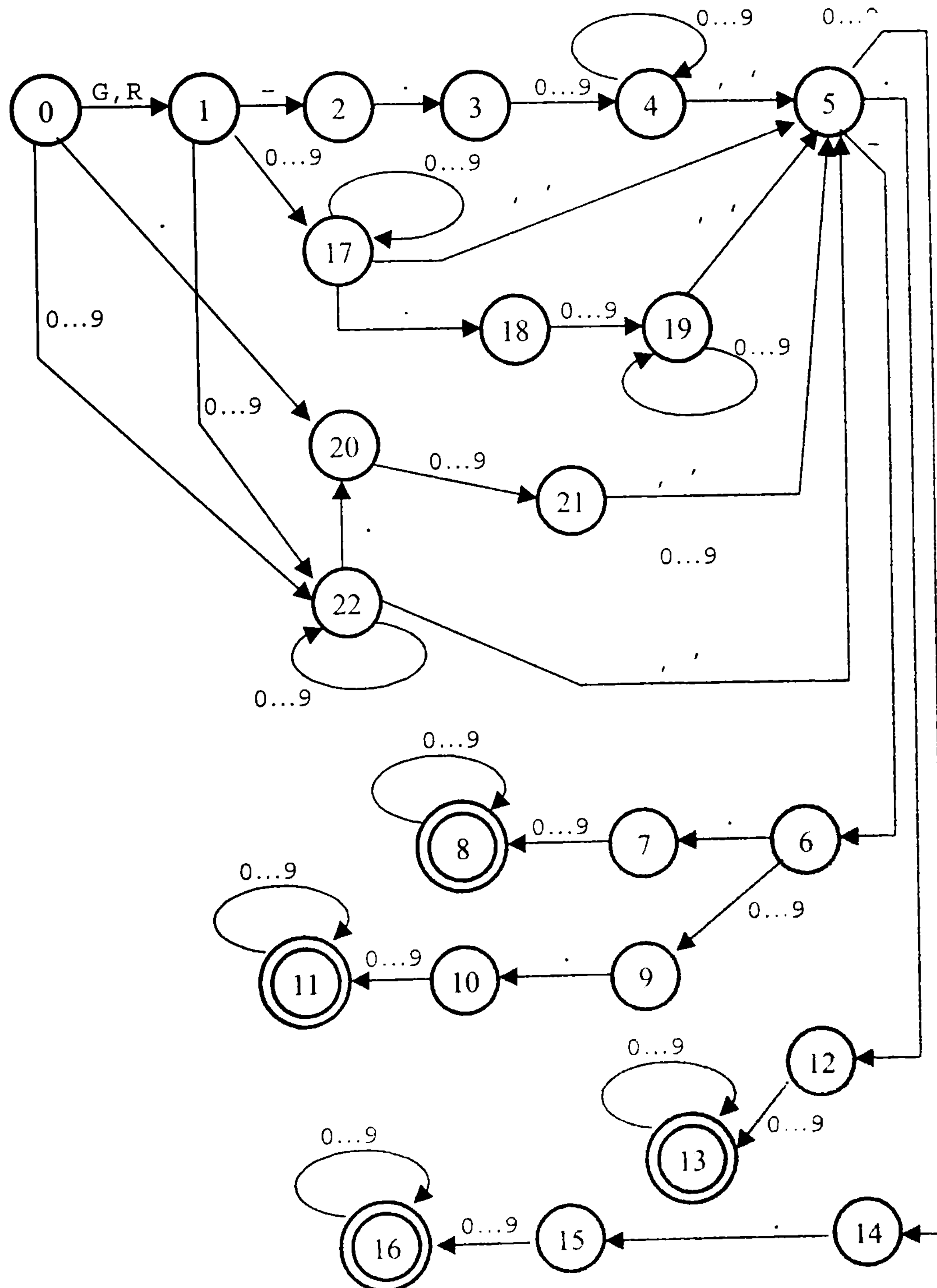
BUPT

Limbajul pe care îl acceptă linia de comandă este definit pe baza următoarei expresii regulate:

$$R = (G-|R|R-|-|\varepsilon)\ (0...9)^*(.|\varepsilon)\ (0...9)^*(\ \ |<)\ (\ \ \varepsilon|-)\ (0...9)^*(.|\varepsilon)\ (0...9)^*$$

unde ε semnifică caracterul vid.

Automatul finit determinist care implementează această expresie regulată este prezentat în continuare:



BUPT

În urma minimizării acestui automatului finit [AHO77, pag. 99] se obţine următoarea configuraţie de stări:

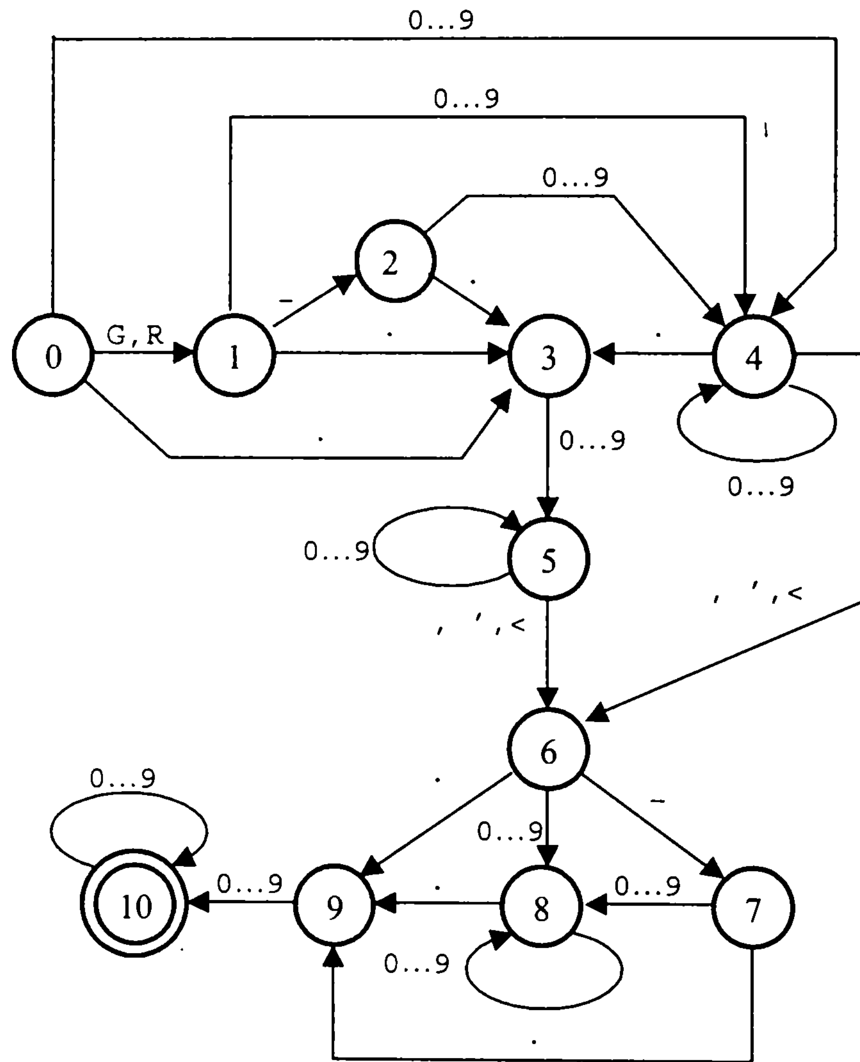


Figura 4.13 Diagrama de tranziţie a stărilor pentru linia de comandă

Linia de comandă va fi implementată pe baza unei ierarhii de clase generate pe baza diagramei de tranziţie a stărilor prezentată mai sus. Metoda de generare a acestei ierarhii de clase [FAI93] a fost prezentată în cadrul capitolului precedent.

4.3.4.3.3 Proiectarea sistemului

Faza de proiectare a sistemului descrisă în cadrul capitolului anterior constă din etapele:

- partiţionare
- configurare

BUPT

Cele două etape au menirea de a diviza sistemul în subsisteme gestionabile, respectiv de a aloca cerințele sistem corespunzătoare acestor subsisteme componentei hardware respectiv software.

Editorul 2D constituie un sistem compact ce nu necesită divizarea în subsisteme. Justețea acestei decizii va fi demonstrată pe baza criteriilor prezente în cadru metodei OOSE pentru asocierea scenariilor, obiectelor și subsistemelor [JAC92, cap. 7 și 10]:

- *Modificările la nivel de sistem*: un criteriu de divizare a unui sistem în subsisteme se bazează pe ușurința cu care modificările cerințelor sistemului se transpun în codul sursă, astfel se urmărește ca fiecare subsistem să fie legat de un singur actor. Actorul principal cu care interacționează editorul 2D este mediul gazdă prin intermediul căruia utilizatorul apelează facilitățile oferite de către editor. Interacțiunea directă între editorul 2D și utilizator se reduce la selecția unui element dintr-o mulțime de elemente ce au aceleași proprietăți.
- *Funcționalitatea*: obiectele ce fac parte din editorul 2D se caracterizează printr-o cuplare funcțională strânsă, demonstrată de următoarele observații:
  - modificări ale unui obiect din cadrul editorului 2D vor avea ca efect modificări la nivelul altor obiecte din cadrul său
  - obiectele comunică cu același actor
  - obiectele aflate în relație comunică cu același actor (rezultă din utilizarea configurației FORK în cadrul diagramelor de trasare a evenimentelor)
  - un obiect efectuează mai multe operații asupra altor obiecte
- *Comunicarea*: subsistemul gravitează în jurul obiectului de control `Editor2D`. De acest obiect sunt strâns legate toate celelalte obiecte de interfață și entitate din cadrul subsistemului.
- *Actorii*: Numărul total de actori cu care interacționează sistemul este doi, din care rolul preponderent îl are mediul gazdă, prin intermediul căruia are loc și interacțiunea cu cel de-al doilea actor, utilizatorul. Este întrunit astfel criteriul de minimizare a numărului de actori care interacționează cu un subsistem.

4.3.4.3.4 Analiza cerințelor software

Analiza orientată spre obiecte a cerințelor software parcurge următorii pași:

- *Rafinarea scenariilor*: sunt rafinate și detaliate scenariile generate la nivelul analizei cerințelor sistemului. În cazul sistemului Editor2D, în cadrul căruia componenta

BUPT

predominantă este cea software, scenariile determinate în etapa de analiză a cerinţelor sistemului sunt preluate fără modificări în etapa de analiză a cerinţelor software.

- *Identificarea claselor şi a obiectelor*: clasele şi obiectele identificate în etapele parcurse până în acest moment reprezintă baza pentru etapa de analiză a cerinţelor sistemului. Clase şi obiecte suplimentare mai pot apărea în etapa de proiectare software.
- *Identificarea atributelor şi a operaţiilor*: pentru fiecare clasă sau obiect determinat până în prezent, atributele şi operaţiile reprezintă elementul suplimentar de detaliere adus de analiza cerinţelor software.

Atributele şi operaţiile principalelor obiecte din cadrul editorului 2D vor fi prezentate în cele ce urmează.

Linia de comandă:

LiniaDeComandă
dimensiune poziţie atribute textBuffer caractereAcceptate poziţieCursor stivaStări
setareParametri citireParametri procesareCaracter citireText afişareText

BUPT

Mouse-ul:

Mouse
stareMouse
CitireStare afişarePointer ascunderePointer citirePoziţie setarePoziţie citireStareButoane resetStareButoane citireDeplasare

Cursorul:

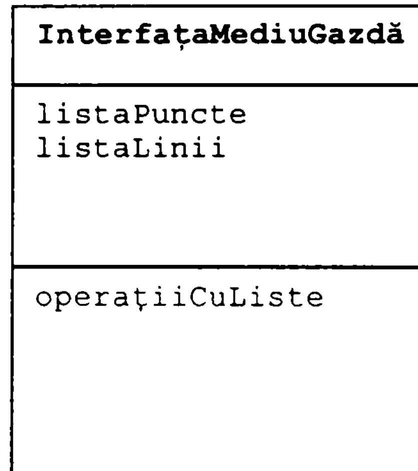
Cursor
StareCursor dimensiuneReticul coordonateCursor zonaDeplasare fcţDesenPeCursor
setareParametri afişareCursor ascundereCursor deplasareCursor

Interfaţa utilizator:

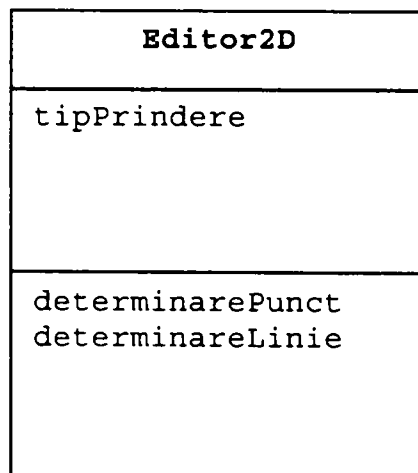
InterfaţaUtilizator
punctPrecedent punctCurent
generarePunct confirmare

BUPT

Interfaţa cu mediul gazdă:



Editorul 2D:



- *Modelului obiectelor*: reprezintă o detaliere a diagramei de relaţii între entităţi prezentată anterior:

BUPT

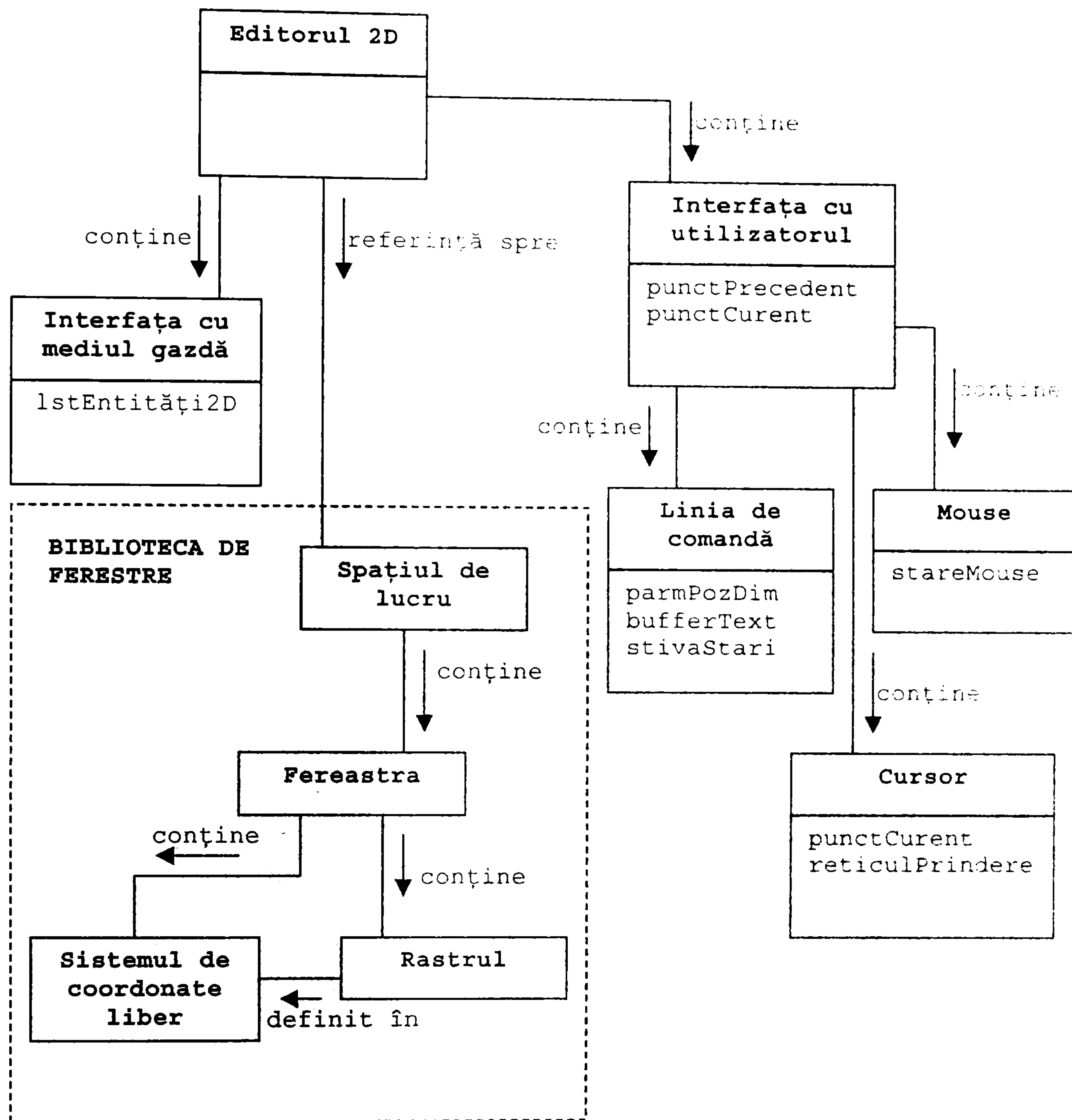


Figura 4.14 Modelul obiectelor pentru editorul 2D

- *Evaluarea claselor și a obiectelor*: se concentrează în această fază asupra proprietăților claselor și a obiectelor, cum ar fi completitudine, dimensiune, nivel de detaliu, atribute și operații asociate. Fişa pentru obiectul LinieComandă este prezentată în continuare:

Identificator obiect: LinieComandă		**Cod sursă:** CdhpCommandLine	
Element revizuit		**Valoare**	**Observații**
1.	**Categoria obiectului**	obiect entitate	
2.	**Completitudine**	obiectul modelează totalitatea	

		comenzilor generate prin intermediul tastaturii pentru generarea unui punct în fereastra activă.	
3.	**Dimensiune**		

4.	**Atribute**	`poziţie pe ecran`
		`dimensiune`
		`buffer text`
		`stiva de stări`
			gramatica ce guvernează linia de comandă este implementată prin intermediul unui automat finit determinist

5.	**Operaţii**		
6.	**Identificator scenariu**	„Generare punct în fereastra activă”	
7.	**Nivel de detaliu**		
8.	**Volatilitate**	obiect tranzient	
9.	**Dependenţa de implementare**		
	hardware	independent de platformă	
	sistem de operare	independent de sistemul de operare	singura componentă dependenta de sistemul de operare este funcţia de citire a unui caracter de la tastatură
	DBMS	–	
10.	**Subsistem candidat**	Editor 2D	
11.	**Agregare**	agregat în cadrul obiectului `UserInput`	
12.	**Asociere**		

4.3.4.3.5 Proiectarea software

Etapa de proiectare software constă pentru editorul 2D din abordarea structurii de moştenire şi de agregare, stabilirea interfeţelor claselor, tratarea excepţiilor şi evaluarea proiectării.

