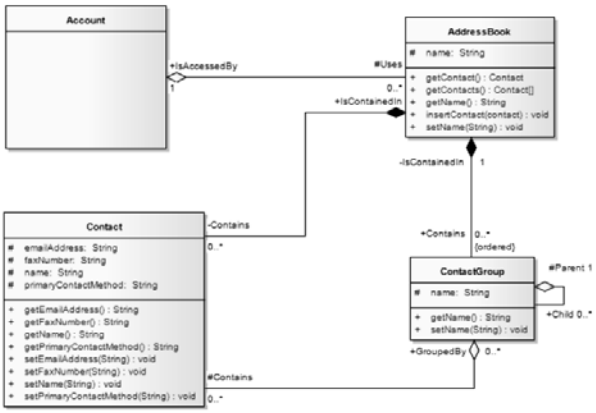
UNIVERSITATEA TEHNICĂ A MOLDOVEI

# INGINERIA PROGRAMĂRII

*Prezentare teoretică şi aplicaţii*




Chişinău
**2012**

UNIVERSITATEA TEHNICĂ A MOLDOVEI

Facultatea Calculatoare, Informatică şi Microelectronică

Catedra Calculatoare

# INGINERIA PROGRAMĂRII

*Prezentare teoretică şi aplicaţii*

Chişinău
**U.T.M.**
**2012**

Lucrarea respectivă este adresată studenţilor specialităţii 526.1 *Calculatoare*, anul III, ciclul I de licenţă, Facultatea *Calculatoare, Informatică şi Microelectronică* pentru a fi utilizată la însușirea disciplinei *Ingineria programării*.

În lucrare sunt prezentate unele aspecte teoretice privind principalele tipuri de diagrame UML şi probleme cu aplicaţii în mediul Enterprise Arhitect, precum și exemple pentru efectuarea lucrărilor de laborator cu tematica stabilită conform programului de învăţământ.

Autori: prof. univ. **Emilian Guţuleac**,
lector univ. **Diana Palii**,
lector sup. **Iurii Țurcanu**

Redactor responsabil: conf. univ., dr. **Sergiu Zaporojan**

Recenzent: conf. univ., dr. **Victor Ababii**

# Introducere

În lucrare sunt abordate unele subiecte legate de procesul dezvoltării și întreținerii unui produs software. Multe dintre tehnicile care se aplică în acest domeniu sunt similare celor utilizate de inginerii care lucrează în industrie, de exemplu de către constructorii de automobile, poduri sau televizoare. De altfel, domeniul se numește *inginerie software*.

În continuare ne vom referi la sisteme software de dimensiuni mari. Problemele care apar la dezvoltarea unor astfel de sisteme nu sunt asemănătoare celor cu care ne confruntăm la scrierea unei aplicații simple. În acest caz, apar probleme noi cum ar fi cele legate de faptul că dezvoltarea unui sistem complex presupune implicarea pe termen lung a unui mare număr de persoane, iar pe parcursul procesului specificațiile se pot modifica sau personalul se poate schimba (prin transfer, promovare etc.). Prin urmare, ingineria software se ocupă și de probleme legate de personal sau de managementul proiectelor, care sunt mai apropiate de domeniul științelor economice decât de informatică.

Problema fundamentală a ingineriei programării constă în *satisfacerea cerinţelor clientului*. Aceasta trebuie realizată într-un mod flexibil şi pe termen lung. Ingineria programării se ocupă de toate etapele dezvoltării programelor: de la extragerea cerinţelor de la client până la întreţinerea şi retragerea din folosinţă a produsului livrat. Pe lângă cerinţele funcţionale, clientul doreşte (de obicei) ca produsul finit să fie realizat cu costuri de producţie cât mai mici. De asemenea, este de dorit ca aceasta să aibă performanţe cît mai bune (uneori direct evaluabile) cu un cost de întreţinere cât mai mic, să fie livrat la timp şi să fie sigur.

În trecut aceste etape au fost ignorate, însă în prezent orice dezvoltator recunoaște importanţa lor, deoarece s-a dovedit că de acestea depinde producerea și refolosirea de software. Aceste faze trebuie să fie efectuate înainte de realizarea codului.

Pentru analiza și proiectarea programelor, s-au creat limbaje de modelare. Unul dintre acestea este limbajul de modelare unificat - UML (The Unified Modeling Language).

UML este succesorul propriu-zis al celor mai bune trei limbaje de modelare anterioare orientate pe obiecte (Booch, OMT *Object Modeling Technique*, and OOSE *Object Oriented Software Engineering*).

UML se constituie din combinarea acestor limbaje de modelare și în plus deține o expresivitate care ajută la rezolvarea problemelor de modelare pe care vechile limbaje nu o aveau. UML este un limbaj de modelare care oferă o exprimare grafică a structurii și comportamentului software [2].

Ingineria sistemelor software include:

• *metodologii de dezvoltare (analiză și proiectare) software* (OMT, OOSE, XP, AP etc.);

• *instrumente software* de analiză, proiectare, modelare, testare, validare, generare teste, generare cod etc. (CASE tools);

• *notații și limbaje vizuale* (UML).

# 1. Fazele ingineriei programării

Există patru faze fundamentale ale metodologiilor ingineriei programării:

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

*Faza de analiză.* Această fază defineşte *cerinţele* sistemului, independent de modul în care acestea vor fi îndeplinite. Aici se defineşte problema pe care clientul doreşte să o rezolve. Rezultatul acestei faze este *documentul cerinţelor*, care trebuie să precizeze clar *ce* trebuie construit.

Faza de analiză poate fi considerată o rafinare a detaliilor. Distincţia dintre detaliile de nivel înalt şi cele de nivel scăzut sunt puse mai bine în evidenţă de abordările *top-down* (unde se merge spre detaliile de nivel scăzut) şi *bottom-up* (care tind spre detaliile de nivel înalt).

Documentul cerinţelor poate fi realizat într-o manieră formală, bazată pe logica matematică, sau poate fi exprimat în limbaj natural. În mod tradiţional, el descrie *obiectele* din sistem şi *acţiunile* care pot fi realizate cu ajutorul obiectelor.

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

- *Obiecte:* documentul trebuie să definească mai întâi vocabularul sistemului, care este bazat în mare parte pe construcţii substantivale pentru identificarea pieselor, părţilor componente, constantelor, numelor şi a relaţiilor dintre acestea;
- *Acţiuni:* documentul trebuie să definească, de asemenea, acţiunile pe care trebuie să le îndeplinească sistemul şi care sunt sugerate în general de construcţii verbale. Exemple de acţiuni sunt: metodele, funcţiile sau procedurile;
- *Stări:* fiecare sistem trece printr-o serie de schimbări de

stare. Exemple de stări sunt: starea iniţială, cea finală sau stările de eroare. Cele mai multe stări depind de domeniul problemei. Stările sunt asociate cu atributele obiectelor sistemului. Un eveniment declanşează o tranziţie de stare care poate conduce la îndeplinirea unei acţiuni de către sistem;

- *Scenarii tipice:* un scenariu este o secvenţă de paşi urmaţi pentru îndeplinirea unui scop. Când sistemul este terminat şi aplicaţia este disponibilă, clientul trebuie să poată utiliza, într-o manieră cât mai facilă şi clar specificată, toate scenariile tipice ale aplicaţiei. Scenariile tipice trebuie să reprezinte majoritatea scenariilor de utilizare ale aplicaţiei. Ponderea acestora variază de la un sistem la altul, dar 90% se consideră o proporţie acceptabilă;
- *Scenarii atipice:* un scenariu atipic trebuie să fie îndeplinit de sistem numai în cazuri speciale. Clientul poate să spere, de exemplu, că o eroare neprevăzută este un eveniment atipic. Totuşi, sistemul trebuie să gestioneze un număr cât mai mare de categorii de erori, prin tehnici stabilite, precum tratarea excepţiilor, monitorizarea proceselor etc.;
- *Cerinţe incomplete sau nemonotone:* o enumerare completă a cerinţelor pentru toate situaţiile care pot apărea în condiţii de lucru reale nu este posibilă. Procesul de stabilire a cerinţelor are o natură iterativă şi nemonotonă. Mulţimea iniţială de cerinţe defineşte posibilităţile sistemului. Noile cerinţe pot infirma soluţiile vechi. Pe măsură ce un sistem creşte în dimensiuni şi complexitate, stabilirea cerinţelor devine din ce în ce mai dificilă, mai ales când procesul de colectare a cerinţelor este distribuit, fiind realizat de indivizi cu specializări diferite.