4.3.4.3.5.1 Modelul de moştenire şi agregare

Iniţiind etapa de proiectare software, faza de tranziţie de la analiza orientată spre obiecte transformă relaţiile între clase şi obiecte determinate în etapa de analiză în relaţii de moştenire şi agregare. Figura următoare ilustrează modelul de moştenire şi agregare pentru editorul 2D:

BUPT

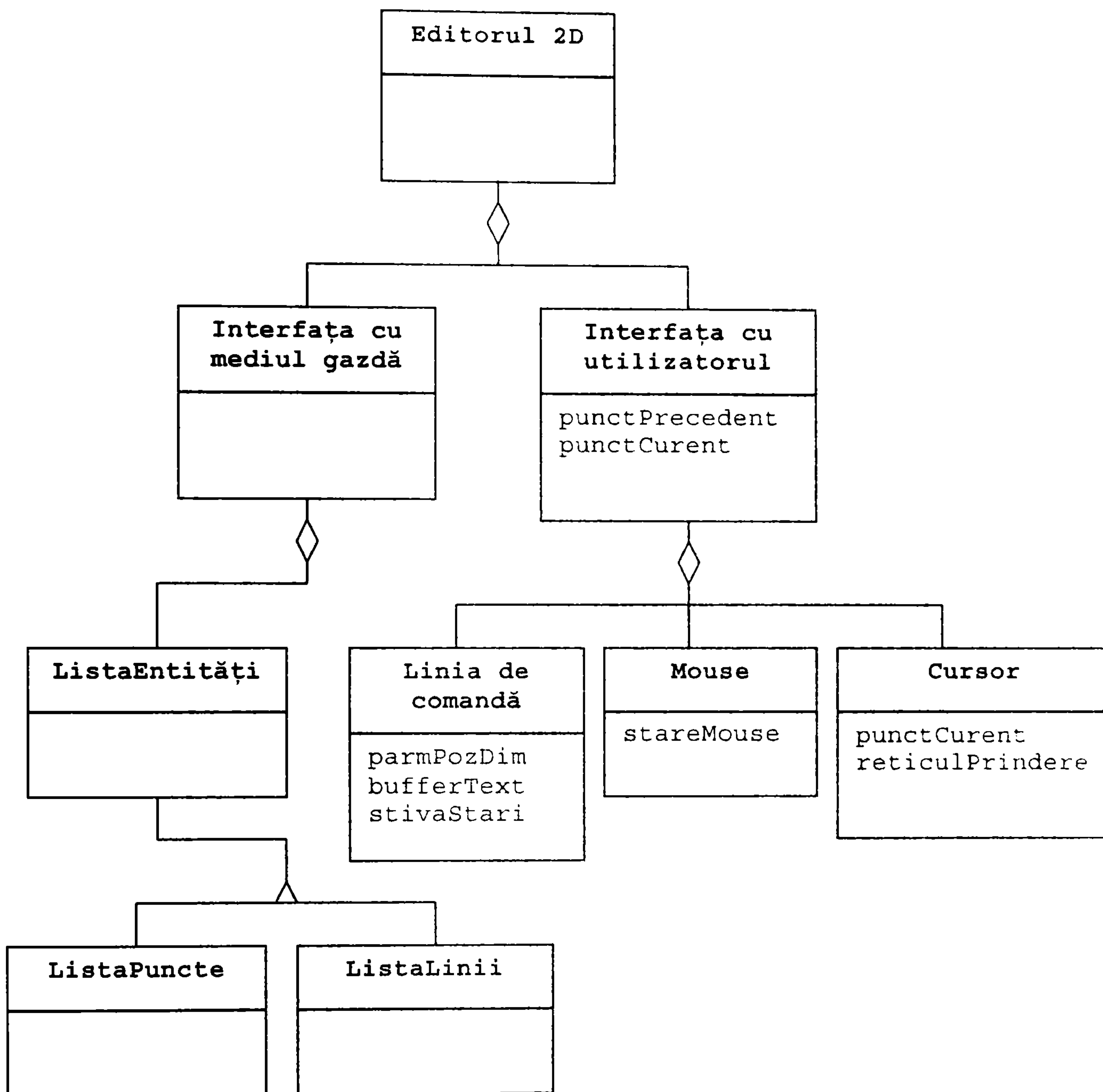


Figura 4.15 Modelul de moştenire şi agregare pentru editorul 2D

Cele două clase `ListaPuncte` şi `ListaLinii` agregate în cadrul interfeței cu mediul gazdă ilustrează dinamica procesului de proiectare, clase noi pot deci să apară pe întreg parcursul procesului de dezvoltare a aplicației. Actualmente entitățile cu care lucrează editorul 2D sunt punctele şi liniile.

Una dintre posibilitățile de dezvoltare a produsului constă în lărgirea mulțimii tipurilor de entități 2D cu care se lucrează, şi anume cercuri şi arce de cerc, texte, cote, cote de nivel etc. Agregarea în cadrul interfeței cu mediul gazdă a listelor de entități reprezintă o facilitate deosebită în procesul de dezvoltare incrementală. Facilitățile suplimentare pot fi adăugate prin agregarea de noi clase la nivelul interfeței cu mediul gazdă. Relația de moştenire ilustrată între lista de entități

BUPT

şi listele de puncte respectiv linii va putea fi implementată utilizând paradigma de metaclasă a cărei suport se regăseşte în cadrul limbajului C++ prin intermediul noţiunii de *template*.

4.3.4.3.5.2 Stabilirea interfeţelor claselor

Criteriile avute în vedere pentru stabilirea interfeţei unei clase au fost analizate în detaliu în cadrul capitolului anterior. Rezultatul aplicării acestora este prezentat în continuare pentru clasa `LinieComandă`:

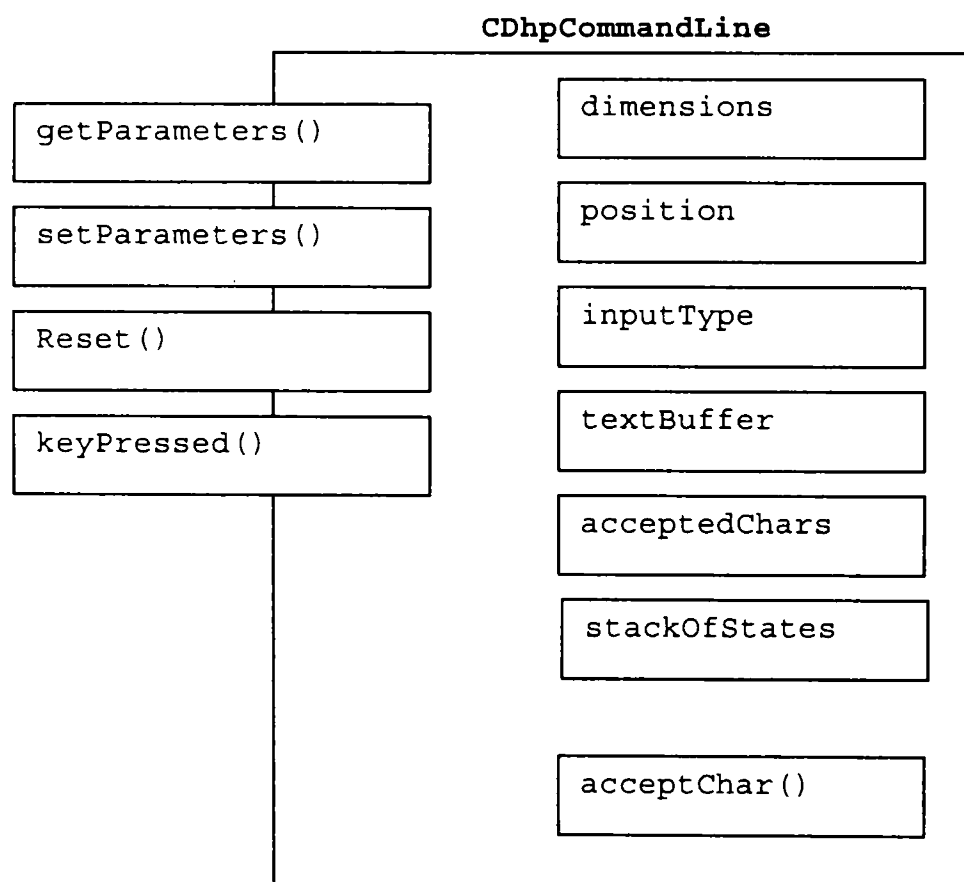


Figura 4.16 Obiectul încapsulat "linie de comandă"

4.3.4.3.5.3 Proiectarea claselor pe baza diagramei de tranziţie a stărilor

Implementarea cerinţelor sistemului referitoare la linia de comandă se bazează pe diagrama de tranziţie a stărilor reprezentată de automatul finit determinist obţinut în etapa de analiză a cerinţelor sistemului. O metodă de transpunere a unui automat finit într-o ierarhie de clase este prezentată din punct de vedere teoretic în [NIE95] şi [FAI93]. Pentru cazul concret al liniei de comandă a cărei diagramă de tranziţie a stărilor este prezentată în figura 4.13 această ierarhie de clase este prezentată în continuare.

BUPT

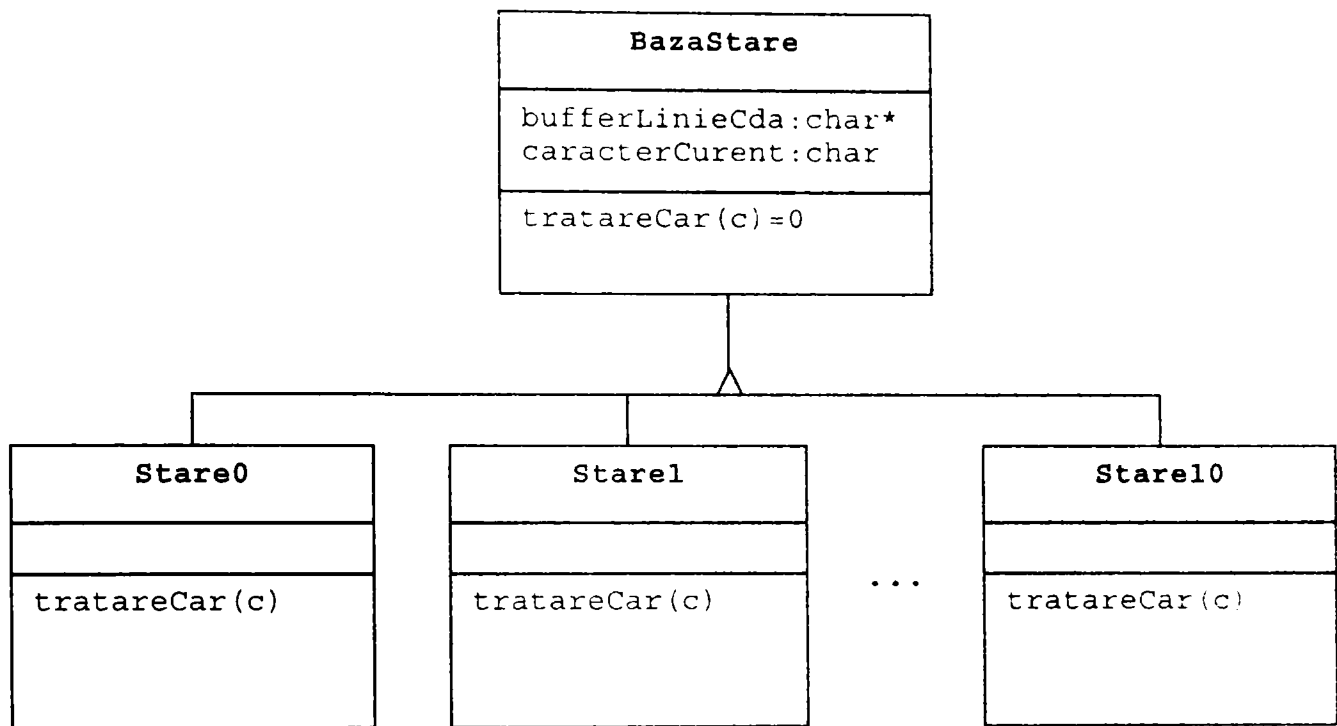


Figura 4.17 Ierarhia de clase rezultată din diagrama de tranziție a stărilor pentru linia de comandă

În cadrul clasei de bază a ierarhiei din figura de mai sus sunt plasate toate datele membre comune tuturor stărilor. Aceste date sunt:

- `*bufferLinieCda`, reprezentând adresa șirului de caractere afișat momentan în linia de comandă
- `caracterCurent`, reprezentând ultimul caracter introdus, ce va determina tranziția automatului finit din starea curentă în altă stare.

Metoda `tratareCar()` prezentă în cadrul clasei de bază reprezintă o metodă pur virtuală, ce determină ca și clasa `BazaStare` să fie o clasă pur virtuală. Metoda va fi redefinită în cadrul fiecărei clase `Stare`*I* derivată din clasa de bază. Fiecare dintre aceste metode `tratareCar()` va procesa caracterele ce determină tranziția din starea curentă în altă stare.

Această soluție aleasă pentru implementarea liniei de comandă s-a dovedit a fi deosebit de avantajoasă din următoarele puncte de vedere:

- Funcționalitatea specificată în cadrul cerințelor sistemului poate fi simulată deja la nivelul modelului dinamic, evitând astfel surprize în faza de testare a produsului software.
- Flexibilitatea acestei soluții rezultă din ușurința cu care se poate modifica sintaxa liniei de comandă. Modificările în cadrul modelului în etapa de proiectare și a codului rezultat sunt minime. Acestea se rezumă doar la ierarhia de clase prezentată mai sus.

BUPT

4.3.4.3.5.4 Tratarea excepțiilor

Mecanismul de tratare a excepțiilor este bazat ca şi în cazul bibliotecii de ferestre pe utilizarea conceptului de programare bazată pe contract implementată prin intermediul mecanismului bazat pe aserțiuni prezentat în detaliu în cadrul capitolului anterior. Codul sursă al clasei `LinieComandă` este prezentat în cadrul anexei.

4.3.4.3.5.5 Evaluarea proiectării

Evaluarea proiectării are loc pe baza criteriilor stabilite în cadru capitolului anterior. Conform acestora, o clasă trebuie să încapsuleze o singură abstractizare şi să reflecte un set bine definit de cerințe.

Următoarele aspecte sunt urmărite pentru a evalua procesul de proiectare în ceea ce priveşte clasa `LinieComandă`:

- *Categoria clasei*: conform clasificărilor descrise, clasa se înscrie în categoria *obiect entitate* a clasificării Jacobson, respectiv *clasă computațională* conform clasificării bazate pe funcționalitate.
- *Încapsularea şi nivelul de abstractizare*: abstractizarea încapsulată de către clasa `LinieComandă` reprezintă linia de comandă disponibilă utilizatorului pentru a introduce de la tastatură coordonatele punctului ce se doreşte a fi introdus.
- *Moştenirea*: clasa `LinieComandă` nu face parte din nici o ierarhie de moştenire în cadrul editorului 2D. Interfața ei a fost pregătită pentru moştenire, astfel că ea poate deveni clasă de bază pentru o specializare ce va satisface particularitățile unui anumit mediu gazdă.
- *Agregarea*: clasa `LinieComandă` este agregată în cadrul clasei `UserInput`, aceasta conținând pe lângă clasa `LinieComandă` şi clasa `Mouse` respectiv `Cursor`. Este satisfăcută relația de tipul „has-a" între clasa `UserInput` şi clasa `LinieComandă`.
- *Operații:* operațiile asociate clasei `LinieComandă` formează un set complet pentru abstractizarea încapsulată, date fiind cerințele sistem asociate acesteia.
- *Atribute:* atributele reprezintă elemente de date asociate instanțelor clasei. Nu există atribute de dimensiuni mari ce ar putea fi candidați pentru clase individuale în urma unei revizuiri a proiectării.

BUPT

4.3.4.4 Implementarea

Procesul de transformare a produselor de lucru specifice etapei de proiectare în cod sursă are loc conform schemei prezentate în cadru capitolului anterior. Codul sursă al interfeței clasei `CdhpCommandLine` se găsește în cadrul anexei.

4.4 Concluzii. Contribuții

Actualul capitol prezintă două aplicații ale căror proces de dezvoltare se bazează pe metoda de dezvoltare software descrisă în cadrul capitolului anterior. Prima aplicație, *biblioteca de ferestre*, reprezintă o bibliotecă de clase ce se constituie într-un server ce pune la dispoziție un set de servicii pentru o aplicație client fără a interacționa direct cu utilizatorul. A doua aplicație, *editorul 2D*, reprezintă un exemplu de aplicație în cadrul căreia interacțiunea cu utilizatorul ocupă o pondere semnificativă.

Dezvoltarea aplicațiilor urmărește etapele de analiză a domeniului, analiză a cerințelor sistemului, proiectarea sistemului, analiza cerințelor software, proiectare software și implementare. O atenție deosebită este acordată delimitării clare a acestor etape din punct de vedere al conținutului, păstrând o notație unitară care să permită urmărirea evoluției entităților de analiză și proiectare pe întreg parcursul procesului de dezvoltare software.

Unul dintre elementele pe care se bazează metoda de dezvoltare software utilizată este reprezentat de *utilizarea scenariilor*:

- *Biblioteca de clase*: Se demonstrează practic faptul că dezvoltarea unei biblioteci de clase nu necesită analiza bazată pe scenarii, dată fiind lipsa interacțiunii directe cu utilizatorul. În cazul bibliotecii de clase configurația de claselor și a interfețelor acestora este dată direct de setul de servicii pe care trebuie să le pună la dispoziție serverul clienților săi.
- *Editorul 2D*: natura aplicației permite o abordare bazată pe scenarii începând chiar din faza analizei cerințelor sistemului. Scenariile definite în această fază vor fi rafinate in cadrul analizei cerințelor software pentru a oferi un ghid pentru stabilirea configurației claselor și pentru implementarea comportamentului dorit la nivelul aplicației.

O atenție deosebită este acordată evaluării proiectării. Sunt aplicate atât criteriile de evaluare calitative enumerate pe parcursul capitolului anterior cât și criterii cantitative, cum ar fi metrica LCOM. Rezultatele aplicării acestor criterii demonstrează justețea modelului ales pentru implementarea cerințelor definite în cadrul aplicației.

BUPT

Tratarea excepțiilor reprezintă un element căruia activitatea de proiectare îi acordă o importanță deosebită. Abordarea aleasă în cadrul editorului 2D este reprezentată de programarea pe bază de contract propusă de Bertrand Meyer. Acest concept a fost transpus într-un set de macrodefiniții menit să asigure consistența utilizării serviciilor oferite clienților de către clasele ce compun sistemul.

Întregul proces de dezvoltare a aplicațiilor a urmărit conformitatea cu elementele de bază ale orientării spre obiecte, abstractizarea datelor, încapsularea, ascunderea informației, reutilizarea prin generalizare / specializare.

Aplicarea tehnologiei orientate spre obiecte a permis asigurarea unei calități și a unei mentenabilități ridicate a aplicațiilor, cerințe ce ar fi fost mult mai dificil de îndeplinit utilizând metodele structurate.

Contribuțiile originale prezente în cadrul prezentului capitol sunt:

- Prezentarea sistemului CAD/CAM *DHP-Bauwerk* localizând contextul aplicațiilor *Bibliotecă de ferestre* și *Editor 2D*, a căror proces de dezvoltare software orientat spre obiecte va fi urmărit în detaliu.
- Determinarea unui set inițial de clase și obiecte pentru cele două aplicații, pe baza regulilor specifice etapei de analiză a domeniului.
- Organizarea informației legate de clasele și obiecte generate în etapa de analiză este prezentată sub forma unor fișe ce atașează fiecărui element revizuit un anumit conținut precum și observațiile necesare pentru următoarele etape ale procesului de dezvoltare.
- Abordarea etapei de analiză a cerințelor sistemului pe bază de scenarii pentru *Editorul 2D*:
  - Identificarea actorilor și a cazurilor de uz necesare implementării cerințelor aplicației
  - Detalierea succesivă a specificației unui caz de uz sub formă de:
    - descriere narativă
    - tabel decizional
    - diagramă de trasare a evenimentelor
- Utilizarea diagramelor de tranziție a stărilor pentru a completa structura de clase și obiecte specifice entităților cu caracter dinamic pronunțat:
  - Descrierea și aplicarea unei metode de generare a unei diagrame de tranziție a stărilor pentru entitatea *Linie de comandă* utilizând metode specifice limbajelor formale și teoriei compilatoarelor.

BUPT

- Aplicarea criteriilor de partiționare și de configurare specifice etapei de proiectare a sistemului pentru a demonstra că atât *Biblioteca de ferestre* cât și *Editorul 2D* reprezintă sisteme compacte ce nu necesită divizarea în subsisteme.
- Abordarea diferențiată a etapei de analiză a cerințelor software pentru cele două aplicații:
  - Generarea diagramei de relații între entități ca element de legătură între analiză și proiectare pentru *Biblioteca de ferestre*.
  - Rafinarea scenariilor definite în etapa de analiză a cerințelor sistemului pentru *Editorul 2D*.
- Condensarea informației aferente etapei de analiză a cerințelor software în cadrul *modelului obiectelor* pentru determinarea atributelor și a operațiilor claselor și obiectelor prezente în cadrul sistemului.
- Abordarea unitară a evaluării activității de analiză și proiectare: în etapa de analiză a cerințelor software este utilizat același format de fișă ca și în cazul analizei cerințelor sistemului.
- Generarea *modelului de agregare și moștenire* pentru *Biblioteca de ferestre* și pentru *Editorul 2D* în etapa de proiectare software. Justețea generării în această etapă a acestui model este susținută de faptul că entităților lumii reale rezultate în etapele de analiză le este atașată suficientă informație pentru a genera o ierarhie de clase care să satisfacă cerințele sistemului.
- Aplicarea considerațiilor specifice subpasului OOD al etapei de proiectare software pentru generarea interfețelor claselor *bibliotecii de ferestre* și *editorului 2D*.
- Aplicarea unei soluții orientate spre obiecte pentru rezolvarea problemei analizei lexicale în cadrul liniei de comandă a *Editorului 2D*. Soluția se bazează pe o ierarhie de clase care implementează un automat finit determinist obținut din expresia regulată ce definește limbajul liniei de comandă.
- Implementarea unui mecanism de tratare a excepțiilor bazat pe conceptul de „programare pe bază de contract" al lui Bertrand Meyer, atât pentru *Biblioteca de ferestre* cât și pentru *Editorul 2D*.
- Analiza calitativă și cantitativă a procesului de proiectare:
  - Aplicarea metricii software LCOM pentru evaluarea cantitativă a procesului de proiectare a claselor și obiectelor.
  - Urmărirea criteriilor de evaluare calitativă a proiectării prezentate în cadrul capitolului anterior.

BUPT

- Implementarea în limbaj C++ a modelului de analiză şi proiectare generat pentru aplicaţiile *Bibliotecă de ferestre* şi *Editor 2D*. Un extras din codul sursă al acestor aplicaţii se regăseşte în cadrul anexei.

BUPT

Concluzii şi perspective

5.1 Concluzii

Prezenta teză de doctorat reprezintă o pledoarie în favoarea abordării întregului proces de dezvoltare al unei aplicaţii pe baza tehnologiei orientate spre obiecte. Principalele elemente pentru care contribuţia orientării spre obiecte este decisivă sunt asigurarea calităţii aplicaţiei, mentenabilitatea acesteia şi potenţialul pentru dezvoltări ulterioare.

Paradigmele orientării spre obiecte pe care se bazează întregul proces de dezvoltare software abordat în cadrul actualei teze sunt următoarele: clasele şi obiectele, încapsularea şi ascunderea informaţiei, abstractizarea datelor şi genericitatea, responsabilităţile, colaborările şi transmiterea de mesaje, moştenirea şi agregarea, polimorfismul, legarea statică şi dinamică şi modularitatea.

Câteva dintre cele mai cunoscute metode de analiză şi proiectare orientate spre obiecte au fost analizate în cadrul prezentei lucrări pentru a identifica elemente care vor constitui baza unei metode proprii de analiză, proiectare şi implementare software. Aceste metode au la bază un set de modele menite să descrie sistemul din punct de vedere static, dinamic şi funcţional. Fiecare metodă propune o abordare completă a procesului de dezvoltare software. Neajunsuri ale acestor metode pot fi considerate în unele cazuri realizarea insuficientă a separaţiei între etapele succesive ale procesului de dezvoltare software, tratarea insuficientă a unor aspecte legate de implementarea în sisteme distribuite sau în contextul aplicaţiilor de timp real. Notaţiile utilizate pentru reprezentarea entităţilor de analiză şi proiectare şi a relaţiilor dintre acestea diferă de la metodă la metodă.

O rezolvare posibilă a problemei lipsei de unitate în ceea ce priveşte notaţiile utilizate este propusă de către grupul OMG prin intermediul limbajului unificat de modelare UML. Acesta reprezintă un semnificativ pas înainte în sensul standardizării şi a unificării în ceea ce priveşte informaţia prelucrată în procesul de analiză şi proiectare.

Actuala teză de doctorat propune o abordare riguroasă şi unitară a procesului de dezvoltare a unei aplicaţii software. Conceptele orientării spre obiecte reprezintă elemente centrale care au marcat configuraţia acestui proces pe parcursul tuturor etapelor sale. Încercând să grupeze câteva dintre cele mai valoroase elemente caracteristice unor metode de dezvoltare software consacrate, procesul de dezvoltare software prezentat încearcă să ofere o cale cât mai directă şi mai uşor de

BUPT

urmărit de la un set de cerinţe enunţate pentru o aplicaţie şi implementarea acestora într-un limbaj de programare orientat spre obiecte.

Utilizarea practică a metodei de dezvoltare software propuse în cazul concret al realizării componentelor aplicaţiei *DHP-Bauwerk* reprezintă o probă a viabilităţii acesteia, subliniind încă o dată faptul că orientarea spre obiecte reprezintă soluţia pentru dezvoltarea unor aplicaţii stabile, fiabile, având un grad înalt de mentenabilitate şi un potenţial considerabil pentru dezvoltări ulterioare.

5.2 Contribuţii

Contribuţia personală în cadrul actualei teze de doctorat este legată de realizarea unei metode de dezvoltare software ca urmare a necesităţii implementării unor componente în cadrul sistemului *DHP-Bauwerk* al firmei Dietrich's AG. Această metodă de dezvoltare a rezultat în urma unui studiu bibliografic amănunţit al unor metode de dezvoltare software consacrate şi introducerea unor tehnici originale, urmărind configurarea procesului de dezvoltare software pentru a permite realizarea de aplicaţii fiabile, uşor de întreţinut şi de dezvoltat, reducând la minimum factorii de risc şi timpul necesar dezvoltării acestora.

Aplicarea în practică a acestei metode este exemplificată pe baza procesului de dezvoltare a două dintre componentele aplicaţiei *DHP-Bauwerk* a firmei Dietrich's AG, *biblioteca de ferestre* şi *editorul 2D*.

Abordările şi sintezele originale legate de faza de studiu bibliografic conţinut în capitolul al doilea sunt:

- *Identificarea şi definirea principalelor paradigme ale orientării spre obiecte* urmărind modul în care acestea îşi găsesc suport nativ la nivelul implementării în cadrul limbajului de programare C++.
- *Identificarea definiţiilor conceptului de clasă şi a celui de obiect* subliniind conexiunea cu paradigmele enunţate anterior.
- *Prezentarea unor modalităţi de creare a claselor şi a obiectelor* în contextul unei aplicaţii.
- *Prezentarea principalelor modele orientate spre obiecte* ce descriu un context al obiectelor unei aplicaţii din punct de vedere static, dinamic şi funcţional.
- *Realizarea unui studiu critic comparativ al câtorva din cele mai cunoscute metode de dezvoltare software orientate spre obiecte*, analizând în cadrul fiecăreia dintre acestea

BUPT

etapele de analiză şi proiectare orientate spre obiecte pornind de la exemplul oferit de aplicația ATM.

- *Analiza măsurii în care diversele metode oferă sprijin pentru implementarea sistemelor de timp real şi a aspectelor legate de procese concurente.*
- *Identificarea unor instrumente CASE* care implementează aceste metode automatizând procesul de dezvoltare software.

Metoda de dezvoltare software propusă în cadrul capitolului al treilea abordează toate etapele dezvoltării unei aplicații, începând cu analiza orientată spre obiecte şi continuând cu proiectarea şi implementarea bazate pe tehnologia orientată spre obiecte. Procesul de dezvoltare a acestei metode a fost ghidat de următoarele idei:

- *Abordarea procesului de proiectare prin rafinări succesive*, insistând asupra posibilității de a urmări deciziile de proiectare din etapa de analiză până în etapa de implementare.
- *Posibilitatea de a urmări conceptele specifice programării orientate spre obiecte pe parcursul întregului proces de dezvoltare.* Aceste concepte au fost utilizate atât în procesul de creare a claselor şi obiectelor de analiză şi proiectare, cât şi la transformarea acestora în clase şi obiecte ale etapei de implementare în limbajul C++.
- *Completarea procesului de dezvoltare cu elemente specifice proiectării structurate* pentru a compensa lipsa unei abordări suficient de riguroase a aspectelor legate de soluții distribuite şi de operare în timp real în cadrul metodelor consacrate de dezvoltare orientate spre obiecte.
- *Generarea unor entități reutilizabile* care să satisfacă într-o proporție cât mai mare conceptele legate de abstractizarea datelor, încapsulare şi ascundere a informației.
- *Evaluarea riscului pe care îl impune abordarea orientată spre obiecte* în cadrul fiecăreia dintre etapele de dezvoltare ale aplicației.
- *Unificarea tratării deciziilor de proiectare la nivel hardware şi la nivel software* pentru a obține o abordare unitară în conformitate cu similitudinea existentă între programarea orientată spre obiecte şi structura modulară a sistemelor de calcul.
- *Verificarea transpunerii corecte în cod a deciziilor luate în etapele de analiză şi proiectare* prin definirea unui mecanism puternic de tratare a excepțiilor bazat pe aserțiuni şi pe conceptul de programare pe bază de contract.
- *Aplicarea unor criterii de evaluare a claselor şi obiectelor* generate în fiecare fază a procesului de dezvoltare software. Criteriile calitative aplicate în fazele de analiză şi

BUPT

proiectare, sunt completate de metrici software ce oferă o evaluare cantitativă a justeţii soluţiei în cadrul procesului de dezvoltare software.

Capitolul al patrulea are un caracter preponderent practic, în cadrul său fiind ilustrat modul de aplicare a metodei de dezvoltare prezentate în capitolul al treilea pentru realizarea unor componente software din cadrul unui sistem CAD/CAM, aplicaţia *DHP-Bauwerk*. Componentele ale căror dezvoltare se urmăreşte în cadrul acestui capitolul sunt:

- *Biblioteca de ferestre*, reprezentând un exemplu de componentă software pentru care nu există interacţiune directă cu utilizatorul. Acest fapt determină o anumită succesiune a etapelor procesului de dezvoltare software, etape care sunt specifice în general bibliotecilor de clase.
- *Editorul 2D*, reprezentând o componentă caracterizată prin interacţiune semnificativă cu utilizatorul, a cărei dezvoltare se bazează în mod esenţial pe conceptele de caz de uz şi de scenariu.

Fiecare etapă a aplicării procesului de dezvoltare software este încheiată cu o evaluare calitativă sau cantitativă a activităţii de analiză şi proiectare pentru a identifica şi remedia în fază cât mai timpurie eventualele erori. Robusteţea aplicaţiilor este asigurată de către un mecanism performant de tratare a excepţiilor.

Anexa prezintă codul sursă al interfeţelor celor două componente software precum şi o ilustrare a modului în care sunt integrate cele două componente în cadrul aplicaţiei *DHP-Bauwerk*.

5.3 Perspective

Metoda de analiză şi proiectare abordată atât din punct de vedere teoretic cât şi practic în cadrul lucrării reprezintă soluţia aleasă pentru implementarea componentelor sistemului *DHP-Bauwerk*.

Perspectivele demersului prezentat se înscriu în două direcţii principale. În primul rând este vorba de dinamica deosebită a domeniului dezvoltării software orientate spre obiecte. Asistăm în momentul de faţă la apariţia unui număr din ce în ce mai mare de instrumente CASE care asistă procesul de analiză şi proiectare software. Un numitor comun al tuturor acestor instrumente software devine limbajul unificat de modelare UML, limbaj promovat de *Object*

BUPT

Management Group şi care cunoaşte în continuare un proces de evoluţie, de perfecţionare, bazat pe experienţa rezultată din practica analizei şi a proiectării din cele mai diverse domenii ale ingineriei software. În acest context, metoda de analiză şi proiectare prezentată în lucrare va cunoaşte o evoluţie continuă în sensul apropierii de limbajul unificat de modelare şi de asimilare a experienţei din ce în ce mai vaste acumulată în cadrul comunităţii dezvoltatorilor software.

A doua direcţie de evoluţie a demersului prezentat este legată de dezvoltarea aplicaţiei *DHP-Bauwerk.* Aplicaţia se doreşte a deveni o soluţie completă pentru realizarea de clădiri cu structură de rezistenţa din lemn. Modulele a căror dezvoltare va cunoaşte în viitorul apropiat o dezvoltare intensă sunt *Dachausmittlung* (construcţia de acoperişuri) respectiv *Deckenkonstruktion* (construcţia de casete de plafon). Dinamica dezvoltării întregii aplicaţii va cunoaşte o evoluţie ascendentă dată de experienţa practică acumulată în utilizarea ei în procesul de realizare a structurilor bazate pe materiale lemnoase. În acest context, metoda de dezvoltare software prezentată în cadrul lucrării va fi aplicată în realizarea unui număr din ce în ce mai mare de module software, asigurând astfel o evoluţie a ei în strânsă legătură cu experienţa practică.

BUPT

Anexa

Anexa prezintă codul sursă al definiţiilor claselor ce intră în componenţa bibliotecii de ferestre şi a editorului 2D. Codul prezentat este rezultatul integrării celor două componente în cadrul aplicaţiei *DHP-Bauwerk*, astfel interfeţele claselor prezintă diferenţe şi completări faţă de structura stabilită în cadrul etapelor de analiză şi proiectare parcurse în cadrul capitolului al patrulea.