*Faza de proiectare.* În baza cerinţelor din faza de analiză, se va stabili *arhitectura* sistemului: componentele sistemului, interfeţele şi modul lor de comportare:

- *Componentele* sunt elementele constructive ale produsului. Acestea pot fi create de la zero sau reutilizate dintr-o

bibliotecă de componente. Componentele rafinează şi capturează semnificaţia detaliilor din documentul cerinţelor;

- *Interfeţele* ajută la îmbinarea componentelor. O interfaţă reprezintă graniţa dintre două componente, utilizată pentru comunicarea dintre acestea. Prin intermediul interfeţei, componentele pot interacţiona;
- *Comportamentul*, determinat de interfaţă, reprezintă răspunsul unei componente la stimulii acţiunilor altor componente.

Documentul de proiectare descrie planul de implementare a cerinţelor. Se identifică detaliile privind limbajele de programare, mediile de dezvoltare, dimensiunea memoriei, platforma, algoritmii, structurile de date, definiţiile de tip global, interfeţele etc.

În această fază trebuie indicate şi *priorităţile critice* pentru implementare. Acestea sugerează sarcinile care, dacă nu sunt executate corect, conduc la eşecul sistemului. Totuşi, chiar dacă priorităţile critice sunt îndeplinite, acest fapt nu duce automat la succesul sistemului, însă creşte nivelul de încredere al produsului.

*Analiza performanţelor* presupune studierea modului în care diferite arhitecturi conduc la diferite caracteristici de performanţă pentru fiecare scenariu tipic. În funcţie de frecvenţa de utilizare a scenariilor, fiecare arhitectură va avea avantaje şi dezavantaje. Un răspuns rapid la o acţiune a utilizatorului se realizează deseori pe baza unor costuri de resurse suplimentare: indecşi, managementul cache-ului, calcule predictive etc.

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

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

*Planul de test* defineşte testele necesare pentru stabilirea calităţii sistemului. Dacă produsul corespunde tuturor cerinţelor din

planul de test, este declarat terminat. *Acoperirea testului* este procentajul din produs verificat prin testare. În mod ideal, o acoperire de 100% ar fi excelentă, însă este rareori îndeplinită. De obicei, un test cu o acoperire de 90% este simplu, însă ultimele 10% necesită o perioadă de timp semnificativă.

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

Echipa trebuie să gestioneze problemele legate de calitate, performanţă, biblioteci şi debug. Scopul este producerea sistemului propriu-zis. O problemă importantă constă în *eliminarea erorilor erorilor critice*. Într-un sistem există trei tipuri de erori:

- *Erori critice:* împiedică sistemul să satisfacă în mod complet scenariile de utilizare. Aceste erori trebuie corectate înainte ca sistemul să fie predat clientului şi chiar înainte ca procesul de dezvoltare ulterioară a produsului să poată continua;
- *Erori necritice:* sunt cunoscute, dar prezenţa lor nu afectează în mod semnificativ calitatea observată a sistemului. De obicei, aceste erori sunt listate în notele de lansare şi au modalităţi de ocolire bine cunoscute;
- *Erori necunoscute:* există întotdeauna o probabilitate mare ca sistemul să conţină un număr de erori nedescoperite încă. Efectele acestor erori sunt necunoscute. Unele se pot dovedi a fi critice, altele pot fi rezolvate prin *patch* sau eliminate în versiunile ulterioare.

*Faza de testare.* În multe metodologii ale ingineriei programării, faza de testare este o fază separată, realizată de o echipă *diferită* după ce a luat sfârşit implementarea. Motivul constă în faptul că un programator nu-şi poate descoperi foarte uşor propriile greşeli. O persoană nouă, care priveşte codul, poate descoperi

greşeli evidente ce-i scapă celui care citeşte şi reciteşte materialul de multe ori. Din păcate, această practică poate determina o atitudine indiferentă faţă de calitate în echipa de implementare.

*Testele de regresiune* (engl. „regression test") sunt colecţii de programe care testează una sau mai multe trăsături ale sistemului. Rezultatele testelor sunt adunate şi dacă există erori, *bug*-ul este corectat. Un test de regresiune valid generează rezultate verificate, numite „standardul de aur". Validitatea rezultatului unui test ar trebui să fie determinată de documentul cerinţelor.

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

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

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

*Testarea unităţilor* testează o unitate ca un întreg. Aici se testează interacţiunea mai multor funcţii, dar numai în cadrul unei singure unităţi. Testarea este determinată de arhitectură. Aceste teste se mai numesc teste „black-box", deoarece numai detaliile interfeţei sunt vizibile pentru test.

*Testarea aplicaţiei* testează aplicaţia ca întreg şi este determinată de scenariile echipei de analiză. Aplicaţia trebuie să execute cu succes toate scenariile pentru a putea fi pusă la dispoziţia clientului. Spre deosebire de testarea internă şi a unităţilor, care se face prin program, testarea aplicaţiei se face de obicei cu scripturi care rulează sistemul cu o serie de parametri şi colectează rezultatele. În trecut, aceste scripturi erau create manual. În prezent, există instrumente care automatizează şi acest proces. Majoritatea aplicaţiilor din zilele noastre au interfeţe grafice utilizator (GUI).

Testarea interfeţei grafice pentru asigurarea calităţii poate pune anumite probleme. Cele mai multe interfeţe, dacă nu chiar toate, au bucle de evenimente, care conţin cozi de mesaje de la mouse, tastatură, ferestre etc. Asociate cu fiecare eveniment sunt coordonatele ecran. Testarea interfeţei presupune deci memorarea tuturor acestor informaţii şi elaborarea unei modalităţi prin care mesajele să fie trimise din nou aplicaţiei, la un moment ulterior.

*Testarea la stres* determină calitatea aplicaţiei în mediul său de execuţie. Aceasta este cea mai dificilă şi complexă categorie de teste. Sistemul este supus unor cerinţe din ce în ce mai numeroase, până când acesta cade. Apoi produsul este reparat şi testul de stres se repetă până când se atinge un nivel de stres mai ridicat decât nivelul aşteptat de pe staţia clientului.

# 2. Metodologii de dezvoltare

Fără o metodologie adecvată, dezvoltarea produsului este haotică. Unele companii folosesc propriile metodologii, altele folosesc metodologii comerciale.

Alegerea unei metodologii de dezvoltare trebuie să ţină cont de natura fiecărui proiect:

- ✓ Bugetul;
- ✓ Mărimea echipei;
- ✓ Tehnologia utilizată;
- ✓ Instrumentele şi metodele;
- ✓ Termenele-limită;
- ✓ Procesele existente deja.

## 2.1. Abordarea „codează şi repară”

Din engleză “code and fix” este cel mai des întâlnită metodologie, de fapt fiind cea mai rapidă şi mai puţin eficientă metodă de dezvoltare a aplicaţiilor. Aici nu există reguli stabilite, deseori se întâlneşte în companii mici, la început, cu puţini programatori.

## 2.2. Metodologia secvenţială

Metodologia secvenţială, cunoscută şi sub numele de metodologie „cascadă”, traversează mai întâi faza de analiză, apoi cea de proiectare, urmată de cea de implementare, iar în final se efectuează testarea. Echipele care se ocupă de fiecare fază pot fi diferite, iar la fiecare tranziţie de fază poate fi necesară o decizie managerială.


**Fig. 2.2.1**. Metodologia secvenţială

## 2.3. Metodologia ”cascadă”

Procesul „curge” de la etapă la etapă, ca apa într-o cascadă, în engleză “waterfall”. Modelul cascadă cu feedback propune remedierea problemelor descoperite în pasul precedent.