A.1 Interfeţele claselor din biblioteca de ferestre

```
#ifndef      DHPINCLUDEDHPWIN
#define      DHPINCLUDEDHPWIN
#define PI   3.141592653589793
#include     <Entity2D.h>

class CdhpWindow;
class CdhpGrid;
class CdhpObject;
class CdhpWindow2D;
```

A.1.1 Spatiul de lucru

```
//parametrii de interfaţă ai spaţiului de lucru-----------------------

struct DhpWorkspaceData
  {
  double   workspace_xmm,  //dimensiunile în [mm] ale spaţiului de lucru
           workspace_ymm;

  long     pixel_xmin, //limitele în pixeli ale spaţiului de lucru
           pixel_ymin,
           pixel_xmax,
           pixel_ymax;

  DhpWorkspaceData(void);
  DhpWorkspaceData(
                   long pxmin,   // dim.[pixel] spaţiu de lucru
                   long pymin,
                   long pxmax,
                   long pymax,
                   double wxmm = 0.0, // dim.[mm] spaţiu de lucru
                   double wymm = 0.0
                  );
  DhpWorkspaceData operator=(
                             DhpWorkspaceData &wsd
```

```
                              );
  };


//spatiul de lucru-------------------------------------------------------
class CdhpWorkspace
  {
  protected:
    CdhpList<CdhpWindow>  windowList;      //lista de ferestre
    DhpWorkspaceData      wsd;             //parametrii de interfaţă
    CdhpWindow   *activeWindow;   //pointer spre fereastra activa
    Long         confnumxpixels, //date interne
                 confnumypixels;
    double       pixel_per_xmm,
                 pixel_per_ymm;

  public:
    CdhpWorkspace(void);
    CdhpWorkspace(                     //constructor
                 long pxmin,           //lim.[pixeli] ale spaţiului de
                                       //lucru în fereastra aplicaţiei
                 long pymin,
                 long pxmax,
                 long pymax,
                 double wxmm = 0.0,   //dim. X a spaţiului de lucru [mm]
                 double wymm = 0.0    //dim. Y a spaţiului de lucru [mm]
                );

    CdhpWorkspace(DhpWorkspaceData &wsdata);
    ~CdhpWorkspace();
    void DeleteWindowList(void);  //şterge lista de ferestre
    void SetData( //setara date ale spaţiului de lucru
                 DhpWorkspaceData &wsdata
                );
    void GetData(//citire date ale spaţiului de lucru
                 DhpWorkspaceData &wsdata
                ){wsdata = wsd;}
    long GetConfnumxpixels(void){return confnumxpixels;}
    long GetConfnumypixels(void){return confnumypixels;}
    double GetPixel_per_xmm(void){return pixel_per_xmm;}
    double GetPixel_per_ymm(void){return pixel_per_ymm;}
    void AddWindow(  //adaugă o fereastră la lista de ferestre
                   CdhpWindow *ptrWin
                  );
    CdhpList<CdhpWindow> &GetWindowList(void){return windowList;}
    void SetActiveWindow( //stabilirea ferestrei active
                         CdhpWindow *newActiveWindow
                        );
    CdhpWindow *GetActiveWindow(void);

  private:
    void ResetDataMembers(void);  //iniţializări date membre
    void CalcInternalData(void);  //calcul valori interne ale clasei
  };
```

BUPT

A.1.2 Fereastra

```
//definiţii pentru fereastra-------------------------------------------
typedef enum
  {
  WIN_3D,
  WIN_XY,
  WIN_XZ,
  WIN_YZ
  } DhpWinType;

typedef enum
  {
  ZOOM_MINUS = -1,
  ZOOM_PLUS = 1
  } DhpZoomType;

#define MAX_ZOOMSTUFE 10  //număr maxim de nivele succesive de zoom


//parametrii de interfata ai ferestrei 2D------------------------------
struct DhpWindowData
  {
  DhpWinType  winType;        //tipul ferestrei

  long        pixel_xmin, //lim.[pixeli] fereastră în spaţiul de lucru
              pixel_xmax,
              pixel_ymin,
              pixel_ymax;

  double      xmin,    //extinderea plană a elementelor din fereastra
              xmax,
              ymin,
              ymax;

  DhpWindowData(void);
  DhpWindowData(
                long p_xmin,
                long p_ymin,
                long p_xmax,
                long p_ymax,
                double m_xmin = 0.0,
                double m_ymin = 0.0,
                double m_xmax = 0.0,
                double m_ymax = 0.0
               );
  DhpWindowData operator=(  //operator de atribuire a datelor ferestrei
                         DhpWindowData &dhpwd
                        );
  };

//Fereastra standard---------------------------------------------------
class CdhpWindow
  {
  protected:
    CdhpWorkspace *workspace; //adresa spaţiului de lucru părinte
```

BUPT

```
    DhpWindowData winData;  //parametrii de interfaţă ai ferestrei 2D

//parametrii calculati pentru reprezentarea în fereastra 2D
    double  fl_xmin,  //limitele ferestrei în spatiul de lucru
            fl_xmax,
            fl_ymin,
            fl_ymax;

    long    pixel_links,  //limita stânga-jos a ferestrei în pixeli
            pixel_unten;

    double  fenster_xmm,  //dimensiunile metrice ale ferestrei
            fenster_ymm;

    double  pixel_f_links,  //marginile în pixeli ale ferestrei
            pixel_f_rechts,
            pixel_f_unten,
            pixel_f_oben;

    double  klip_links, //limitrele de decupare ale ferestrei
            klip_rechts,
            klip_unten,
            klip_oben;

    double  mass; //raport pentru transformare din metric în pixeli

    short   flg_welche_zoom_stufe; //nivelul curent de zoom

    double  xrahmmin[MAX_ZOOMSTUFE],  //acoperirea plana a niv. De zoom
            xrahmmax[MAX_ZOOMSTUFE],
            yrahmmin[MAX_ZOOMSTUFE],
            yrahmmax[MAX_ZOOMSTUFE];

  public:
    CdhpWindow(
                CdhpWorkspace *ws
              );
    CdhpWindow(
                CdhpWorkspace *ws,
                DhpWinType type,
                long pxmin,
                long pymin,
                long pxmax,
                long pymax
              );
    virtual ~CdhpWindow(){};
    virtual void CalcPointPers(  //proiectie perspectiva 3D→2D
                                double coord[3],
                                double pers[2],
                                short min_max
                              ){}
    virtual void DrawLine( //desenarea unei linii (metodă pur virtuală)
                            double xa,
                            double ya,
                            double xe,
```

BUPT

```
                        double ye,
                        short farbe,
                        unsigned short lineStyle,
                        short mode
                       ) = 0

    virtual void DrawPoint(  //desenare punct (metodă pur virtuală)
                         double x,
                         double y,
                         short color,
                         short mode,
                         short pointDim
                        ) = 0;
    DhpWinType GetType(void){return winData.winType;}
    short IsActive(void);
    void SetData(  //setarea date fereastră
                DhpWindowData &wdata
               );
    void GetData(  //citire date fereastră
                DhpWindowData &wdata
               );
    double GetMass(void){return mass;}
    void SetHotBorder(  //setare variabila pentru defilare fereastră
                     short newHotBorder
                    ){hot_border = newHotBorder;}

//setarea parametrilor pentru fereastra
    void SetExtBoundingBox( //setare acoperire plană
                         double x_min, //noile valori ptr. Bbox
                         double y_min,
                         double x_max,
                         double y_max
                        );
    virtual void CalcParameters(void);  //calcul parametri pt. Desenare
    void CalcParameters( //alinierea reprezentării cu fereastra XZ
                       CdhpWindow *sourceWindow
                      );
    short Zoom(
              DhpZoomType sign,  //plus / minus
              long pixel_x1 = 0, //coord.[pixeli] ale ramei lupei
              long pixel_y1 = 0,
              long pixel_x2 = 0,
              long pixel_y2 = 0
             );
    short Zoom(..//zoom prin aliniere cu fereastra sursă
              DhpZoomType sign,  //plus / minus
              CdhpWindow *sourceWindow
             );
    short ZoomPossible(    //testează dacă operaţia zoom este posibila
                      DhpZoomType sign
                      );
    short OnBoundary( //test şi modificare acoperire plană dacă
                     //cursorul este în zona sensibilă pentru defilare
                    long *x,  //coordonatele curente ale cursorului
                    long *y
                   );
```

BUPT

```
//conversii
    virtual void WorkspacePixelToPoint( //conversie pixel → metric
                                       double &pointX,  //coord.metrice
                                       double &pointY,
                                       long pixelX,     //coord.pixeli
                                       long pixelY
                                      );
    virtual void PointToWorkspacePixel( //conversie metric → pixel
                                       long &pixelX,  //coord.pixeli
                                       long &pixelY,
                                       double pointX, //coord. Metric
                                       double pointY
                                       );
    void WorkspaceToScreen(//conversie pixel workspace → ecran
                            long &px,
                            long &py
                           );
    void ScreenToWorkspace(//conversie pixel ecran → workspace
                            long &px,
                            long &py
                            );
    void PixelToMetricDistance(//conversie distanţă pixeli → metric
                                double &mdistance, //distanta metrica
                                long pdistance,    //distanta in pixeli
                                short coord=0      //coordonata ptr care
                                                   //se calc. Distanta
                               );
    void MetricToPixelDistance(//conversie distanţă metric→ pixeli
                                long &pixelDist,  //distanţa în pixeli
                                double metricDist //distanţa metrică
                               );
    long klip(  //decupare 2D
              double &x1, //limitele ferestrei de decupare
              double &y1,
              double &x2,
              double &y2
             );
    virtual void CalcPersAndDraw( //calculul coordonate 2D şi desenare
                                 short calc,
                                 short draw
                                ) = 0;
    virtual void Draw( // desenarea ferestrei şi a conţinutului ei
                      short all_refresh,
                      short draw
                     ) = 0;
    virtual void ModifyWindow(// modificare dimensiuni fereastră
                              long pxmin, //noile coordonate [pixeli]
                              long pymin,
                              long pxmax,
                              long pymax,
                              short draw = TRUE
                             ){};
    CdhpWorkspace *GetWorkspace(void){return workspace;};
```

BUPT

```
  private:
    static double klip_precision; //"ε" ptr.bordare limite decupare
    static double bbox_border;  //margine de bordare a reprezentării
                                //din fereastră
    static short hot_border; // lăţimea zonei sensibile ptr defilare;
    void ResetDataMembers(void);
    long masstab(void);
    long fenster( //conversie sist.coord.2Dcontinuu → sist.coord.ecran
                  double fenster_links,
                  double fenster_rechts,
                  double fenster_unten,
                  double fenster_oben
                 );
    void CalcClippingRegion(void);
  };
```

A.1.3 Rastrul

```
//definiţii pentru rastru---------------------------------------------
typedef enum{
             GRID_ACTIV,
             GRID_INACTIV,
             GRID_INACTIV_BUT_VISIBLE
            } DhpGridState;

//Rastrul-------------------------------------------------------------
class CdhpGrid
  {
  private:
    double stepX,
           stepY;  //pasul metric între punctele rastrului
    short  colour; //culoarea puctelor rastrului
    short  drawMode; //modul de desenare a rastrului
    double crtPoint[2]; //coordonatele punctului curent al rastrului
    short  stateCounter; //contorul de stări a rastrului
    DhpGridState  state; //starea curentă a rastrului
    DhpGridState  prevState; //starea precedentă a rastrului
    CdhpWindow2D  *parentWindow; //fereastra părinte a rastrului
    struct wndDta{//extinderea porţiunii vizibile a rastrului
      double LdownG[2];
      double LupG[2];
      double RdownG[2];
      double RupG[2];
    }wData;

  public:
    CdhpGrid(
             double sx = 1.0, //setarea implicită pentru stepX
             double sy = 1.0, //setarea implicită pentru stepY
             short  dm = 3,   //modul de desenare a rastrului
             DhpGridState st = GRID_INACTIV //starea rastrului
            );
    void SetGrid(
                 double sx, //setarea pentru stepX
```

BUPT

```
                double sy, //setarea pentru stepY
                DhpGridState st //setarea stării rastrului
               );
  void GetGrid(
                double &sx, //stepX
                double &sy, //stepY
                DhpGridState &st //starea rastrului
               );
  short  GetColor(void){return this->colour;};
  double GetPrecision(){return precision;}
  void   SetCrtPoint(double crtP[2]);
  void   GetCrtPoint(double crtP[2]);
  void   Draw() = 0;
  void   SetParent(CdhpWindow2D *win){parentWindow = win;};
 private:
  static double precision; //precizia la comparaţia a două puncte
                           //în rastru
  void DrawColumnOfGridPoints(
                              double Ymin,
                              double Ymax,
                              double Xcrt,
                              wndDta *winData
                             ) = 0;
 };
```

A.1.4 Sistemul de coordonate liber

```
//sistemul de coordonate liber 2D (2DFCS)----------------------------
class CDhp2DFCS
 {
 protected:
  char    name[50]; //numele sistemului de coordonate
  double  P0[2],    //originea 2DFCS în coordonate globale
          P1[2],    //punct pe axa X a 2DFCS în coordonate globale
          P2[2];    //punct pe axa Y a 2DFCS în coordonate globale

 public:
  CDhp2DFCS(); //constructorul implicit al 2DFCS
  CDhp2DFCS(
            double origin[2], //originea 2DFCS
            double ptOnX[2],  //puctul pe axa X
            char   name[50]   //numele 2DFCS
           );
  CDhp2DFCS(
            double origin[2],        //originea 2DFCS
            double angleToGlobalX,   //unghiul (Xglobal,X2DFCS)
            char   name[50]          //numele 2DGCS
           );
  void GetPoints(
                 double P0[2],
                 double P1[2],
                 double P2[2]
                )const;
  void GetPoints(
```

BUPT

```
                  CdhpEd2dPoint &P0,
                  CdhpEd2dPoint &P1,
                  CdhpEd2dPoint &P2
                );
   void   SetPoints(
                    double P0[2],
                    double P1[2],
                    double P2[2]
                  );
   void   SetPoints(
                    double P0[2],
                    double angle
                  );
   char   *GetName();
   void   SetName(
                  char *nameFCS
                );
   double GetAngle(void);
   void   CoordTransLocalToGlobal(//conversie coordonate local→global
                                  double pGlobal[2],
                                  double pLocal[2]
                                );
   void   CoordTransGlobalToLocal(//conversie coordonate global→local
                                  double pGlobal[2],
                                  double pLocal[2]
                                );
   virtual ~CDhp2DFCS(){}

  private:
   void makeVersor(//determinarea versorului vectorului p1→p2
                   double p1[2],
                   double p2[2],
                   double coord[2]
                  );
   void makeOrtho(//determinare vectorul v perpendicular pe u
                  double u[2],
                  double v[2]
                );
  };
```

A.1.5 Fereastra 2D

```
//Fereastra 2D--------------------------------------------------------
class CdhpWindow2D : public CdhpWindow
  {
  protected:
    CdhpGrid grid; //rastrul
    CdhpList <CDhp2DFCS> listOf2DFCS; //lista de sist. Coord.2D
    CDhp2DFCS *active2DFCS; //sist coord 2D activ
    CdhpList<CdhpEd2dPoint> listPoint; //lista de pucte din fereastră
    CdhpList<CdhpEd2dLine>  listLine;  //lista de linii din fereastră

      public:
    CdhpWindow2D(
```

BUPT

```
                CdhpWorkspace *ws
               );

    CdhpWindow2D(
                CdhpWorkspace *ws,
                DhpWinType type,
                long pxmin,
                long pymin,
                long pxmax,
                long pymax
               );
    virtual ~CdhpWindow2D(){};
    void CoordTransLocalToGlobal(//transf.coord. sist.local→global
                                 double pGlobal[2],
                                 double pLocal[2]
                                );
    void CoordTransGlobalToLocal(//transf.coord. sist. Global→local
                                 double pGlobal[2],
                                 double pLocal[2]
                                );
    void GetActive2DFCS(//determinarea sistemului activ
                        double P0[2],
                        double P1[2],
                        double P2[2]
                       );
    void GetActive2DFCS(//determinarea sistemului activ
                        CdhpEd2dPoint &P0,
                        CdhpEd2dPoint &P1,
                        CdhpEd2dPoint &P2
                       );
    virtual void LoadListOf2DFCS(//încărcarea listei de sist.coord.
                                 char *section
                                ){};
    virtual void SaveListOf2DFCS(//salvarea listei de sist.coord.
                                 char *section,
                                 CdhpList <CDhp2DFCS> *listOf2DFCS
                                ){};
    void Add2DFCS(
                 CDhp2DFCS *FCS
                );
//metode de acces la interfața rastrului
    void RestoreGridState(void){this->grid.RestoreState();};
    void SetGridState(DhpGridState st){this->grid.SetState(st);};
    void SwitchGridState(void);
    void SetCrtGridPoint(
                         double crtPoint[2]
                        );
    void GetCrtGridPoint(
                         double crtPoint[2]
                        );
    void GetGrid(
                double &ax,
                double &ay,
                DhpGridState &st
               ){this->grid.GetGrid(ax, ay, st);};
    void SetGrid(
```

BUPT

```
                double ax,
                double ay,
                DhpGridState st
               ){this->grid.SetGrid(ax, ay, st);};
    void AtractionGrid(double crtMouseCoord[2]);
    double GetGridPrecision();
    virtual void SetPoint( //desenarea unui punct
                          double P[2],
                          short color,
                          short mode
                         ) = 0;
    void ClearListPoint(//şterge lista de puncte ataşată ferestrei
                        DeleteMode delMode=DONT_DELETE_CONTENT
                        );
    void ClearListLine(//şterge lista de linii ataşată ferestrei
                       DeleteMode delMode=DONT_DELETE_CONTENT
                      );
    void ClearAllLists(//şterge toate listele de entităţi 2D din fer.
                       DeleteMode delMode=DONT_DELETE_CONTENT
                      );
    void AddPoint(//adaugă un punct la lista de puncte a ferestrei
                  CdhpEd2dPoint *point
                 );
    void AddLine(//adaugă o linie la lista de linii a ferestrei
                 CdhpEd2dLine *line
                );
    void DrawLine(//desenarea unei linii (metodă pur virrtuală)
                  double Sx,
                  double Sy,
                  double Ex,
                  double Ey,
                  short color,
                  unsigned short lineStyle,
                  short mode) = 0;
  };

#endif     //_DHPINCLUDE_DHPWIN
```

A.2 Interfeţele claselor editorului 2D

```
class CdhpEditor2D;

#define PI          3.141592653589793
```

A.2.1 Elementele modulului UserInput

```
//definiţii de tip pentru modulul UserInput---------------------------
#define LEFT_BUTTON   0x00
#define RIGHT_BUTTON  0x01
#define MIDDLE_BUTTON 0x02

typedef enum{
```

BUPT

```
            MOUSE_VORHANDEN,
            MOUSE_NICHT_VORHANDEN
           }DhpMouseState;

typedef enum{
            INPUT_CHAR,
            INPUT_INT,
            INPUT_FLOAT,
            INPUT_COORD
           }DhpInputType;

typedef void (*CallFunction)(
                              double coord[2],
                              void *ptr
                             );

//Clasa CdhpMouse------------------------------------------------------
class CdhpMouse
  {
  protected:
    DhpMouseState status;
  public:
    CdhpMouse(){status = MOUSE_NICHT_VORHANDEN;};
    DhpMouseState GetMouseState(void){return status;};
    virtual void  ShowMouse(void) = 0;
    virtual void  HideMouse(void) = 0;
    virtual void  SetMousePosition(
                                   long x,
                                   long y
                                  ) = 0;
    virtual void  GetMousePosition(
                                   long *x,
                                   long *y
                                  ) = 0;
    virtual void  SetMouseRange(
                                long xmin,
                                long xmax,
                                long ymin,
                                long ymax
                               ) = 0;
    virtual void  SetMouseSpeed(
                                short horizontal_speed,
                                short vertical_speed,
                                short double_speed_treshold
                               ) = 0;
    virtual short getbutton(
                            short nr
                           ) = 0;
    virtual void  getmotion(
                            long *px,
                            long *py
                           ) = 0;

    virtual struct xycoord moveto(
                                  long x,
                                  long y
```

BUPT

```
                              ) = 0;
    virtual short GetShowMouseState(void) = 0;
    virtual void  clear_mouse_buttons(void) = 0;
  };

//Clasa CdhpCursor------------------------------------------------------
class CdhpCursor
  {
  protected:
    CdhpEditor2D    *parent;    // editorul de care este ataşat cursorul
    DhpWindowData   wData;      // limitele ferestrei in care se
                                // deseneaza cursorul
    short           state;      // Starea cursorului (0 = invizibil,
                                // !0 = vizibil)
    short           skipDraw;   // Daca este 2 se evita doar desenarea
                                // cursorului.
                                // Daca este 1 se evita desenarea
                                // cursorului si a desenului suplimentar.
    Long            x, y;       // coordonatele cursorului
    CallFunction    drawFunc;   // funcţie suplimentară de desenare pe
                                // cursor
    void            *drawObj;   // obiectul suplimentar care se deseneaza
  public:
    CdhpCursor()
      {
      parent = NULL;
      state = 0;
      skipDraw = 0;
      x = y = 0;
      drawFunc = NULL;
      drawObj = NULL;
      }

    void SetParent(
                   CdhpEditor2D *editor
                  )
      {
      this->parent = editor;
      }

    void SetCursorDraw(
                       CallFunction func,
                       void *obj
                      )
      {
      this->drawFunc = func;
      this->drawObj = obj;
      }

    void SetSkipDraw(
                     short sd
                    )
      {
      this->skipDraw = sd;
      }
```

BUPT

```
    virtual void Draw(void)=0;
    virtual void Hide(void)=0;
    virtual void Move(
                     long cx,
                     long cy
                    )=0;
    virtual short GetFangRadius(void)=0;
    virtual void IncreaseFangRadius(void)=0;
    virtual void DecreaseFangRadius(void)=0;
  };

//Clasa CdhpCommandLine-----------------------------------------------
class CdhpCommandLine
  {
  protected:
    short rows, // numărul de rânduri ale liniei de comandă
          columns; // numărul de coloane ale liniei de comandă
    short IALin, // variabile pentru InputArea
          IACol,
          IALen,
          IABColor,
          IAFColor;
    DhpInputType IAType;  // tipul de date introduse (pereche de
                          // coordonate sau valoare individuală)
    char  *IAText;  //antetul liniei de comandă
    char  AcceptedChars[101];  //alfabetul liniei de comandă
    short CaretPosition;  //poziția curentă a cursorului
    short Stack[21]; //implementare a automatului finit
    short topOfStack;//
    short AcceptChar(short c);

  public:
    CdhpCommandLine(
                    short rs=0,
                    short cs=0
                   );
    ~CdhpCommandLine();

    void SetParameters(
                       short rs,
                       short cs
                      );

    short GetRowsNr(void)
      {
      return rows;
      }

    short GetColsNr(void)
      {
      return columns;
      }

    void  SetInputArea(
                       short lin,
                       short col,
```

BUPT

```
                        short len,
                        short fColor,
                        short bColor,
                        DhpInputType type
                       );
    void  ResetInputArea(void);
    void  KeyPress(
                    short KeyAscii
                   );

    char  *GetInputText()
      {
      return IAText;
      }

    DhpInputType GetInputType(void)
      {
      return this->IAType;
      }

    virtual void WriteText(
                            short lin,
                            short col,
                            short foreColor,
                            short backColor,
                            char *text
                           )=0;
    virtual void  Clear(
                         short lin1,
                         short col1,
                         short lin2,
                         short col2,
                         short color
                        )=0;
    short GetCrtStateOfStack(){return Stack[topOfStack];};
  };

//Clasa CdhpCurrentCoord2D-------------------------------------------
//clasă auxiliară pentru afişarea coordonatelor curente ale cursorului
//în statusbar.
Class CdhpCurrentCoord2D
  {
  public:
    virtual void WriteCoord(
                             double x,
                             double y,
                             char c1='X',
                             char c2='Y',
                             short show=3) = 0;
    virtual void Clear(void) = 0;
  };

//Clasa CdhpUserInput------------------------------------------------
class CdhpUserInput
  {
  protected:
```

BUPT

```
    CdhpCursor              *cursor;            //Cursorul
    CdhpCommandLine         *commandLine;       //Linia de comanda
    CdhpMouse               *mouse;             //Mouse-ul
    CdhpEditor2D            *parent;            //Editorul 2D
    CdhpCurrentCoord2D      *coord;             //afisarea coordonatelor
    bool                    existPrevPoint;   //distincţia între situaţia
                            //introducerii primului punct şi cea a
                            //introducerii punctelor următoare
    CdhpEd2dPoint           prevPoint;          //punctul anterior selectat
    long                    pCrtX, //pozitie curenta [pixeli] mouse
                            pCrtY;
    double                  currCoord[2];       //poziția curentă [m] mouse
    CallFunction            track; //Funcţie de desenare „Gummiband"
    void                    *trackObj; //obiectul de desenat pe post de
                                       //„Gummiband"
    CdhpList <CdhpEd2dEntity> listBlinkedEntities; //listă de entităţi
                                   //care satisfac un anumit criteriu de
                                   //prindere (pentru confirmare de către
                                   //utlilizator)
    short                   dirArrow;

  public:
    CdhpUserInput(
                  CdhpMouse *ms = NULL,
                  CdhpCursor *curs=NULL,
                  CdhpCommandLine *cl=NULL,
                  CdhpCurrentCoord2D *crd = NULL,
                  CdhpEditor2D *editor=NULL
                 );
    ~CdhpUserInput(void);

    short GetPoint(//returnează punct în fereastra activă
                   CdhpEd2dPoint &p
                  );
    short GetConfirmation(
                          char *mesaj1 = NULL,
                          char *mesasj2 = NULL
                         );
    void  SetCursor(
                    CdhpCursor *curs
                   )
      {
      this->cursor = curs;
      }

    void  SetCursorDraw(
                        CallFunction func,
                        void *obj
                       )
      {
      cursor->SetCursorDraw(func, obj);
      }

    void  SetCommandLine(
                         CdhpCommandLine *cl
                        )
```

```
  {
  this->commandLine = cl;
  }

void  WriteText(
                short x,
                short y,
                short foreColor,
                short backColor,
                char *text
               );
void  SetInputArea(
                   short lin,
                   short col,
                   short len,
                   short fColor,
                   short bColor,
                   DhpInputType type
                  );
void  SetMouse(
               CdhpMouse *m
              )
  {
  mouse = m;
  }

void  SetParent(
                CdhpEditor2D *edit
               );
void  SetExistPrevPoint(
                        bool newValue = true
                       )
  {
  this->existPrevPoint = newValue;
  }

bool  GetExistPrevPoint()
  {
  return this->existPrevPoint;
  }

void  SetPrevPoint(
                   double x,
                   double y
                  );
short GetPrevPoint(
                   CdhpEd2dPoint &P
                  );
void  SetCoord(
               CdhpCurrentCoord2D *c
              )
  {
  this->coord = c;
  }

short GetFangRadius()
```

BUPT

```
    {
    return cursor->GetFangRadius();
    }

  void  Reset();
  void  GetTrackFunction(
                          CallFunction &func,
                          void **obj
                         );
  void  SetTrackFunction(
                          CallFunction func,
                          void *obj
                         );
  void  ZoomPlus(void);
  short GetOrthoDistFromOrig(
                              short VertHoriz,
                              char c,
                              double *dist
                             );
  void  AddToListBlinkedEntities(
                                  CdhpEd2dEntity *Entity
                                 );
  void  ClearListBlinkedEntities(
                                  DeleteMode delMode =
                                             DONT_DELETE_CONTENT
                                 );
  virtual short  KeyboardHit()=0;
  virtual short  GetCh()=0;
  virtual void   RedrawProject()=0;
  virtual void   SetMouseCapture(void)=0;
  virtual void   ReleaseMouseCapture(void)=0;
  virtual void   SetErrorMessages(
                                   char *str
                                  )=0;
  virtual short  ShiftPressed() = 0;
  virtual long   Clock() = 0;
  virtual short  ShowDialogTempFunctions()=0;

private:
  void  GetMouseOrKey(
                       long *px,
                       long *py,
                       short *ev_type,
                       short *ev_data
                      );
  short IsCommandLineCorrect(
                              const char *str
                             );
  //transformarea şirului introdus în linia de comandă în coordonate
  //de punct
  short PointToCoord(
                      const char *str,
                      double coord[2]
                     );
  void  DistanceToCoord(
                         double distance,
```

```
                    double currCoord[2]
                   );
//procesarea evenimentelor de mouse şi de tastatură
void  FilterMouseOrKeyForInputCoord(long *px,
                                    long *py,
                                    short *ev_type,
                                    short *ev_data
                                   );
void  FilterMouseOrKeyForDouble(
                                long *px,
                                long *py,
                                short *ev_type,
                                short *ev_data
                               );
void  DirectionalArrow(
                        short direction
                       );
void  GetCoord(
                short *ev_type,
                short *ev_data,
                CdhpEd2dPoint &p
               );
};
```

A.2.2 Elementele modulului Editor2D

```
#include <Entity2D.h>
#include <UsrInput.h>
//clasa editor 2D----------------------------------------------------
class CdhpEditor2D
  {
  public:
    CdhpWorkspace *workspace; //spaţiul de lucru ataşat
    CdhpUserInput *ui;        //interfaţa utilizator

  private:
    bool tempFunc; //fanion activare funcţii temporare
    long catchType; //tipul de prindere setat
    //lista de puncte situate în interiorul reticulului cursorului
    CdhpList <CdhpEd2dPoint>  listPointsFromReticle;
    //lista de linii situate în interiorul reticulului cursorului
    CdhpList <CdhpEd2dLine>   listLinesFromReticle;
    //lista de puncte situate în interiorul reticulului cursorului care
    //au fost deja refuzate de utilizator
    CdhpList <CdhpEd2dPoint>  listPointsRefused;
    //lista de linii situate în interiorul reticulului cursorului care
    //au fost deja refuzate de utilizator
    CdhpList <CdhpEd2dLine>   listLinesRefused;

  public:
    CdhpEditor2D(
                 CdhpWorkspace *work,
                 CdhpUserInput *usrinput
                );
```

BUPT

```
~CdhpEditor2D();
CdhpWindow2D *GetWindow(void);
short  GetPoint(
                CdhpEd2dPoint &p
               );

void   SetPrevPoint(
                    double x,
                    double y
                   )
  {
  this->ui->SetPrevPoint(x, y);
  }

short  GetPrevPoint(
                    CdhpEd2dPoint &P
                   );
void   ResetPrevPoint(void)
  {
  this->ui->SetExistPrevPoint(false);
  }

void   Reset(void)
  {
  this->ui->Reset();
  }

short  GetLine(
               CdhpEd2dPoint &P1,
               CdhpEd2dPoint &P2
              );
void   SetInputArea(
                    short lin,
                    short col,
                    short len,
                    short fColor,
                    short bColor,
                    DhpInputType type
                   );
void   SetTrackFunction(
                        CallFunction func,
                        void *obj
                       );
void   SetCursorDraw(
                     CallFunction func,
                     void *obj
                    );
short  GetConfirmation(
                       char *mesaj1=NULL,
                       char *mesasj2 = NULL
                      );
short  GetOrthoDistFromOrig(
                            short VertHoriz,
                            char c,
                            double *dist
                           );
```

BUPT

```
    void    WriteText(
                     short lin,
                     short col,
                     short Bcolor,
                     short Fcolor,
                     char *text
                    );
    void    SetCatch(
                    long catchType
                   );
    void    GetCatch(
                    long &catchType
                   );
    short   GetTempPoint(
                          short button,
                          CdhpEd2dPoint &p
                         );
    short   MiddleBetweenTwoPoints(
                                    CdhpEd2dPoint &p
                                   );

  private:
    bool   PointInReticule(
                            CdhpEd2dPoint *P,
                            CdhpEd2dPoint click,
                            double metricReticleSide
                            );
    bool   PointInList(
                        CdhpEd2dPoint *P,
                        CdhpList <CdhpEd2dPoint> &list
                        );
    bool   ValidatedPoints(
                            CdhpList <CdhpEd2dPoint> &list,
                            CdhpEd2dPoint &P
                           );
    bool   LineRefused(
                        CdhpEd2dLine  *L
                        );
    bool   StandardEndPoint(
                             CdhpEd2dPoint &click
                             );
    bool   EndPoint(
                   CdhpEd2dPoint &click
                   );
    bool   IntersPoint(
                        CdhpEd2dPoint &click
                        );
    bool   CatchLine(
                     CdhpEd2dPoint click,
                     CdhpEd2dLine &line
                     );
    bool   CatchPerpPoint(
                           CdhpEd2dPoint &click
                           );
    bool   CatchHorizPoint(
                            CdhpEd2dPoint &click
```

BUPT

```
                                );
    bool   CatchVertPoint(
                               CdhpEd2dPoint &click
                               );
    void   CorectCoords(
                             CdhpEd2dPoint &click
                            );
  };
```

A.3 Mecanismul de aserţiuni bazat pe excepţii

A.3.1 Codul sursă al mecanismului de aserţiuni bazat pe excepţii

```
#ifndef EXCEPTASSERT
#define EXCEPTASSERT

class CdhpAssertion;

#define ASSERTION(type,exp) \
throw \
CdhpAssertion(CdhpAssertion::type,#type"("exp")",__FILE__,__LINE__)

#if defined ALL_ASSERTIONS || defined ASSERT_REQUIRE
#define REQUIRE(exp)  \
if(!(exp)){ASSERTION(Require,#exp);} else {;}
#else
#define REQUIRE(exp)
#endif

#if defined ALL_ASSERTIONS || defined ASSERT_ENSURE
#define ENSURE(exp)  \
if(!(exp)){ASSERTION(Ensure,#exp);} else {;}
#else
#define ENSURE(exp)
#endif

#if defined ALL_ASSERTIONS || defined ASSERT_CHECK
#define CHECK(exp)  \
if(!(exp)){ASSERTION(Check,#exp);} else {;}
#else
#define CHECK(exp) if(!(exp)){;} else {;}
#endif

#if defined ALL_ASSERTIONS
#define INVARIANT protected: virtual void invariant() const {
#define END_INVARIANT }
#define CHECK_INVARIANT invariant()
#else
```

BUPT

```
#define INVARIANT
#define END_INVARIANT
#define CHECK_INVARIANT
#endif

#if defined ALL_ASSERTIONS
#define ASSERT(exp)\
if(!(exp)){ASSERTION(Assert,#exp);} else {;}
#else
#define ASSERT(exp)
#endif

class CdhpAssertion
  {
  public:
    enum Type{Require,
              Ensure,
              Check,
              Assert
             };

    CdhpAssertion(
                  Type assertionType,
                  const char *reason,
                  const char *file,
                  int line
                 ):
                  theType(assertionType),
                  theReason((char *)reason),
                  theFile((char*)file),
                  theLine(line){}

    CdhpAssertion(
                  const CdhpAssertion &other
                 ):
                 theType(other.theType),
                 theReason(other.theReason),
                 theFile(other.theFile),
                 theLine(other.theLine)
                 {};

    const char* getReason() const;
    const char* getFile() const;
    int getLine() const;
    Type getType() const;
    virtual ~CdhpAssertion(){};

  private:
    Type  theType;
    char  *theReason;
    char  *theFile;
    int   theLine;
  };
#endif  //EXCEPTASSERT
```

BUPT

A.3.2 Exemplu de implementare la nivelul clasei CDhp2DFCS

```
class CDhp2DFCS{
  protected:
    char     name[50]; //numele sistemului de coordonate
    double   P0[2],    //originea (în coordonate globale)
             P1[2],    //versorul P0->P1 reprezintă axa X a sistemului
             P2[2];    //versorul P0->P2 reprezintă axa Y a sistemului
  public:
    CDhp2DFCS();       //constructor implicit

    CDhp2DFCS(
              double origin[2],
              double ptOnX[2],
              char name[50]
             )
      {
      double axaX[2],
             axaY[2];
      REQUIRE(fabs(origin[0]-ptOnX[0])>PRECISION ||
              fabs(origin[1]-ptOnX[1])>PRECISION);
      REQUIRE(strlen(name) < 50);
      this->makeVersor(origin, ptOnX, axaX);
      this->makeOrtho(axaX,axaY);
      strcpy(this->name, name);
      this->P0[0] = origin[0];
      this->P0[1] = origin[1];
      this->P1[0] = origin[0] + axaX[0];
      this->P1[1] = origin[1] + axaX[1];
      this->P2[0] = origin[0] + axaY[0];
      this->P2[1] = origin[1] + axaY[1];
      CHECK_INVARIANT;
      }

    void SetPoints(
                   double P0[2],
                   double P1[2],
                   double P2[2]
                  )
      {
      this->P0[0] = P0[0];
      this->P0[1] = P0[1];
      this->P1[0] = P1[0];
      this->P1[1] = P1[1];
      this->P2[0] = P2[0];
      this->P2[1] = P2[1];
      CHECK_INVARIANT;
      }

    void SetName(
                 char *nameFCS
                )
      {
      REQUIRE(strlen(nameFCS < 50);
      strcpy(this->name,nameFCS);
      }
```

BUPT

```
    virtual ~CDhp2DFCS(){}

  private:
    void makeVersor(
                    double p1[2],
                    double p2[2],
                    double coord[2]
                   );

    void makeOrtho(
                   double u[2],
                   double v[2]
                  );

  INVARIANT
    ASSERT((VECT_LENGTH(makeVector(P0,P1)-1.0) < PRECISION)
    ASSERT((VECT_LENGTH(makeVector(P0,P2)-1.0) < PRECISION)
    ASSERT(SCALPRD(makeVector(P0,P1),makeVector(P0,P2) < PRECISION)
  END_INVARIANT
  };
```

A.4 Implementarea aplicației Dhp-Bauwerk

A.4.1 Sistemul de ferestre

Implementarea sistemului de ferestre în cadrul mediului de proiectare *DHP-Bauwerk* este prezentată în figura de mai jos. Fereastra principală conține o reprezentare 2D a planului casei. Rastrul și sistemele de coordonate libere pot fi setate în cadrul cutiei de dialog afişate. Situația prezentată corespunde sistemului de coordonate „FKS 001" activ; rastrul având pasul de 0.5 m atât pe direcția X cât şi pe direcția Y este definit în sistemul de coordonate activ.

Fereastra din dreapta jos a are rolul de vizualizare a ultimului perete prelucrat în cadrul etajului actual.