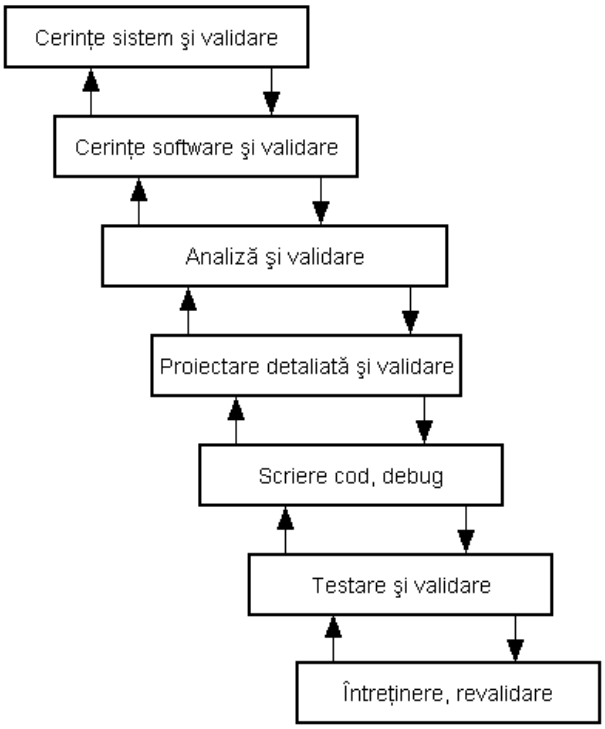



**Fig. 2.3.1.** Metodologia ”cascadă”

*Avantaje*

Metodologia secvenţială este potrivită când complexitatea sistemului este mică, iar cerinţele sunt statice. Forţează menţinerea unei discipline de lucru care evită presiunea scrierii codului înainte de a cunoaşte precis ce produs va trebui de fapt construit.

*Dezavantaje*:

Unul dintre principalele dezavantaje ale metodologiei secvenţiale este faptul că acordă o foarte mare importanţă fazei de analiză.

Comunicarea dintre echipe este o problemă: cele patru echipe pot fi diferite, iar comunicarea dintre ele este limitată. Modul principal de comunicare sunt documentele realizate de o echipă şi trimise următoarei echipe cu foarte puţin feedback.

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

Îngheţarea prematură a cerinţelor conduce la obţinerea unui produs insuficient structurat care nu execută ceea ce doreşte utilizatorul.

## 2.4. Metodologia ciclică

Metodologia ciclică, cunoscută şi sub numele de metodologia „spirală", încearcă să rezolve unele din problemele metodologiei secvenţiale. Această metodologie are patru faze, însă în fiecare fază se consumă mai puțin timp, după care urmează mai multe iteraţii prin toate fazele.

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

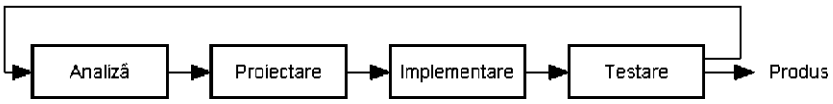



**Fig. 2.4.1.** Metodologia ciclică

## 2.5. Metodologia spirală

*Ciclul 1* – Analiza preliminară:

1. Obiective, alternative, constrângeri.
2. Analiza riscului şi prototipul.
3. Conceperea operaţiilor.
4. Cerinţele şi planul ciclului de viaţă.
5. Obiective, alternative, constrângeri.
6. Analiza riscului şi prototipul.

*Ciclul 2* – Analiza finală:

7. Simulare, modele, benchmark-uri.
8. Cerinţe software şi validare.
9. Plan de dezvoltare.
10. Obiective, alternative, constrângeri.
11. Analiza riscului şi prototipul.

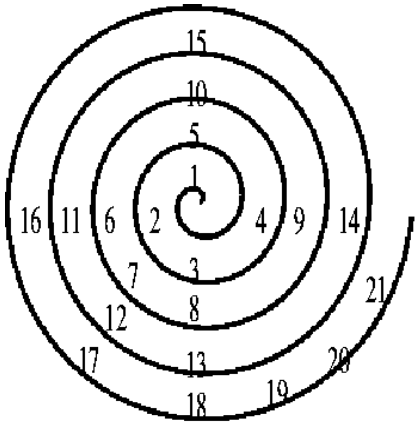



**Fig. 2.5.1.** Metodologia spirală

*Ciclul 3* – Proiectarea:

12. Simulare, modele, benchmark-uri.
13. Proiectarea produsului software, validare şi verificare.

14. Integrare şi plan de test.
15. Obiective, alternative, constrângeri.
16. Analiza riscului şi prototipul operaţional.

*Ciclul 4* – Implementarea şi testarea:

17. Simulare, modele, benchmark-uri.
18. Proiectare detaliată.
19. Cod.
20. Integrarea unităţilor şi testarea acceptării.
21. Lansarea produsului.

*Avantaje*

Metodologia ciclică se bazează pe ideea perfecţionării incrementale a metodologiei secvenţiale. Permite feedback-ul de la fiecare echipă în ceea ce priveşte complexitatea cerinţelor. Există etape în care pot fi corectate eventualele greşeli privind cerinţele. Clientul poate arunca o privire asupra rezultatului şi poate oferi informaţii importante mai ales în faza dinaintea lansării produsului. Echipa de implementare poate trimite echipei de analiză informaţii privind performanţele şi viabilitatea sistemului.

*Dezavantaje*

Metodologia ciclică nu are nici o modalitate de supraveghere, care să controleze oscilaţiile de la un ciclu la altul. În această situaţie, fiecare ciclu produce un efort mai mare de muncă pentru ciclul următor, ceea ce încarcă orarul planificat şi poate duce la eliminarea unor funcţii sau la o calitate scăzută. Lungimea sau numărul de cicluri poate creşte foarte mult. De vreme ce nu există constrângeri asupra echipei de analiză ca să facă lucrurile cum trebuie chiar de la început, acest fapt duce la scăderea responsabilităţii. Echipa de implementare poate primi sarcini la care ulterior se va renunţa. Echipa de proiectare nu are o viziune globală asupra produsului şi deci nu poate realiza o arhitectură completă. Nu există termene-limită precise. Ciclurile continuă fără o condiţie clară de terminare. Echipa de implementare poate fi pusă în situaţia

nedorită în care arhitectura şi cerinţele sistemului sunt în permanentă schimbare.

### 2.6. Metodologia hibridă ecluză

Metodologia ecluză (engl. „watersluice”), propusă de Ronald LeRoi Burback (1998), separă aspectele cele mai importante ale procesului de dezvoltare a unui produs software de detaliile mai puţin semnificative şi se concentrează asupra rezolvării primelor. Pe măsură ce procesul continuă, detaliile din ce în ce mai fine sunt rafinate, până când produsul poate fi lansat. Această metodologie hibridă preia natura iterativă a metodologiei spirală, la care adaugă progresul sigur al metodologiei cascadă.

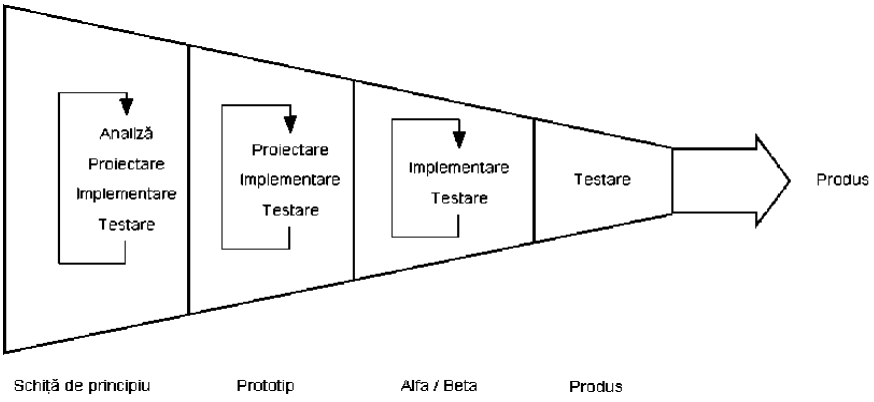



**Fig. 2.6.1.** Metodologia hibridă ecluză

*Etapele* principale ale metodei sunt: schiţa de principiu, prototipul, versiunile alfa/beta şi produsul finit.

În prima etapă, *schiţa de principiu*, echipele lucrează simultan la toate fazele problemei. Echipa de analiză sugerează cerinţele. Echipa de proiectare le discută şi trimite sarcinile critice echipei de implementare. Echipa de testare pregăteşte şi dezvoltă

mediul de test în funcţie de cerinţe. Echipa de implementare se concentrează asupra sarcinilor critice, care în general sunt cele mai dificile. Când componentele critice au fost realizate, sistemul este gata a face tranziţia spre stadiul de prototip.

În cea de a doua etapă, de *prototip*, cerinţele şi documentul cerinţelor sunt îngheţate. Schimbările în cerinţe sunt încă permise, însă ar trebuie să fie foarte rare şi numai dacă sunt absolut necesare, deoarece modificările cerinţelor în acest stadiu al proiectului sunt foarte costisitoare. Este posibilă totuşi ajustarea arhitecturii, pe baza noilor opţiuni datorită tehnologiei. După ce sarcinile critice au fost terminate, echipa de implementare se poate concentra asupra extinderii acestora, pentru definirea mai multor aspecte ale aplicaţiei.

Acum produsul este gata pentru lansarea versiunilor *alfa şi beta*. Arhitectura este îngheţată, iar accentul cade pe implementarea şi asigurarea calităţii. Prima versiune lansată se numeşte în general *alfa*. Produsul este încă imatur; numai sarcinile critice au fost implementate la calitate ridicată. O a doua lansare reprezintă versiunea *beta*. Rolul său este de a convinge clienţii că aplicaţia va fi un produs adevărat.

Când o parte suficient de mare din sistem a fost construită, poate fi lansat în sfârşit *produsul*. În această etapă, implementarea este îngheţată şi accentul cade pe asigurarea calităţii. Scopul este realizarea unui produs competitiv. În produsul finit nu se acceptă erori critice.

*Avantaje*

Metodologia ecluză acceptă faptul că oamenii fac greşeli şi că nici o decizie nu trebuie să fie absolută. Echipele nu sunt blocate într-o serie de cerinţe sau într-o arhitectură imobilă care se pot dovedi mai târziu inadecvate sau chiar greşite. Totuşi, pentru respectarea termenelor-limită, metodologia impune date de îngheţare a unor faze. Există timp suficient pentru corectarea greşelilor decizionale pentru atingerea unui nivel suficient de ridicat de încredere. Se pune mare accent pe comunicarea dintre echipe,

ceea ce reduce cantitatea de cod inutil la care ar trebui să se renunţe în mod normal.

*Dezavantaje*

Metodologia presupune asumarea unor responsabilităţi privind delimitarea etapelor şi îngheţarea succesivă a fazelor de dezvoltare. Ea presupune crearea unui mediu de lucru în care acceptarea responsabilităţii pentru o decizie care se dovedeşte mai târziu greşită să nu se repercuteze în mod negativ asupra individului. Se doreşte, de asemenea, schimbarea atitudinii echipelor faţă de testare, care are loc încă de la început, şi faţă de comunicarea continuă, care poate fi dificilă, întrucât cele patru faze reprezintă perspective diferite asupra realizării produsului.

## 2.7. Prototipizarea

Prototipul ajută clientul în a-şi defini mai bine cerinţele şi priorităţile. Prototipul este de obicei produs cât mai repede, deci, stilul de programare este de obicei neglijent. Se disting două feluri de prototipuri:

- → Abandonabil (“throw-away”) – doar obţinerea unei specificaţii;
- → Evoluţionar – crearea unui schelet al aplicaţiei.

*Avantaje*

Permite dezvoltatorilor să elimine lipsa de claritate a specificaţiilor. Oferă utilizatorilor şansa de a schimba specificaţiile într-un mod ce nu afectează grav durata de dezvoltare. Întreţinerea este redusă, deoarece validarea se face pe parcursul dezvoltării. Se poate facilita instruirea utilizatorilor finali înainte de terminarea produsului.

*Dezavantaje*

Deoarece prototipul rulează într-un mediu artificial, anumite dezavantaje ale produsului finit pot fi scăpate din vedere de clienţi. Clientul nu înţelege de ce produsul necesită timp suplimentar pentru

dezvoltare, având în vedere că prototipul a fost realizat atât de repede. Deoarece au în fiecare moment şansa de a face acest lucru, clienţii schimbă foarte des specificaţiile.

## 2.8. Modelul V

Pentru reglementarea dezvoltării de software, în administraţia federală germană a fost propus modelul "V-Modell", care evidenţiază testarea pe tot parcursul ciclului de dezvoltare. Trecerea la faza următoare se face numai după ce toate produsele din faza curentă au trecut testele de verificare şi validare. Procesul de verificare şi validare are scopul detectării cât mai multor erori în ciclul de dezvoltare.

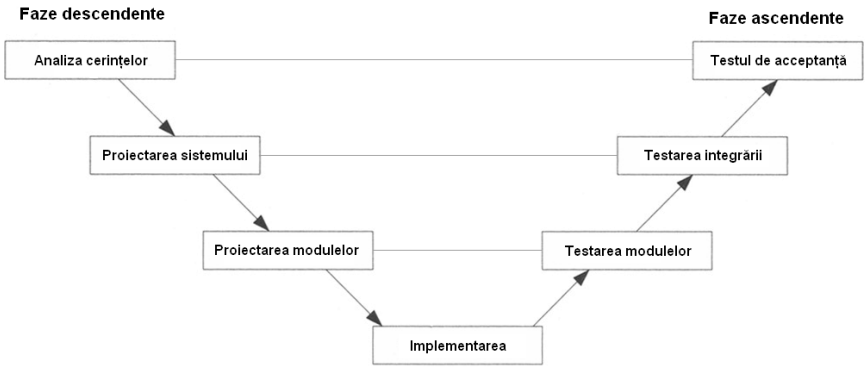



**Fig. 2.8.1.** Modelul V

## 2.9. Metoda de programare „agilă"

Metoda de programare „agilă" este potrivită dezvoltării rapide de aplicaţii.

***Principiile metodelor agile*:**

- *Implicarea clientului*

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

- *Livrarea incrementală*
  Programul este dezvoltat incremental. Clientul indică cerinţele care trebuie incluse la fiecare iteraţie.
- *Oamenii, nu procesul*
  Abilităţile echipei de dezvoltare trebuie recunoscute şi exploatate. Echipa trebuie lăsată să-şi contureze propriile modalităţi de lucru, fără a i se da reţete.
- *Acceptarea schimbării*
  Echipa trebuie să se aştepte ca cerinţele să se schimbe, iar proiectarea sistemului trebuie făcută astfel încât să se adapteze uşor la aceste schimbări.
- *Menţinerea simplităţii*
  Concentrarea asupra simplităţii atât în programele dezvoltate, cât şi în procesul de dezvoltare. Oricând este posibil, trebuie eliminată complexitatea din sistem

*Problemele metodelor agile.* Este dificilă menţinerea interesului clienţilor implicaţi în proces. Membrii echipei pot fi incapabili să se adapteze la implicarea intensă caracteristică metodelor agile. Când există mai mulţi factori de decizie, este dificil a schimba prioritatea lor. Menţinerea simplităţii necesită un lucru suplimentar.

## 2.10. Extreme Programming

*Extreme Programming* (XP) consideră că dezvoltarea programelor nu înseamnă ierarhii, responsabilităţi şi termene-limită, colaborarea oamenilor din care este formată echipa. Aceştia sunt încurajaţi să îşi afirme personalitatea, să ofere şi să primească cunoştinţe. XP consideră că dezvoltarea de programe înseamnă în primul rând scrierea de programe, nu documente, şedinţe şi rapoarte.

*Carta drepturilor dezvoltatorului*:

→ Ai dreptul să ştii ceea ce se cere, prin cerinţe clare, cu declaraţii clare de prioritate;

→ Ai dreptul să spui cât timp îţi va lua să implementezi fiecare cerinţă şi să îţi revizuieşti estimările în funcţie de experienţă;

→ Ai dreptul să îţi accepţi responsabilităţile, în loc ca acestea să-ţi fie atribuite;

→ Ai dreptul să munceşti calitativ în orice moment;

→ Ai dreptul la linişte, distracţie şi la muncă productivă şi plăcută.