BUPT

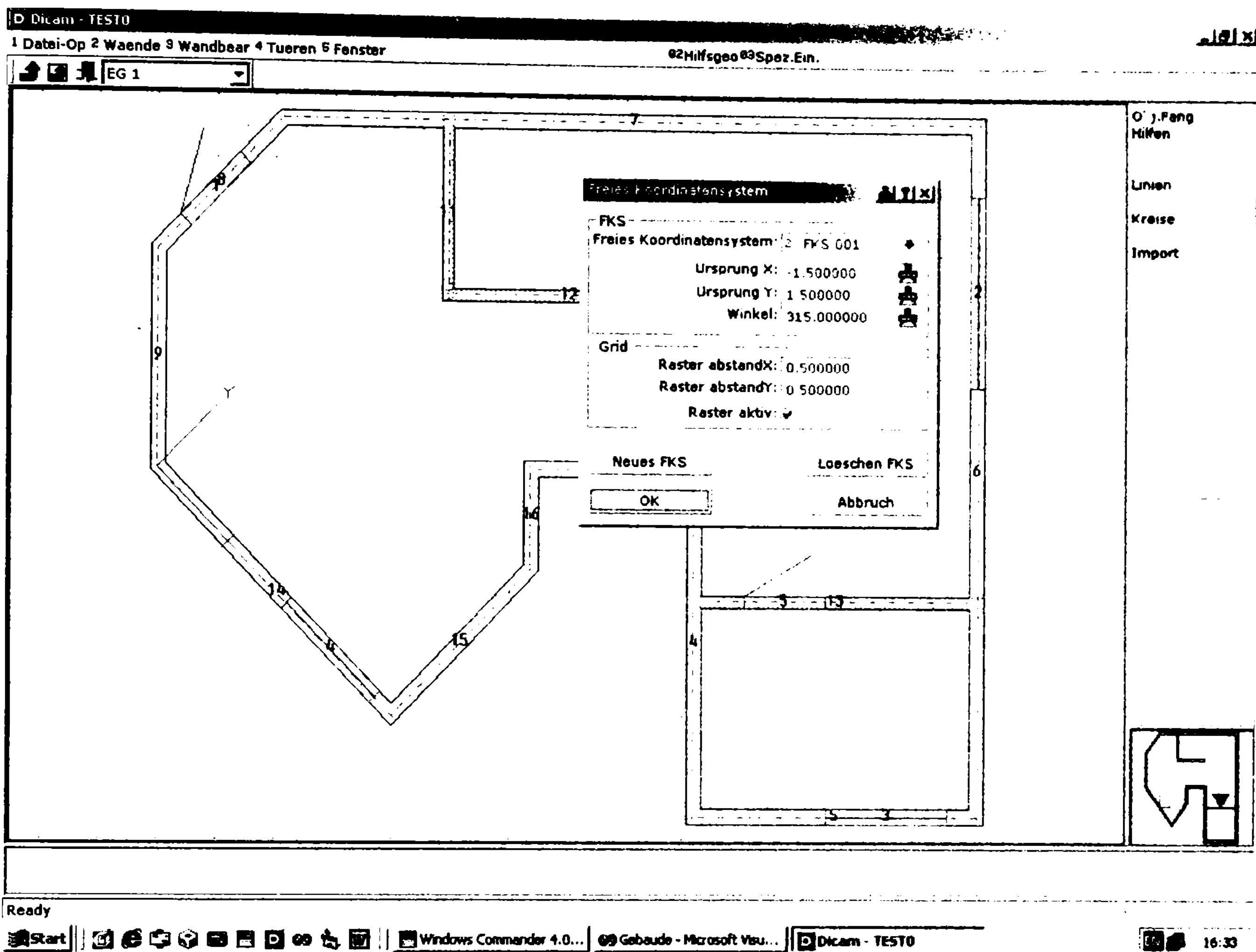


Figura A.1: Sistemul de ferestre integrat în cadrul mediului DHP-Bauwerk. Ferestre 2D

Fereastra 3D aparţine modulului *DICAM: Freie Konstruktion* şi ilustrează imaginea spaţială în reprezentare „cadru de sârmă" a etajului din figura de mai sus.

Universitatea tehnică
TIMIŞOARA
Biblioteca centrală

BUPT

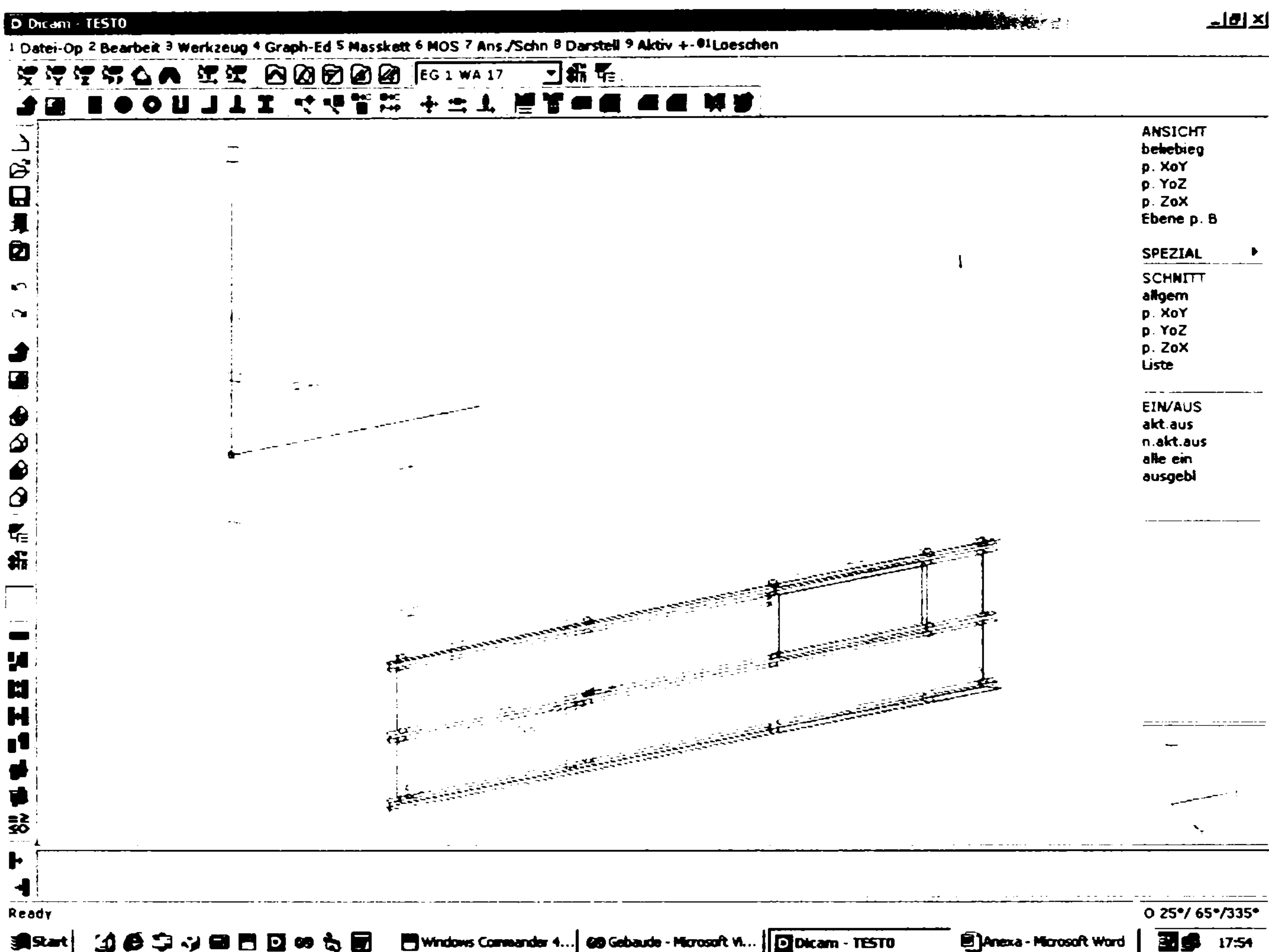


Figura A.2: Sistemul de ferestre integrat în cadrul mediului DHP-Bauwerk. Fereastra 3D

A.4.2 Editorul 2D

Imaginile următoare prezintă editorul 2D integrat în cadrul modulelor mediului *DHP-Bauwerk.* Toate domeniile care beneficiază de funcționalitățile editorului 2D au la dispoziție toate serviciile puse la dispoziție de către acesta, incluzând diversele tipuri de prinderi şi funcții de generare de puncte auxiliare, oferind utilizatorului o soluție unitară şi uşor de utilizat pentru generarea de puncte în cadrul fiecărui domeniu.

A.4.2.1 Integrarea editorului 2D în modulul „Grundriss"

Procesul de construire a pereților ce formează etajul curent este ilustrat în figura de mai jos. Cazul de uz pe baza căruia se desfăşoară această operație este „introducerea unui punct în fereastra activă". Punctul introdus reprezintă capătul al doilea al peretelui curent.

BUPT

Coordonatele sunt introduse prin intermediul liniei de comandă, utilizând coordonate polare relative.

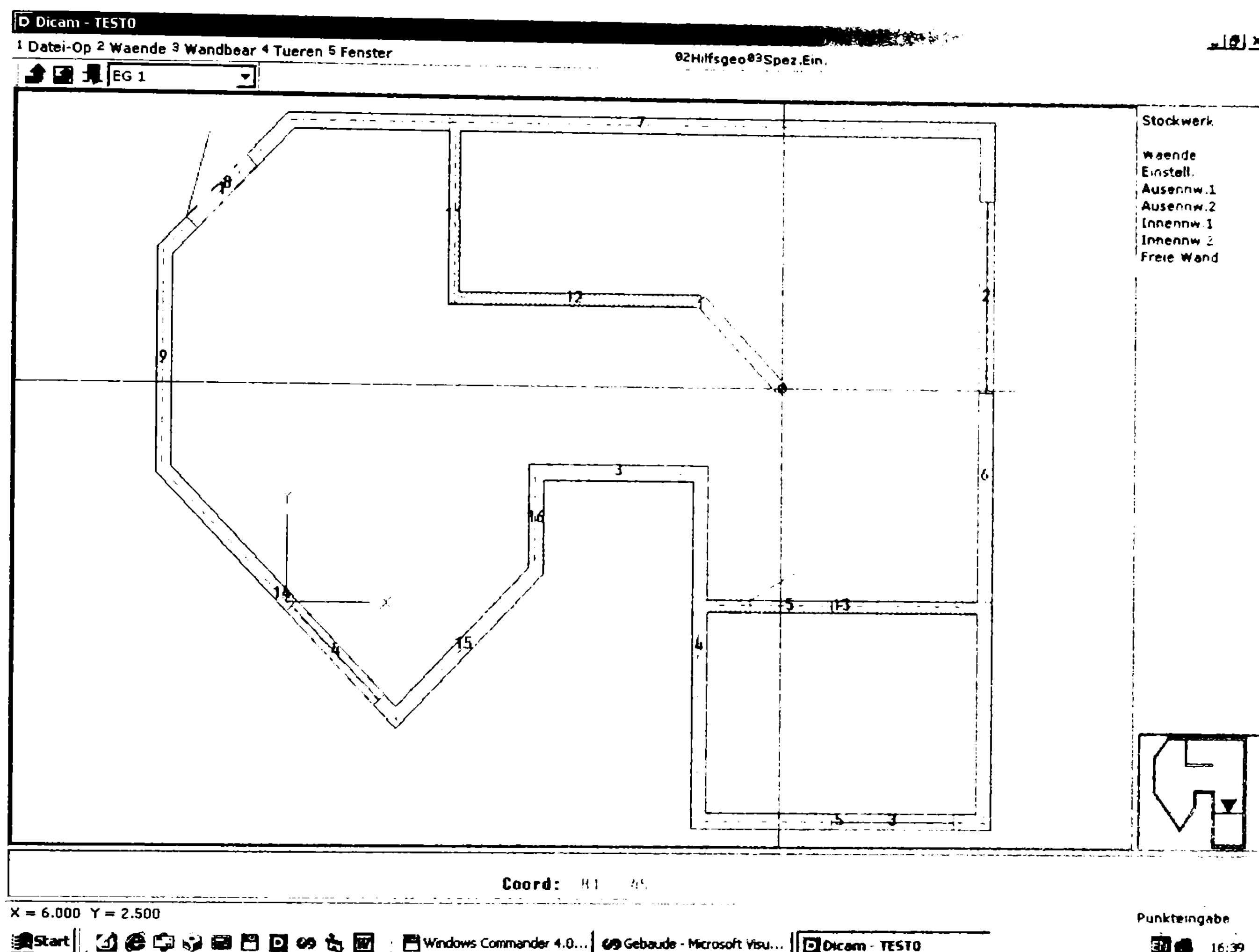


Figura A.3: Generarea unui nou perete în cadrul modulului „Grundriss"

A.4.2.2 Integrarea editorului 2D în modulul „Wandkonstruktion"

Modulul Wandkonstruktion utilizează editorul 2D într-o configurație cu trei ferestre. Fereastra principală conține vederea din față a peretelui, fereastra de jos conține vederea de sus a peretelui iar fereastra din dreapta conține vederea din stânga a acestuia. Figura ilustrează momentul introducerii unei noi bare în perete. Bara este atașată cursorului (efect obținut cu ajutorul variabilelor membre `CdhpCursor.drawFunc` și `CdhpCursor.drawObj`), serviciul apelat în acest moment în cadrul editorului 2D este „generare punct în fereastra activă", bazat pe cazul de uz omonim.

BUPT

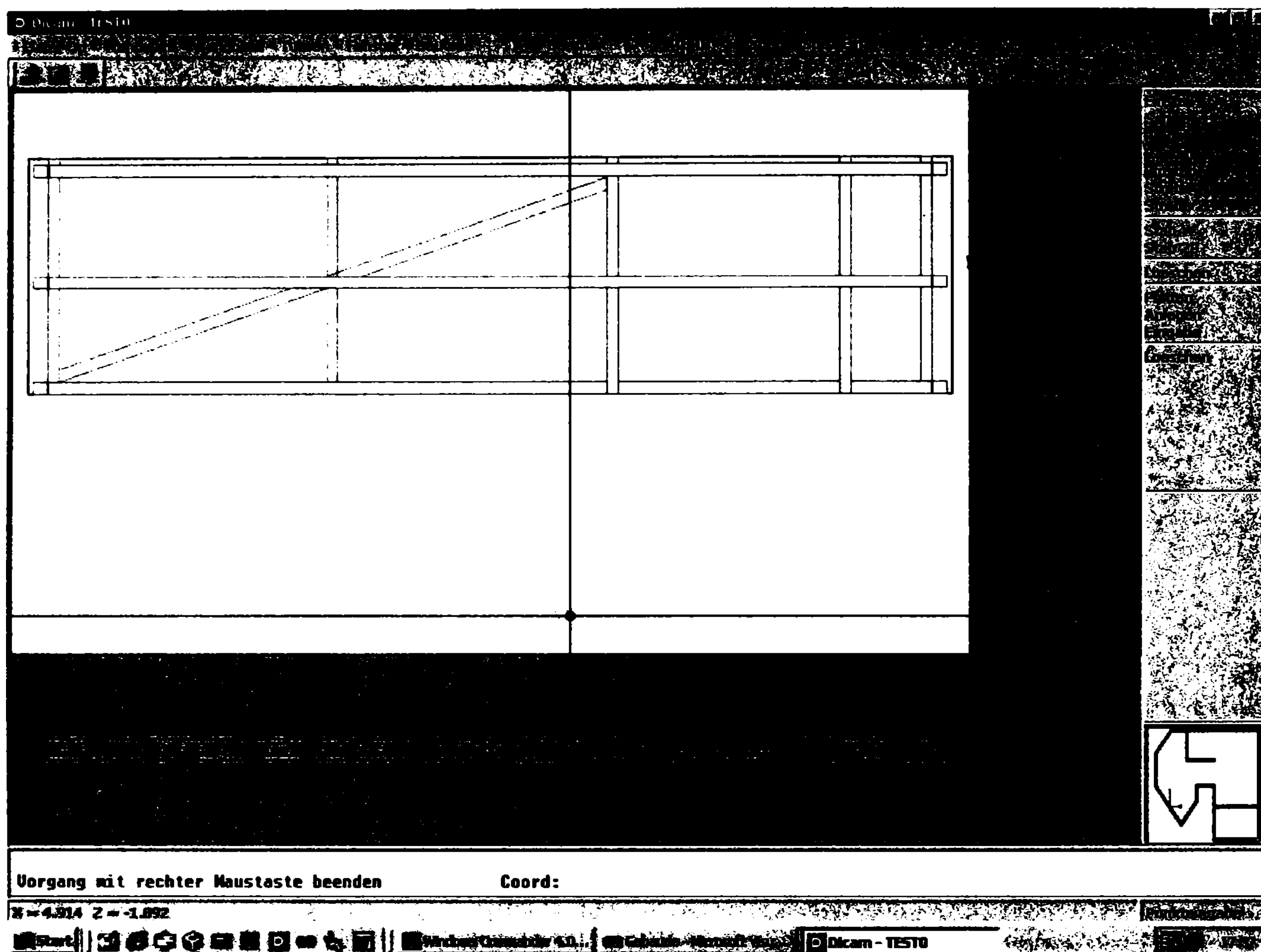


Figura A.4: Introducerea unei noi bare în cadrul peretelui curent.