*Carta drepturilor clientului:*

→ Ai dreptul la un plan general, să ştii ce poate fi făcut, când şi la ce preţ;

→ Ai dreptul să vezi progresul într-un sistem care rulează şi care se dovedeşte că funcţionează, trecând teste repetabile pe care le specifici tu;

→ Ai dreptul să te răzgândeşti, să înlocuieşti funcţionalităţi şi să schimbi priorităţile;

→ Ai dreptul să fii informat de schimbările în estimări suficient de devreme pentru a putea reduce cerinţele, astfel încât munca să se încheie la data prestabilită. Poţi chiar să te opreşti la un moment dat şi să rămâi cu un sistem folositor care să reflecte investiţia până la acea dată.

*Caracteristici:*

1. Echipa de dezvoltare nu are o structură ierarhică; fiecare contribuie la proiect folosind maximul din cunoştinţele sale.
2. Proiectul există în mintea tuturor programatorilor din echipă, nu în documentaţii, modele sau rapoarte.
3. Cerinţele clientului sunt exprimate ca scenarii sau povestiri (“user stories”).
4. Acestea sunt scrise pe bileţele, iar echipa de dezvoltare le împarte în sarcini de implementare (care stau la baza planificării orarului de lucru şi a estimării costurilor).

5. La orice moment, un reprezentant al clientului este disponibil pentru clarificarea cerinţelor.
6. Scrierea de cod este activitatea cea mai importantă.
7. La fiecare pas se face numai ce este absolut necesar în momentul următor.
8. Codul se scrie cât mai simplu şi se optimizează (refactoring) de câte ori este posibil.
9. Se scrie mai întâi codul de test.
10. Dacă apare necesitatea rescrierii sau eliminării codului, aceasta se face fără milă.
11. Modificările codului sunt integrate continuu (de câteva ori pe zi).
12. Se programează în echipă (programare în perechi); echipele se schimbă la sfârşitul unei iteraţii (1-2 săptămâni).
13. Se lucrează 40 de ore pe săptămână, fără un lucru suplimentar.

## 2.11. Metodologia Open Source

Open source = sursă la vedere. O abordare recentă, apărută ca urmare a dezvoltării mijloacelor de comunicaţie: FTP, e-mail, grupuri de discuţie. Exemple clasice: sistemul de operare Linux, browser-ul Netscape 5. Codul-sursă este transmis utilizatorului final într-o manieră nonproprietară (fără patent), pe baza unei licenţe open-source.

*Critici*

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

*O iteraţie tipică*

Dezvoltatorul realizează proiectarea şi codarea (individual sau în echipă). Ceilalţi dezvoltatori sau comunitatea de utilizatori realizează debugging-ul şi testarea. Noile funcţionalităţi dorite şi erorile depistate sunt trimise iniţiatorului proiectului. Se lansează o

nouă versiune cu erorile corectate şi se analizează noile cerinţe. Se distribuie o listă de sarcini către comunitatea de utilizatori, căutîndu-se membri voluntari care să execute sarcinile de pe listă.

## 2.12. Rational Unified Process

**Rational Unified Process (**RUP**)** este un proces iterativ de dezvoltare software creat de Rational Software şi dezvoltat de IBM după preluarea acestei companii în 2002. Varianta generică şi deschisă a RUP se numeşte Unified Process (UP). Rădăcinile RUP sunt ancorate în modelul spirală. RUP a fost rezultatul unui studiu amplu, care a căutat să determine cauzele comune ale eşecului proiectelor software, reprezentând de fapt o concentrare a celor mai bune practici de dezvoltare a sistemelor.
RUP se bazează pe un set de principii:

- dezvoltarea iterativă;
- managementul cerinţelor (identificarea şi specificarea corectă a tuturor nevoilor utilizatorilor, dar şi permanenta urmărire a modificării acestor nevoi);
- folosirea unei arhitecturi bazate pe componente;
- modelarea vizuală;
- verificarea calităţii software-ului produs;
- controlul modificărilor ulterioare.

Pentru determinarea corectă a cerinţelor RUP, se consideră necesare desfăşurarea următoarelor activităţi:

- analiza problemelor şi situaţiilor apărute în activitatea întreprinderii;
- înţelegerea corectă a nevoilor utilizatorilor şi managerilor;
- definirea sistemului (este necesară sublinierea importanţei diagramelor *use-case*);
- actualizarea permanentă a scopurilor urmărite de sistem şi rafinarea definirii sistemului (se ajunge la un document detaliat care specifică cerinţele software-ului).

Se foloseşte contextul general (UofD - Universe of Discourse) care este alcătuit din următoarele:

- modelul lexical care descrie vocabularul folosit;
- modele ale scenariilor posibile care descriu comportamentul sistemului;
- un model al regulilor de afaceri care descrie regulile organizaţiei;
- un hipertext view, o prezentare a configuraţiei şi un model de bază.

**Arhitectura bazată pe componente** permite dezvoltarea unui sistem care este uşor de înţeles, întreţinut şi extins.

RUP promovează dezvoltarea arhitecturii încă din primele iteraţii, arhitectură care devine un prototip și evoluează cu fiecare iteraţie pentru ca în final să se ajungă la arhitectura finală.

**Modelarea vizuală** – se face prin diagrame UML, presupune separarea clară a modelului conceptual al software-ului de codul propriu-zis pentru a menţine o viziune de ansamblu a proiectului,

Ciclul de viaţă al proiectului în RUP este împărţit în patru faze principale:

- faza de iniţializare a proiectului;
- faza de elaborare;
- faza de construcţie;
- faza de tranziţie.

În faza de iniţiere se stabileşte contextul general de dezvoltare al proiectului (completat de elemente cum ar fi diagrama *use-case* generală, planul proiectului, analiza riscurilor şi o descriere a proiectului), factorii de succes şi o prognoză financiară.

Pentru a considera această fază încheiată, trebuie satisfăcute următoarele criterii:

- acordul managerilor privind scopurile urmărite şi estimările de resurse necesare;
- fidelitatea diagramei *use-case* generale;
- credibilitatea estimării resurselor, priorităţilor şi riscurilor;
- calitatea prototipului;
- corelaţia dintre costurile estimate şi cele posibile.

În faza de elaborare se efectuează analiza domeniului şi se stabileşte o formă iniţială pentru arhitectura sistemului. Se urmăreşte satisfacerea următoarelor criterii:

- modelarea completă a cazurilor de utilizare şi identificarea completă a actorilor implicaţi;
- o descriere corectă a arhitecturii software (arhitectura sistemului presupune specificarea părţilor sistemului, a modurilor de interconectare ale acestora şi a regulilor de interacţiune dintre părţile sistemului);
- existenţa unui prototip de arhitectură care să încorporeze cazurile de utilizare semnificative;
- existenţa unui plan de dezvoltare globală pentru proiect.

În faza de construcţie se realizează în principal dezvoltarea componentelor şi a trăsăturilor sistemului. La finalul acestei faze se obţine un sistem funcţional care poate fi testat.

În cadrul fazei de tranziţie, produsul fazei anterioare se implementează, sistemul fiind pregătit pentru testare şi validare de către utilizatori. Se compară produsul cu documentaţia dezvoltată în faza de iniţiere. Dacă se constată discrepanţe între nevoile şi aşteptările utilizatorilor întreg procesul de dezvoltare se reia.

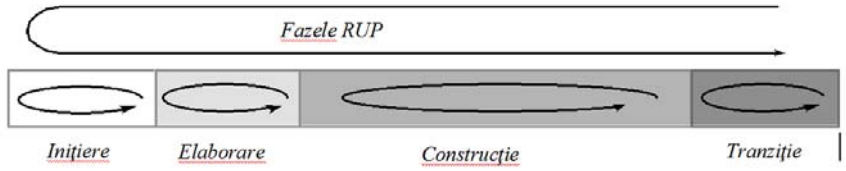



**Fig. 2.12.1.** Modelul fazelor RUP

*Fazele RUP*:

1. *Iniţiere* - stabilirea cazurilor de utilizare pentru sistem.
2. *Elaborare* - dezvoltarea unei înţelegeri asupra domeniului problemei şi asupra arhitecturii sistemului.
3. *Construcţie* - proiectarea, programarea şi testarea sistemului.
4. *Tranziţie* - instalarea sistemului în mediul lui de operare.

*Practici recomandate de RUP:*

- dezvoltarea iterativă a software-lui;
- managementul cerinţelor;
- folosirea arhitecturilor bazate pe componente;
- modelarea vizuală a software-lui;
- verificarea calităţii software-lui;
- controlarea modificărilor din software.

## 2.13. Reverse Engineering

*Reverse engineering = inginerie inversă*. Se analizează sistemele anterioare şi se încearcă reutilizarea componenelor existente Software, hardware, documentaţie şi metode de lucru. Unele componente nu pot fi utilizate, altele trebuie modificate (în faza de proiectare). Componentele selectate sunt integrate direct în sistem.

## 2.14. Metodologia de dezvoltare Offshore

*Offshore = în larg*. Companiile consideră *outsourcing*-ul pentru funcţiile care pot fi dezvoltate mai ieftin în altă ţară.

*Motive pentru outsourcing:*

1. Reducerea costurilor;
2. Creşterea eficienţei;
3. Concentrarea asupra obiectivelor critice ale proiectului;
4. Accesarea flexibilă a unor resurse care altfel nu ar fi accesibile (de ex., personal înalt calificat);
5. Dimensiunea mare a proiectului.

*Etape:*

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

# 3. UML - limbaj de modelare unificat

## 3.1. Generalităţi

Pentru analiza și proiectarea programelor s-au creat limbaje de modelare. Unul dintre aceste limbaje de modelare este limbajul de modelare unificat - UML (The Unified Modeling Language). UML nu este un simplu limbaj de modelare orientat pe obiecte, ci în prezent este limbajul universal standard pentru dezvoltatorii software din toata lumea. UML este succesorul propriu-zis al celor mai bune trei limbaje de modelare anterioare orientate pe obiecte (metoda Booch creata de Grady Booch, OMT *Object Modeling Techniques*, și OOSE *Object Oriented Software Engineering*). UML se constituie din combinarea acestor limbaje de modelare şi în plus are o expresivitate care ajuta la rezolvarea problemelor de modelare pe care vechile limbaje nu o aveau. Limbajul este destinat vizualizării, specificării, construirii şi documentării sistemelor de aplicaţii, dar are limitări în ceea ce priveşte generarea codului. UML reuneşte cele mai bune tehnici şi practici din domeniul ingineriei programării, care şi-au dovedit eficienţa în construirea sistemelor complexe.

UML este un limbaj de modelare care ofera o exprimare grafică a structurii și comportamentului software. Pentru aceasta exprimare grafica se utilizează notațiile UML. UML este un limbaj de modelare vizual, orientat pe obiect, care descrie (reprezintă) proprietăţile structurale şi dinamice ale unui sistem software. Fiecare tehnică de modelare dă o vedere diferită, statică sau dinamică a unei aplicaţii.

Notaţiile UML constituie un element esential al limbajului pentru realizarea propriu-zisă a modelarii, şi anume, partea reprezentării grafice pe care se bazează orice limbaj de modelare. Modelarea în acest limbaj se realizează prin combinarea notațiilor UML în cadrul elementelor principale ale acestora denumite diagrame. În cadrul UML descoperim 9 tipuri de diagrame: diagrama cazurilor de utilizare, diagrama de secvență, diagrama de

colaborare, diagrama de clase (cea mai utilizată), diagrama de stări, diagrama de componente, diagrama de obiecte, diagrama de activități, diagrama de desfășurare.

### 3.2. UML apariţie şi evoluţie

În octombrie 1994, Grady Booch, lider stiinţific al Rational Corporation, autor al metodei ce-i poartă numele şi a unor cărţi de referintă în domeniu, face echipă cu James Rumbaugh, autorul principal al metodei OMT, pe care-l determină să-şi părăsească (cel puţin temporar) vechiul loc de muncă (General Electric) şi să treacă la firma Rational. După un an de activitate, Booch şi Rumbaugh prezintă în octombrie 1995 la conferinţa OOPSLA caracteristicile de bază ale unei noi metode de analiză şi proiectare, rezultată prin unificarea metodei lui Booch (OOD) cu OMT, metodă denumită Metoda unificată (Unified Method). Prima documentaţie a metodei menţionate anterior a fost făcută publică în decembrie 1995, având numărul de versiune 0.8. La sfârșitul aceluiaşi an celor doi li se alatură şi Ivar Jacobson.

În iunie 1996, apare versiunea 0.9, urmată la scurt timp, octombrie 1996, de versiunea 0.91. Versiunea 0.9 aduce şi schimbarea denumirii din Metoda unificată (Unified Method) în Limbajul unificat de modelare (Unified Modeling Language). Cooptarea lui Jacobson în echipă se concretizează printre altele în detalierea conceptului de cazuri de utilizare (use case) şi prezentarea unei descrieri mai amănunţite a diagramelor cazurilor de utilizare. Conceptul de stereotip este mai bine explicitat, se modifică denumirile unor diagrame.

La 17 noiembrie 1997, OMG a anuntat adoptarea specificaţiei UML ca limbaj standard de modelare, schimbarea denumirii din Metodă unificată în Limbaj de Modelare Unificat
UML a fost conceput ca un limbaj universal care să fie utilizat la modelarea sistemelor (indiferent de tipul şi scopul pentru care au fost construite), la fel cum limbajele de programare sunt folosite în diverse domenii.

## 3.3. Tipuri de diagrame UML

Analiza unei aplicaţii implică realizarea mai multor categorii de modele, dintre care cele mai importante sunt:

*Modelul de utilizare* - realizează modelarea problemelor şi a soluţiilor acestora în maniera în care le percepe utilizatorul final al aplicaţiei. Diagrama asociată: diagrama cazurilor de utilizare.

*Modelul structural*- se realizează pe baza analizei statice a problemei şi descrie proprietăţile statice ale entităţilor care compun domeniul problemei. Diagrame asociate: diagrama de componente, diagramă de clase.

*Modelul comportamental* - priveşte descrierea funcţionalităţiilor şi a succesiunii în timp a acţiunilor realizate de entităţile domeniului problemei. Diagrame asociate: diagrama de stări, diagrama de colaborare, diagrama de interacţiune.

Principalele părţi ale UML sunt:

*Vederile (View)* **-** surprind aspecte particulare ale sistemului de modelat. Un view este o abstractizare a sistemului, iar pentru construirea lui se foloseşte un număr de diagrame.

*Diagramele* **-** sunt grafuri care descriu conţinutul unui view. UML are nouă tipuri de diagrame, care pot fi combinate pentru a forma toate view-urile sistemului.

*Elementele de modelare* **-** sunt conceptele folosite în diagrame care au corespondenţă în programarea orientată - obiect, cum ar fi: clase, obiecte, mesaje şi relaţii dintre acestea: asocierea, dependenţa, generalizarea. Un element de modelare poate fi folosit în mai multe diagrame diferite şi va avea acelaşi înteles şi acelaşi mod de reprezentare.

*Mecanismele generale* **-** permit introducerea de comentarii şi alte informaţii despre un anumit element.

Modelarea unui sistem poate fi o muncă foarte dificilă. Ideal ar fi ca pentru descrierea sistemului să se folosească un singur graf, însă de cele mai multe ori acesta nu poate să surprindă toate informaţiile necesare descrierii sistemului. Un sistem poate fi descris luând în considerare diferite aspecte:

- ✓ *Funcţional*: este descrisă structura statică şi comportamentul dinamic al sistemului;
- ✓ *Non-funcţional*: necesarul de timp pentru dezvoltarea sistemului;
- ✓ *Din punct de vedere organizatoric*: organizarea lucrului, maparea modulelor de cod. Aşadar, pentru descrierea unui sistem sunt necesare un număr de view-uri, fiecare reprezentând o proiecţie a descrierii intregului sistem şi care reflectă un anumuit aspect al sau.

Fiecare vedere (view) este descrisă folosind un număr de diagrame care conţin informaţii relative la un anumit aspect particular al sistemului. Aceste view-uri se acoperă unele pe altele, deci este posibil ca o anumită diagramă să facă parte din mai multe view-uri.