BUPT

A.4.2.3 Integrarea editorului 2D în modulul „Dachausmittlung"

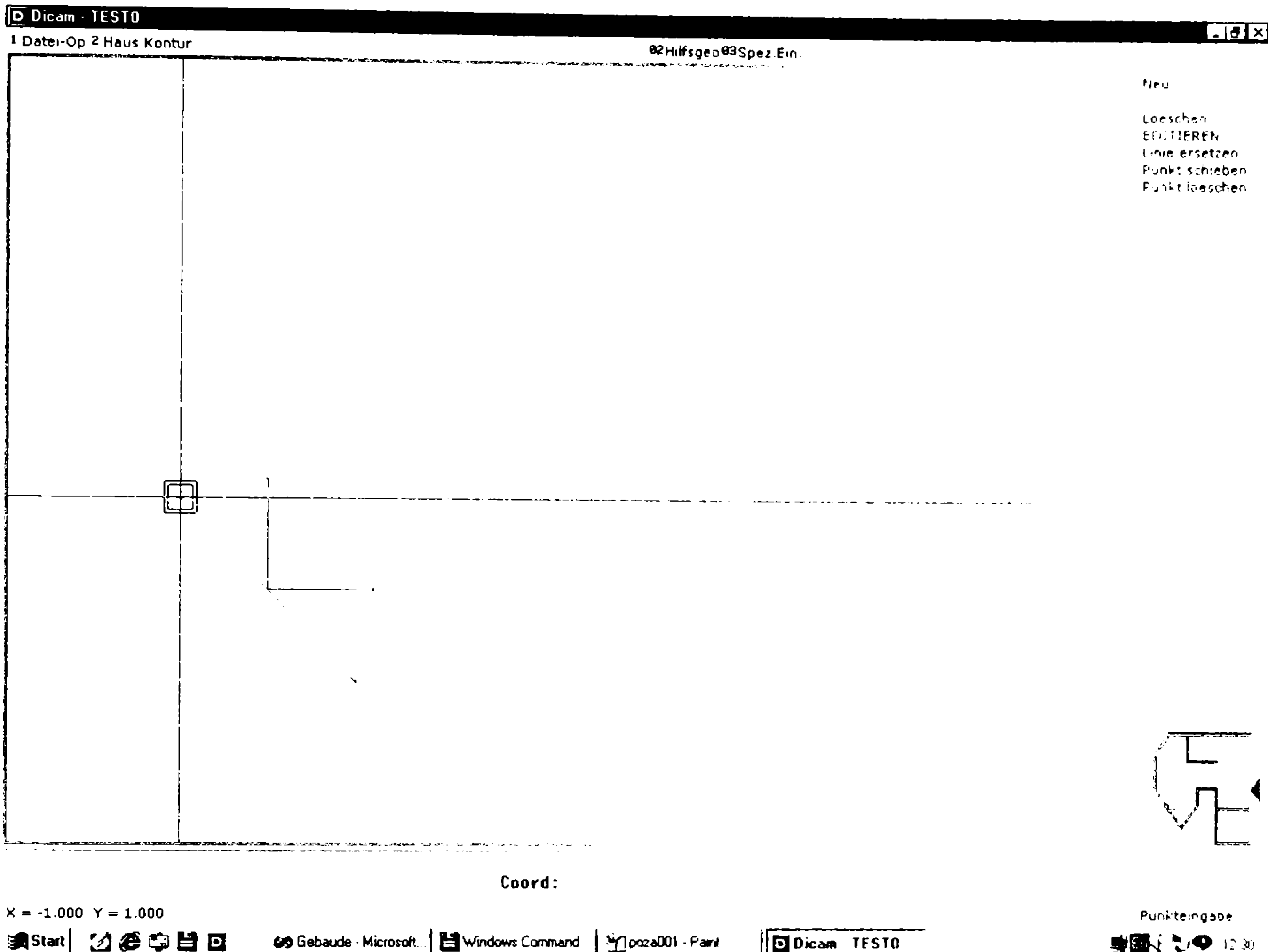


Figura A.5: Generarea conturului exterior al acoperişului utilizând editorul 2D

Editorul 2D este utilizat în cadrul modulului „Dachausmittlung" pentru generarea conturului exterior al acoperişului. În figura de mai sus este prezentat procesul de generare a acestui contur utilizând funcţia „generare punct în fereastra activă". Latura reprezentată în culoare verde reprezintă segmentul de dreaptă curent. Reprezentarea dublată a reticulului cursorului semnifică faptul că prinderile setate sunt active.

A.5 Referiri în literatura de specialitate cu privire la aplicaţia DHP-Bauwerk

Aplicaţia *DHP-Bauwerk* a firmei Dietrich's AG reprezintă o prezenţă de succes pe piaţa produselor de proiectare şi fabricaţie asistată de calculator. Referirile din literatura de specialitate cu privire la aplicaţia *DHP-Bauwerk* şi la principala componentă a sa, *DICAM*, o prezintă în ipostaza de sistem ce pune la dispoziţia utilizatorului o soluţie completă în domeniul construcţiilor din lemn.

BUPT

Figura următoare prezintă un extras din pliantul de prezentare a produsului *DHP-Bauwerk*.

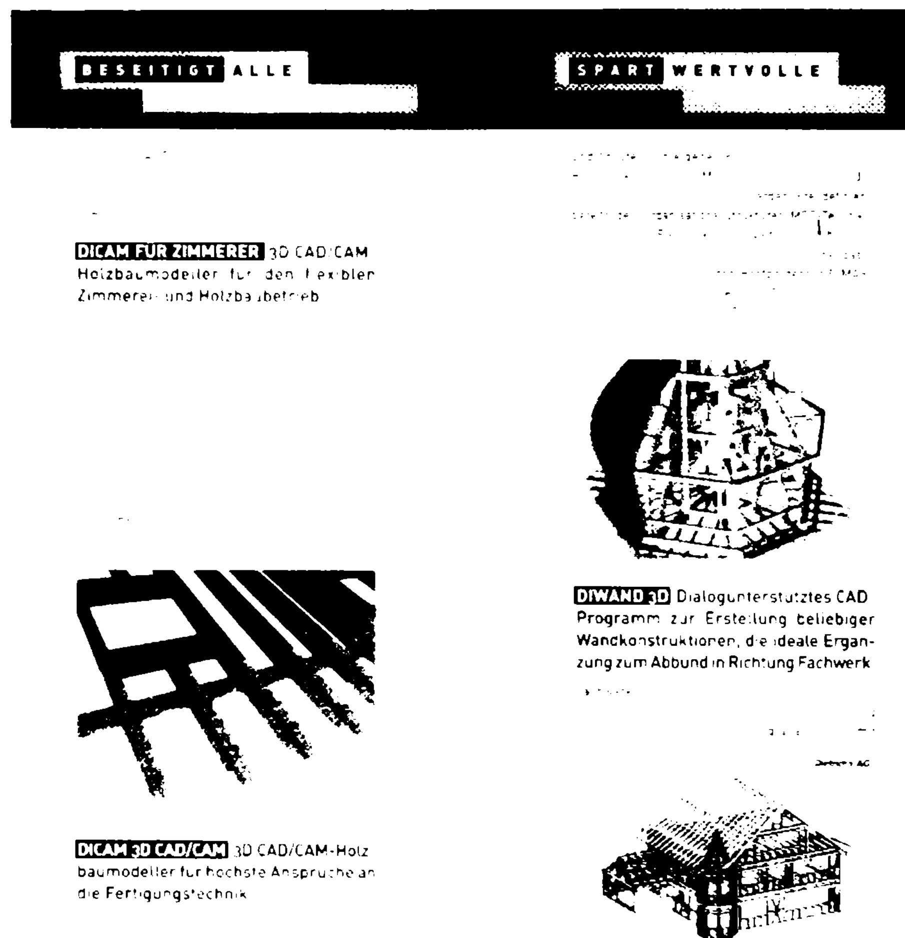


Figura A.6 Pliant de prezentare a aplicației *DHP-Bauwerk*

Revista Holz Kurier nr. 35 din 31.08.2000 elogiază interfața grafică a aplicației *DHP-Bauwerk*, evidențiind simplitatea utilizării și eficiența activității de generare și gestiune a proiectului în cadrul aplicației. Figura următoare prezintă un extras al articolului din revista menționată.

BUPT

Figura A.7 Articol în revista Holz Kurier nr.35 31.08.2000

Aplicația DHP-Bauwerk reprezintă o soluție credibilă chiar și pentru proiectarea structurilor care se abat de la formele clasice întâlnite în domeniul construcțiilor. Un exemplu în acest sens este oferit de pavilionul sub formă de frunză de la Expo 2000, prezentat în figura următoare.

BUPT

Figura A.8 Pavilion la Expo 2000 proiectat cu *DHP-BAUWERK*

Revista Quadriga, prin numărul 3 din 1999, remarca prezența la târgul internațional Ligna de la Hanovra a produsului *DICAM*, subliniind poziția de lider a firmei Dietrich's în domeniul produselor software destinate proiectării și execuției asistate de calculator pentru construcții în lemn. Figura următoare prezintă un extras al acestui articol.

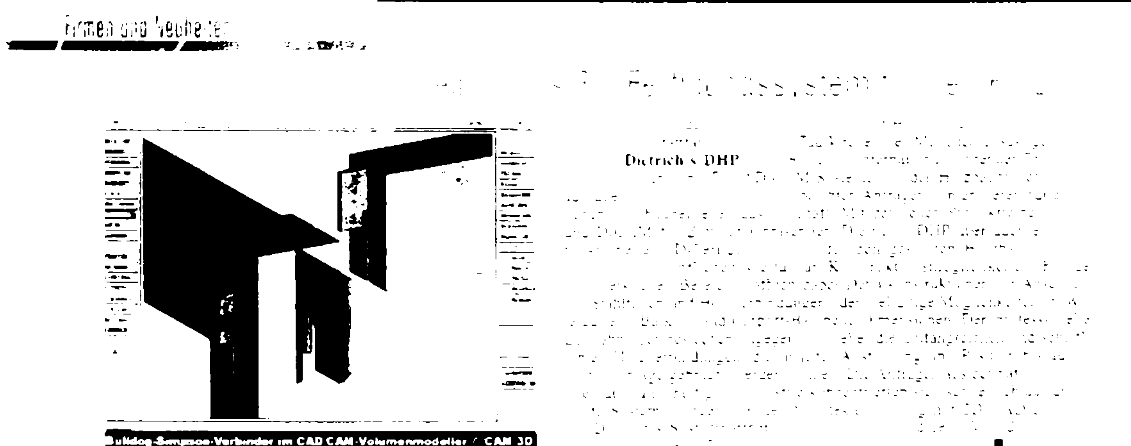


Figura A.9 Articol în revista Quadirga 3/99

Pentru activitatea didactică în domeniul construcțiilor în lemn, aplicația *DHP-Bauwerk* și în special modulul *DICAM – Freie Konstruktion*, reprezintă un instrument foarte apreciat. Figura următoare prezintă un extras dintr-un articol didactic publicat în revista „Der Zimmermann" pentru care ilustrațiile au fost realizate cu ajutorul modulului *DICAM – Freie Konstruktion*.

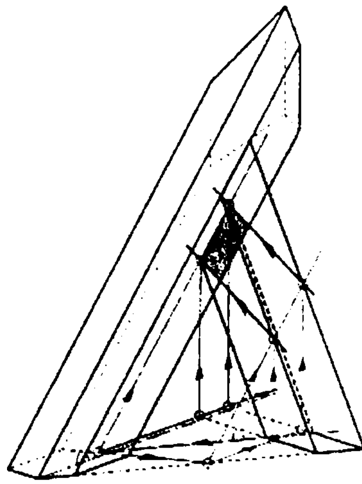


Bild 19: *Die Abschnittsfläche der Klaue (dunkel schraffiert) liegt in der konstruierten und hell schraffierten Schnittfläche (3D-Darstellung mit Hilfe von DICAM)*

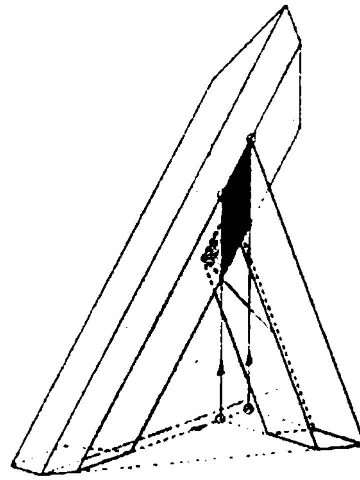


Bild 21: *Die Senkelschmiege des Schifters liegt in der Ebene der Seitenfläche des Gratsparrens (3D-Darstellung mit Hilfe von DICAM)*

Der Zimmermann 8/2000 **9**

Figura A.10 Articol în revista „Der Zimmermann" 8/2000

O prezentare a modulului *DICAM Freie Konstruktion* și a modului de abordare a unui proiect apare în revista Lignarius 11/1999. Sunt subliniate calitățile sistemului, care „permite construcția direct în 3D într-un mod deosebit de simplu și eficace, dată fiind necesitatea unei excelente vederi în spațiu pentru realizarea construcțiilor din lemn complexe".

BUPT

DICAM – 3d CAD/CAM

Dietrich's AG

Les constructions en bois complexes demandent de la part des concepteurs une excellente vision spatiale. Cette constatation est le point de départ de la programmation de ce logiciel de DAO. Il permet de construire directement en 3D de façon extrêmement simple et efficace. L'auteur nous montre deux exemples de constructions réalisés avec ce programme et nous explique les avantages de son utilisation.

Die Erstellung und Bearbeitung von [illegible]plexen [illegible]nstrukt[illegible]nen im H[illegible]lz-bau stellt höchste Ansprüche an das technische Fachwissen und das räumliche Vorstellungsvermögen des Konstrukteurs.

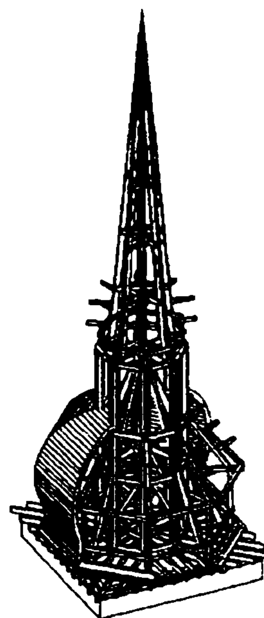
Aus dieser Erfahrung heraus hat die Dietrich's AG das 3D CAD/CAM System DICAM entwickelt, mit dem es ermöglicht wird eine Konstruktion direkt räumlich zu erstellen und dabei gleichzeitig alle Handhabungsvorteile von Bauteilbibliotheken, Baubeschlagsbibliotheken und einer durchgängigen CAM Komponente zu nutzen. Da DICAM grundsätzlich im 3D Modus arbeitet, setzt der Konstrukteur die Maus wie einen "Multifunktionsarm" ein, der entsprechend seiner "Bestückung" die verschiedensten Bearbeitungen durchführt. Am Beispiel einiger Projekte werden die bisher ungeahnten Konstruktionsmöglichkeiten, die Geschwindigkeit und der Bearbeitungskomfort von DICAM deutlich.

Projekt 1

Dieser Kirchturm ist in der Nähe von Dortmund (Deutschland) abgebunden und aufgestellt worden. Die Besonderheit des Projekts liegt darin, daß die gesamte Konstruktion auf einer Abbundanlage (P10) abgebunden worden ist und viele Bearbeitungen aufgrund von Einschränkungen der Abbundanlage sehr aufwendig hätten gefertigt werden müssen.

Durch den Einsatz von "Punktsymbolen" und dem Postprozessor DI-[illegible] jedo[illegible] fast alle [illegible]arbeitungen dennoch auf der Anlage durchgeführt werden und die Konstruktion vollständig im CAD erstellt werden. Die Punktsymbole in DICAM ermöglichen es an Bauteile wie z.B. Baubeschläge direkt die zur Montage notwendigen Bohrungen, Schlitze etc. an zu hängen. Wird ein solches Bauteil in die Konstruktion eingefügt, so werden automatisch alle Bearbeitungen auf die angrenzenden Hölzer übertragen. Löscht oder verschiebt man den Beschlag, so werden auch nur diese spezifischen Bearbeitungen verschoben oder gelöscht. Da Punktsymbole nicht zwangsläufig mit Bauteilen verbunden werden müssen, sonder auch als "freie Bearbeitungen" wie z.B. besondere Profile oder Verbindungen etc. erstellt und zugewiesen werden können, verfügt der Konstrukteur über eine annähernd grenzenlose Vielfalt an Möglichkeiten. Um doppelte Eingabe zu vermeiden, werden anschließend die entsprechenden Teile der Konstruktion einfach kopiert und /oder gespiegelt. Da alle Maschinenbearbeitungen erhalten bleiben und automatisch, entsprechend der Spiegelung oder dem Kopierpunkt umgerechnet werden, können selbst aufwendige Konstruktionen extrem schnell vollständig abgearbeitet werden.

Bearbeitungen, die standardmäßig mit einer Abbundanlage nicht möglich sind, können schon bei der Erstellung von Punktsymbolen aus verschiedenen Teilbearbeitungen der Aggregate der Anlage zusam-



8

Figura A.11 Articol în revista Lignarius 11/1999

Unul dintre noile module din cadrul aplicației DHP-Bauwerk care va fi disponibil începând cu toamna anului 2001 este Wandkonstruktion. Figura următoare ilustrează prezentarea acestui modul în cadrul numărului din ianuarie 2001 al buletinului Dietrich's AG.

BUPT

CODENAME POO 3D NEUES WANDSYSTEM

Mit viel Fleiß, Eifer und Durchhaltevermogen ist das neues Wandsystem aus dem Hause Dietrich's entstanden Die Neuentwicklung des Wandsystems auf der Basis des Dietrich's 3D-Volumenmodell setzt wieder einmal vollig neue Maßstabe in der Arbeitsvorbereitung

Einige Punkte in Kurze.