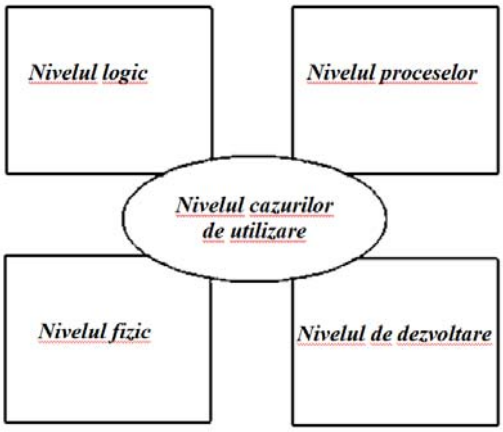



**Fig. 3.3.1.** Vederile UML

*Nivelul logic* – ne arată părţile din care este compus sistemul, precum şi interacţiunea lor. Aceasta reprezintă un set de abstractizări, care vor accentua clasele şi obiectele. Deci, putem spune că nivelul logic descrie obiectul model al sistemului. Diagrama UML care ne arată nivelul logic este *diagrama claselor*. De asemenea, *diagrama de stare*, *diagrama obiectelor*, *diagrama*

*secvenţială* şi *diagrama de colaborare*. Fiecare tip de diagramă are compartimentul său propriu.

*Nivelul proceselor* – acest nivel descrie procesele ce au loc în sistem. Nivelul proceselor arată comunicarea dintre aceste procese şi examinează ceea ce trebuie să se întâmple în sistem. Nivelul proceselor este deosebit de util, atunci când sistemul va avea un număr simultan de fire de execuţie sau procese, iar diagrama UML ce reprezintă nivelul proceselor este *diagrama de activităţi*.

*Nivelul fizic* acest nivel modelează mediul de execuţie a sistemului. Nivelul fizic constă în maparea artefactelor software pe hardware-ul care-l găzduieşte. Diagrama de desfăşurare UML este utilizată pentru a modela nivelul fizic al sistemului.

*Nivelul de dezvoltare* – descrie modulele sau componentele sistemului ce includ pachete, subsisteme şi biblioteci. Acest nivel ne oferă schema-bloc a sistemului. Modul în care este privit, nivelul de dezvoltare este util în gestionarea straturilor sistemului. Diagrama UML care ne arată nivelul de dezvoltare include: *diagrama componentelor* şi, de asemenea, *diagrama pachetelor*.

*Nivelul cazurilor de utilizare* arată funcţionalitatea sistemului. Cu alte cuvinte, acest nivel ne oferă o descriere generală a modului în care va fi utilizat sistemul. Nivelul cazurilor de utilizare capturează obiectivele şi scenariile utilizatorului. Aşadar s-ar putea spune, că descrie ceea ce face un sistem din punct de vedere al unui observator extern. Folosind cazurile de utilizare este util în a defini şi a explica structura şi funcţionalitatea sistemului în celelalte patru niveluri. Deci, acest nivel plus unu este un ghid pentru modelele din cele patru niveluri. Modelul patru plus unu este important, deoarece ne ajută să ne asigurăm că am reflectat şi documentat toate aspectele importante ale sistemului nostru.

*Diagramele funcţionale* se bazează exclusiv pe *cazurile de utilizare (use case diagram)* pentru specificarea cerinţelor sistemului. Modul de reprezentare a unei diagrame funcționale este ilustrat în figura 3.2 și are urmatoarea semnificatie: *Actorul* participa la *Cazul de utilizare* reprezentat în diagramă. Atât actorii,

cât și cazurile de utilizare trebuie să poarte denumiri unice. Între cazurile de utilizare există și anumite relații pe care le vom descrie ulterior. Cazurile de utilizare arată *ce* anume trebuie proiectat, fară a da vreo indicație cum să se facă acest lucru.

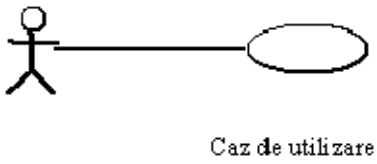



**Fig. 3.3.2.** Diagrama cazurilor de utilizare

### Diagramele statice

Diagramele *statice*, numite și diagrame *structurale*, arată legăturile dintre diferitele elemente de structură ale modelului.

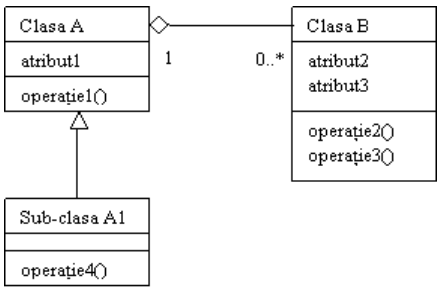



**Fig. 3.3.3.** Diagrama de clase

Pe de altă parte, clasa A este legată printr-o relație de *generalizare* de subclasa A1 care îi moștenește proprietățile (*atribut1* și *operaţie1*), având în plus *operație4*.

- Pentru analiză, diagrama de clase este utilă, deoarece descrie structura entităților manipulate de utilizator.
- În realizarea modelului, de obicei, printr-o diagramă de clase se reprezintă structura programelor orientate spre obiect sau, mai precis, modulele limbajului de dezvoltare.

*Diagramele de componente (component diagram)*, al căror mod de reprezentare este dat în figura 3.4, constituie concepte de configurare a programelor în pachete de programe, în fișiere-sursă sau în biblioteci.

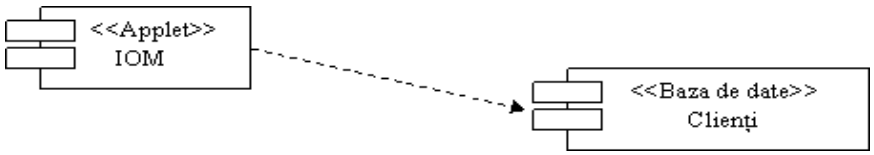



**Fig. 3.3.4.** Diagrama de componente

Aceste concepte arată cum se leagă între ele fișierele-sursă, pachetele de programe și bibliotecile, în cadrul sistemului informatic proiectat. Astfel, în figura menționată sunt reprezentate pachetul de programe de tip <<Applet>>, care cuprinde toate programele de interfață om-masină (*IOM*) și care comunică cu pachetul de programe de tip <<Baza de date>> numit *Clienţi*. Cele două pachete de programe pot fi amplasate pe mașini diferite sau în biblioteci diferite în cadrul sistemului informatic.

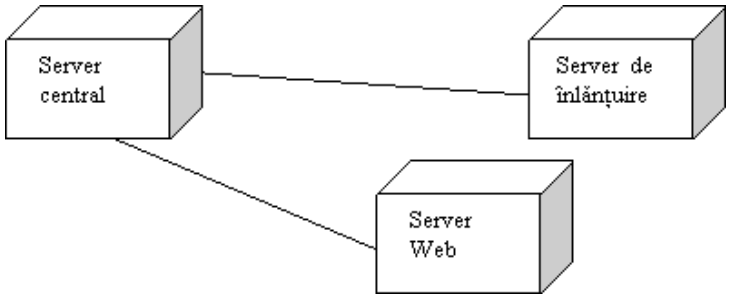



**Fig. 3.3.5.** Diagrama de desfășurare

Diagramele *de desfășurare* sau *(deployment diagram)*, al căror mod de reprezentare se observă în figura 3.5, corespund structurii de rețea informatică ce preia sistemul de programe și modul în care sunt instalate componentele de rețea. Astfel, din

figura 3.5 rezultă că sistemul local este constituit din serverul central, la care sunt legate un server de înlănțuire și un server Web.

### Diagramele dinamice (sau comportamentale)

Diagramele *dinamice*, numite și diagrame *comportamentale*, arată cum interacționează între ele diferite elemente ale modelului având următoarea componență:

- *Diagramele de activități (activities diagram)*, al căror mod de reprezentare este ilustrat în figura 3.6. Ele reprezintă regulile de înlănțuire ale activităților în cadrul sistemului, de exemplu, navigarea într-un site Web. Activitățile sunt reprezentate prin dreptunghiuri ovalizate, iar trecerea de la o activitate la alta prin săgeți, care se întâlnesc în noduri de stare marcate prin linii verticale. Ansamblul activităților are un punct de intrare și un punct de ieșire, marcate ca în figura 3.6.