Sie arbeiten grundsatzlich im 3D-Modell des Gebaudes Dies erleichtert die Orientierung und Ausarbeitung von Anschlussdetails enorm

Fur die schnelle Bearbeitung werden einzelne Wande auf Knopfdruck in Draufsicht/Schnitt geklappt Die in Dietrich's neuem System nun standardisierte Grundrisseingabe teilt schon bei der Eingabe die Wand in die verschiedenen Lagen auf Die Aufteilung wird entsprechend den in der Bibliothek ausgewahlten Wandtypen vorgenommen Diese intelligente Wand ubernimmt dann die Positionierung von Fenstern, Turen, Standern Isolierung oder Beplankung etc in der Tiefe der Wand selbstandig. Komplizierte Fenster Tur , oder Anschlussdetails werden automatisch pass-genau erstellt

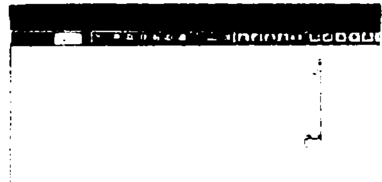
Das neue Wandsystem bietet dem Benutzer die Moglichkeit, in den gewohnten 2D-Ansichten

schnell zu arbeiten und den Komfort eines durchgangigen 3D-Gebaudemodell zu genießen

Fenster und Turenbibliothek mit beliebigen Formen der Offnungslocher Anschlags- und Offnungsseiten sind integriert

Besonders zeitsparend ist die schon bekannte Verschneidung der Wande mit anderen Bauteilen oder Gebaudeteilen Jede Lage des mehrschichtigen Wandaufbaus wird gesondert verschnitten Somit konnen Uberlappungen, Rucksprungmaße, Luftschlitze fur die Montage etc in jeder Wandlage einzeln angepasst werden Die Ausbildung von Taschen fur Pfetten, Anschlussdetails an Traufe und Ortgang Wande zwischen Sparren und Deckenbalken, schiefwinkelige Wande im Dachgeschoss usw. Alle diese Problemfalle und noch viele mehr konnen nun automatisch erledigt werden Sollten Ihnen die Moglichkeiten in den Bibliotheken nicht ausreichen, konnen Sie jeden Anschluss jedes Fenster oder jede Tur in 3D nacharbeiten

Die Verknupfung aller Bauteile mit dem Bauteilkatalog lasst Sie jede nur erdenkliche Liste erstel-



len Preise, Maße und Massen Gewichte Verschnitte, Anzahl und Art Stahlteile usw Rationelles Arbeiten erhalt mit dem Bauteilkatalog eine neue Bedeutung

Detaillierte Wandzeichnungen mit Ansicht, Draufsicht, Schnitt und Kennzeichnung der Wandposition im Stockwerk auf Knopfdruck

Alle gangigen Abbundanlagen bis hin zur PBA werden mit Steuerdaten versorgt

Mit OpenGL konnen Sie schon wahrend der Konstruktion einen Spaziergang durch das Gebaude machen oder Ihre Kunden mit photorealistischen Bildern begeistern Fur versierte "Internetfreaks konnen Sie sogar kleine Filme erstellen und diese per E-Mail versenden Unsere VRML Schnittstelle macht's moglich

Am besten Sie uberzeugen sich selbst von der Leistungsfahigkeit des neuen Systems Erste [illegible]

Figura A.12 Prezentarea modulului Wandkonstruktion în buletinul Dietrich's AG 1/2001

BUPT

Abrevieri şi notaţii

Abrevieri

ATCS	Sistem de control al traficului aerian (*Air Traffic Control System*)
ATM	Automat bancar (*Automated Teller Machine*)
CAD	Proiectare asistată de calculator (*Computer Aided Design*)
CAM	Fabricaţie asistată de calculator (*Computer Aided Manufacturing*)
CASE	Inginerie software asistată de calculator (*Computer-aided Software Engineering*)
CBO	Cuplare între obiecte (*Coupling between Objects*)
CNP	Cod numeric personal
CRC	Responsabilităţi şi colaborări ale clasei (*Class Responsability and Collaboration*)
DBMS	Sistem de gestiune a bazelor de date (*Database management system*)
DIT	Adâncimea arborelui de moştenire (*Depth of Inheritance Tree*)
ERD	Diagramă de relaţii între entităţi (*Entity Relationship Diagram*)
FP	Plan de zbor (*Flight Plan*)
IDL	2 Interactive Data Language
LCOM	Lipsa de coeziune a metodelor (*Lack of Cohesion of Methods*)
MFC	3 Microsoft Foundation Classes
NOC	Numărul de copii (*Number of Children*)
OA&D	Analiză şi proiectare orientată spre obiecte (*Object Analisys and Design*)
OCM	Modelul de comunicare al obiectelor (*Object Communication Model*)
OMA	*4 Arhitectura gestiunii obiectelor (*Object Management Architecture*)*
OMG	5 Object Managment Group
OMT	Tehnica modelării obiectelor (*Object Modelling Technique*)
OOA	Analiza orientată spre obiecte (*Object-oriented Analysis*)
OOD	Proiectarea orientată spre obiecte (*Object-oriented Design*)
OODLE	6 Object-Oriented Design LanguagE
OOP	Programare orientată spre obiecte (*Object-oriented Programming*)
OOSE	7 Object-oriented Software Engineering
OTW	8 Object Technology Workbench

BUPT

OWL	9 ObjectWindows Library
PIN	10 Număr personal de identificare (*Personal Identification Number*)
RD	Proiectare recursivă (*Recursive Design*)
RDD	Proiectare orientată spre responsabilități (*Responsability-driven design*)
RFC	Răspunsul pentru o clasă (*Response for a Class*)
RTSA	Analiza structurată de timp real (*Real Time Structured Analysis*)
SCL	Sistem de coordonate liber (definit de utilizator)
TDA	Tip de date abstract
UML	Limbajul unificat de modelare (*Unified Modelling Language*)
VDT	Terminal clasic (*Video Display Terminal*)
WMC	Numărul ponderat de metode pe clasă (*Weighted Methods per Class*)

Notații

$\{I_i\}$	Set de variabile ale instanței din clasa C_i utilizate de metoda M_i
$2D_{continuu}$	Spațiu bidimensional cu metrică continuă, în care este descris modelul
$2D_{discret}$	Spațiu bidimensional cu metrică discretă, reprezentat de matricea de pixeli a dispozitivului de afișare
$3D_{continuu}$	Spațiu tridimensional cu metrică continuă, în care este descris modelul
$A \times B$	Produsul cartezian al mulțimilor A și B
C	Obiect de control
C_i	Clasa i
E	Obiect entitate
$f : A \rightarrow B$	Aplicație între mulțimile A și B
$G_x(V, E)$	Graf simplu unidirecțional cu noduri în mulțimea V și arce în mulțimea E
I	Obiect interfață
I_x	Set de variabile ale instanței pentru clasa X
M_j	Metoda j a clasei C_i
M_x	Setul de metode al clasei X
$O_{i,j}$	Obiectul de proiectare i cu atributul j
P_0	Punct situat în originea sistemului de coordonate
P_x	Punct situat pe axa Ox a sistemului de coordonate
P_y	Punct situat pe axa Oy a sistemului de coordonate

BUPT

P_z	Punct situat pe axa Oz a sistemului de coordonate
R	Expresie regulată ce defineşte limbajul Σ
S	Model de proiectare
SC_{global}	Sistemul de coordonate global
X	Clasă
x_{DA}, y_{DA}	Coordonatele X şi Y ale dispozitivului de afişare
x_F, y_F	Coordonatele X şi Y ale ferestrei
x_{SL}, y_{SL}	Coordonatele X şi Y ale spaţiului de lucru
ε	Caracterul vid
Σ	Alfabet al limbajului liniei de comandă

BUPT

Bibliografia

ABB83 — Abbott, R.J., *Program Design by Informal English Description*, Communication of the ACM, Vol. 26, No. 11, Nov. 1983

ACC93a — Advanced Concept Center, Martin Marietta, *Object-Modelling Technique: Object-Oriented Analysis*, Course Notes, Vol. 1, 2, Apr. 1993

ACC93b — Advanced Concept Center, Martin Marietta, *Object-Modelling Technique: Object-Oriented Design*, Course Notes, Vol. 1, 2, Apr. 1993

AHO77 — Aho, A.V., Ullman, J.D., *Principles of Compiler Design*, Addison-Wesley, 1977

BAC71 — Bachmann , K. H., ş.a., *Kleine Enzyklopedie der Mathematik*, VEB Bibliographisches Institut, 1971

BAU2000 — *Dietrich's Neues System*, Bauen mit Holz 6/2000

BEN82 — Ben-Ari, M., *Principles of Concurrent Programming*, Prentice Hall International, Englewood Cliffs, NJ, 1982

BOO86 — Booch, G., *Object-Oriented Development*, in IEEE Transactions on Software Engineering, Volume SE-12, No. 2, Feb. 1986

BOO87 — Booch, G., *Software engineering with Ada*, Second Edition, Benjamin / Cummings, 1987

BOO91 — Booch, G., *Object-Oriented Design with Applications*, Benjamin / Cummings, 1991

BOO93 — Booch, G., *Object-Oriented Analysis and Design with Applications*, Second Edition, Benjamin / Cummings, 1993

BOO99 — Booch, G., Rumbaugh, J., Jacobson, I., *The Unified Modelling Language User Guide*. Reading, MA. Addison Wesley Longman, Inc., 1999.

BUR99 — Burkhard, R.,*UML Objektorientierte Modellierung für die Praxis*, Addison-Wesley 1999

CHA93 — Champeaux D. de, et al., *Object-Oriented System Development*, Addison-Wesley, 1993

CLA93 — Clamage, S.D., *Debugging C++*, Object World Conference, San Francisco, June 17, 1993

BUPT

COA90 Coad, P., and Yourdon, E., *Object-Oriented Analysis*, Yourdon Press, 1990

COA91 Coad, P., and Yourdon, E., *Object-Oriented Analysis*, 2nd Edition, Yourdon Press, 1991

COA91a Coad, P., and Yourdon, E., *Object-Oriented Design*, Yourdon Press, 1991

COP92 Copelien, J.O., *Advanced C++*, Addison Wesley, Reading, MA, 1992

DAH70 Dahl, O., et al., *Simula 67 Common Base Language*, S-22, Norwegian Computing Center, Oslo, Norway, 1970

DEM79 DeMarco, T., *Structured Analysis and SystemSpecification*, Prentice Hall, Englewood Cliffs, NJ, 1979

DEU88 Deutsch, M.S., *Focusing Real Time Systems Analysis on User Operations*, IEEE Software, Sept. 1988

DEU91 Deutsch, M.S., and Nielsen, K.W., *User Scenarios and the Basis for System Architecture and Design*, Proceedings 13th International Conference on Software Engineering, Austin Texas, May 13-16, 1991

DIE99 Dietrich's AG, *DICAM – 3d CAD/CAM*, Lignarius, nov. 1999

DIJ68 Dijkstra, E.W., *Co-operating Sequential Processes in Programming Languages*, F.Genuys Ed., Academic Press, New York, NY, 1968

ETZ98 Etzkorn, L., Davis, C., Li, W., *A Practical Look at the Lack of Cohesion in Methods Metric*, Journal of Object-Oriented Design, September 1998 Vol. 11, No. 5

FAI93 Faison, T., *Object-Oriented State Machines*, Software Development, Sept. 1993

FAR94 Fara, H., Mireştean, M., Rus C., *Friendly Interface for Wood Manufacturing C.A.D.*, International Conference on Technical Informatics, Timişoara, 16-19. 11. 1994

FAR95 Fara, H., *Detecția erorilor in alocările dinamice de memorie*, Sesiunea de comunicări ştiinţifice ACADRES'95, Fundaţia universitară "Banatul", Universitatea Banatului, 5 mai 1995

FAR96 Fara, H., Rus, C., *Dialog Manager for Applications Running under MS-DOS*, Sesiunea de comunicări ştiinţifice ACADRES'96, Fundaţia universitară "Banatul", Universitatea Banatului Timişoara, 16 nov. 1996

FAR2000 Fara, H, *Exception-based Assertion Mechanism for Object-oriented Systems used in the Implementation of a 2d-Editor for a CAD/CAM System*, CONTI'2000: 4th International Conference on Technical Informatics, 12-13 oct. 2000, Timişoara, Romania

GEH84 Gehani, N., *Ada Concurrent Programming*, Prentice Hall, Englewood

BUPT

Cliffs, NJ, 1984

GOL83 Goldberg, A., and Robson, D., *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, MA, 1983

GOM84 Gomaa, H., *A software Design Method for Real-Time Systems*, Comm. ACM Vol. 27, No. 9, Sept. 1984

HAR87 Harel, D., *State Charts: A visual Formalism for Complex Systems*, Science of Computer Programming, Vol. 8., North-Holland, Amsterdam, 1987

HAT87 Hatley, D.J., and Pirbhai, I.A., *Strategies for Real-Time System Specification*, Dorset House, New York, NY, 1987

HAT97 Hatley, D.J., *System Development - A Practical Guide*, Dorset House, New York, NY, 1997

HIN94 Hines, K., and Sanders B., *Communications System Design with Object-Oriented Analysis / Recursive Design*, Object Magazine, March-April, 1994

HOA74 Hoare, C.A.R., *Monitors: An Operating Systems Structuring Concept*, Comm. ACM, Vol. 17, No. 10, Oct. 1974

HOL2000 *Grafische Bedienerführung*, Holzkurier, Heft 35, 31 August 2000

HOR98 Horton, I., *Introduction to Microsoft Visual C++ 6.0*, Wrox Press, 1998

JAC92 Jacobson, I., et al., *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, Workingham, England, 1992

KOR90 Korson, *Understanding Object-Oriented: A Unifying Paradigm*, Comm. ACM, Vol. 33, No. 9, Sept 1990

KRU98 Kruchten, Ph., *The Rational Unified Process - An Introduction*. Reading, MA. Addison Wesley Longman, Inc., 1998.

LAN93 Lang, N., *Shlaer-Mellor Object-Oriented Analysis Rule*, Software Engineering Notes, ACM Press, Vol. 18, No.1, Jan. 1993

LIA98 Liang, W-Y, O'Grady, P, *Design with objects: an approach to object-oriented design*, Computer-Aided Design, Vol. 30 Nr. 12, oct. 1998

LIP91 Lippman, S., *C++ Primer*, Second Edition, Addison-Wesley, Reading, MA, 1991

LIS86 Liskov, B. and Guttag, J., *Abstraction and Specification in Program Development*, The MIT Press, Cambridge, MA, 1986

MEL93 Mellor, S.J., *The Shlaer-Mellor Method*, Presentation at Hughes Aircraft Company Object-Week, El Segundo, CA, Apr., 1993

MEY88 Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, New York, 1988

BUPT

MIC97 Microsoft, *Introduction to C++, A Short Guide to Programming in C++*, Microsoft Corporation, 1997

MUR93 Murray, R.B., *C++ Strategies and Tactics*, Addison-Wesley, Reading, MA, 1993

MUR96 Mureşan, V., *Modelator de solide pentru proiectarea asistată de calculator*, Teza de doctorat, Universitatea "Politehnica" Timişoara, 1996

MUR99 Mureşan, I., Mureşan, V., Fara, H., Stoicănescu, C., Raţiu, D., *Abordare orientată pe obiect pentru implementarea unui sistem integrat de proiectare a construcţiilor din lemn*, Sesiune comunicări ştiinţifice „Zilele academice timişene, 1999

MUR99a Mureşan, I., Fara, H., Mihali, D., Simu, S., *Metode de generare automată a haşurilor în cadrul documentaţiei de execuţie generate de un sistem CAD/CAM.*, Sesiune comunicări ştiinţifice „Zilele academice timişene, 1999

NIE88 Nielsen, K.W. and Shumate K.C., *Designing Large Real-Time Systems with Ada*, McGraw-Hill, New York, 1988

NIE90 Nielsen K.W., *Ada in Distributed Real-Time Systems*, McGraw-Hill, New York, NY, 1990

NIE92 Nielsen, K.W., *Object-Oriented Design with Ada*, Bantam Books, New York 1992

NIE95 Nielsen, K.W., *Software Development with C++, Maximizing Reuse with Object Technology*, Academic Press, 1995

OMG99 Object Management Group, *OMG Unified Modelling Language Specification*, Version 1.3, June 1999

PAG80 Pages-Johnes, M., *The Practical Guide to Structured System Design*, Yourdon Press, New York, NY, 1980

PER90 Perry, D.E. and Kaiser, G.E., *Adequate Testing and Object-Oriented Programming*, Journal of Object-Oriented Programming, Jan./Feb. 1990

QUA98 Quatrani, T., *Visual Modelling with Rational Rose and UML*, Reading, MA. Addison Wesley Longman, Inc., 1998

QUA99 *DICAM – ein neues 3D-Fertigugssystem für den Holzbau*, Quadriga 3/99

RAD98 Rader, K., *Mixin-Style Programming in C++*, Copyright © 1998 Kirk Rader, http://www.kirkrader.com/examples/cpp/mixin.htm

ROW98 Rowlett, T., *Building an Object Process Around Use Cases*, Journal of Object-Oriented Programming, March/April 1998

BUPT

RUM91 Rumbaugh, J. et all., *Object-Oriented Modelling and Design*, Prentice Hall, 1991

SAV2000 Savii, G., Fara, H., *Object-oriented Development Process based upon Use Cases for a 2D Editor in a CAD/CAM System*, CONTI'2000: 4th International Conference on Technical Informatics, 12-13 oct. 2000, Timişoara, Romania

SCH2000 Schumacher, R., *Schiftmethoden im Vergleich*, Der Zimmermann, aug. 2000

SHL88 Shlaer, S. and Mellor, S., *Object Lifecycles: Modelling the World in Data*, Yourdon Press, 1988

SHL90 Shlaer, S. and Mellor, S., *Recursive Design*, Computer Language, Mar. 1990

SHL92 Shlaer, S. and Mellor, S., *Object Lifecycles: Modelling the World in States*, Yourdon Press, 1992

SHL93 Shlaer, S., et all., *Shlaer-Mellor Method*, 1993 Project Technology, Inc., 2560 Ninth Street, Suite 214, Berkley, CA, 94710

SHU92 Shumate, K. and Keller M., *Software Specification and Design: A Disciplined Approach for Real-Time Systems*, John Wiley, New York, NY, 1992

STR91 Stroustrup, B., *The C++, Programming Language*, Second Edition, Addison-Wesley, Reading, MA, 1991

TEX97 Texel, P. P., Williams, Ch., *Use Cases Combined With Booch/Omt/Uml: Process and Products*, Prentice Hall, 1997

WAR85 Ward, P.T. and Mellor, S.J., *Structured Development for Real-Time Systems*, Volumes 1-3, Yourdon Press, New York, NY, 1985

WIR90 Wirfs-Brock, R., et al, *Designing Object-Oriented Software*, Prentice-Hall, Englewood Cliffs, NJ, 1992

YOU79 Yourdon, E., and Constantine, L.L. *Structured Design*, Prentice-Hall, Englewood Cliffs, NJ, 1979

DHP01 ***, *Editor2D.doc*, Raport tehnic intern, Dietrich's AG, 1999-2001

DHP02 ***, *Bearbeit.doc*, Raport tehnic intern, Dietrich's AG, 1999-2001

DHP03 ***, *DiWand.doc*, Raport tehnic intern, Dietrich's AG, 1999-2001

DHP04 ***, *Dicam012.doc*, Raport tehnic intern, Dietrich's AG, 1999-2001

DHP05 ***, *Dicam013.doc*, Raport tehnic intern, Dietrich's AG, 1999-2001

BUPT