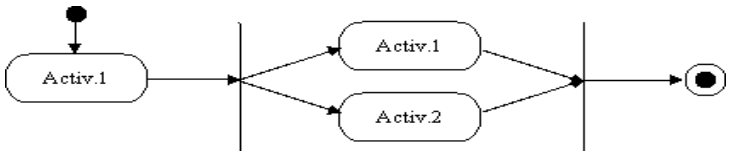



**Fig. 3.3.6.** Diagrama de activitate

- *Diagramele de stare (states diagram)*, al căror mod de reprezentare este dat în figura 3.7. Ele reprezintă ciclul de viață comun obiectelor aceleiași clase și permit completarea cunoștințelor claselor atât în cadrul analizei, cât și în cazul concepției. Convenția de reprezentare este inversă diagramelor de activități, adică stările sunt marcate prin dreptunghiuri cu colturi rotunjite, iar drumul de la o stare la alta prin săgeți care reprezentă acțiuni specifice. Ca și la diagramele de activități, există mai multe căi prin care se poate ajunge de la o stare la alta. Alegerea unei căi sau a alteia depinde de condițiile în care se afla sistemul la momentul respectiv. În

diagrama din figura 3.7 nu sunt menționate punctele și condițiile de alegere.

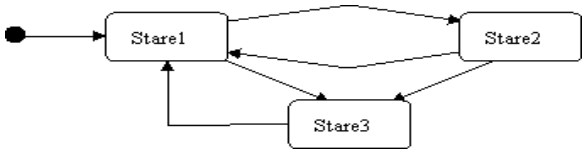



**Fig. 3.3.7.** Diagrama de stare

- *Diagramele de secvență (sequence diagram)*, al căror mod de reprezentare este ilustrat în figura 3.8. Ele servesc, în primul rând, dezvoltării de scenarii în cadrul analizei utilizării sistemului. În aceste diagrame, mesajele sunt reprezentate orizontal, pe o axă a timpului de sus în jos, de atâtea ori, de câte ori apar.

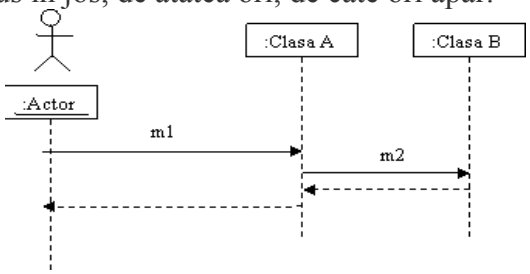



**Fig. 3.3.8.** Diagrama de secvență

- *Diagramele de colaborare (cooperation diagram)*, al căror mod de reprezentare este dat în figura 3.9.

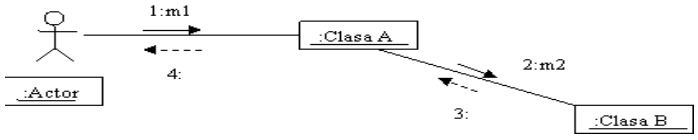



**Fig. 3.3.9.** Diagrama de colaborare

În diagramele de colaborare exista o singură cale, care unește doua elemente (clase, actori). Mesajele care circulă pe această cale împreună cu sensul lor sunt marcate pe margine cu săgeți. De obicei, diagramele de colaborare se construiesc pe baza diagramelor de secvență și arată schimburile de mesaje dintre obiecte în cazul anumitor funcționări particulare ale sistemului. Atât diagramele de secvență, cât și diagramele de colaborare sunt *diagrame de interacțiune UML*.

*Organizarea ierarhică a diagramelor* este prezentată în figura 3.10.

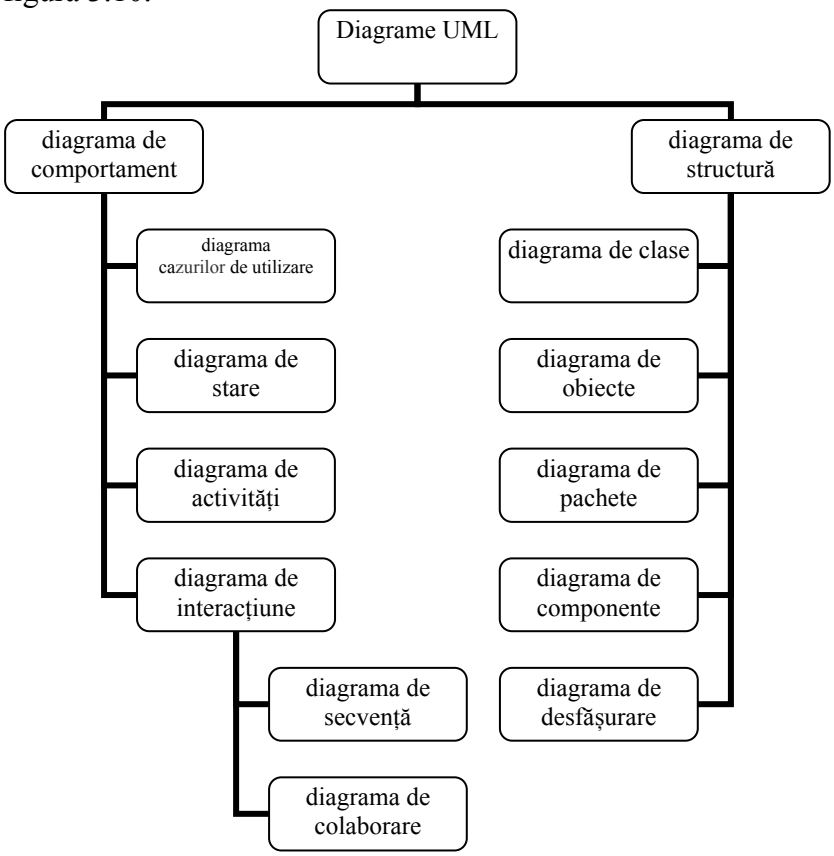



**Fig. 3.3.10.** Organizarea ierarhică a diagramelor

# 4. Lucrări de laborator

## 4.1. Lucrarea de laborator nr. 1

**Tema:** Elaborarea documentului de specificare a cerințelor

**Scopul:**

1. *Identificarea cerințelor funcționale și ne-funcționale ale sistemului.*
2. *Identificarea cerințelor utilizatorilor.*
3. *Identificarea cerințelor sistemului.*
4. *Elaborarea documentului de specificare a cerințelor.*

### Consideraţii teoretice

Ingineria cerințelor este procesul de stabilire a *serviciilor* cerute sistemului de către clienți, precum și a *constrângerilor* în care acesta va fi dezvoltat și va opera.

*Cerințele* sunt descrieri ale serviciilor oferite de sistem și ale constrângerilor care sunt generate de-a lungul desfășurării procesului de inginerie a cerințelor.

*Cerinţe funcţionale.* Sunt descrieri ale serviciilor pe care trebuie să le ofere sistemul, cum trebuie să reacţioneze şi să se comporte sistemul în situaţii particulare.

*Exemplu.* Fie un sistem tip bibliotecă, unde utilizatorii pot căuta, adăuga şi tipări articole pentru studiu personal. Exemple de cerinţe funcţionale pot fi:

- *Utilizatorul trebuie să poată căuta articole în toate bazele de date sau într-un subset selectat.*
- *Sistemul trebuie să ofere instrumente de vizualizare potrivite pentru citirea articolelor.*
- *Fiecare comandă trebuie să aibă alocat un identificator unic (ORDER_ID).*

*Cerinţe nefuncţionale.* Acestea definesc proprietăţile şi constrângerile sistemului cum ar fi: fiabilitatea, timp de răspuns şi

cerinţe de stocare. Constrângerile sunt proprietăţi ale componentelor de stocare de date, reprezentări ale sistemului etc.

Constrângeri asupra serviciilor sau funcţiilor oferite de sistem cum ar fi constrângerile de timp, constrângeri ale procesului de dezvoltare, standarde etc. Pot fi specificate cerinţe de proces care impun un anumit sistem CASE, un limbaj de programare sau o metodă de dezvoltare.

*Exemplu. Fiabilitatea, timpul de răspuns, cerințe pentru spațiul de stocare, cerințe ale sistemului de intrări-ieşiri etc.*

Cerințele nefuncționale pot fi mai critice decât cele funcționale. Dacă nu vor fi îndeplinite sistemul nu va fi util scopului în care a fost dezvoltat.

***Cerinţe nefuncţionale ce țin de următoarele**:*

- *Transport:* defineşte constrângerile şi cerinţele care afectează transmiterea de informaţii între noduri. Rețele, relee, protocoale, calitatea serviciilor de transport, chiar şi transmiterea pe suporturi fizice este inclusă aici.
- *Persistenţa*: detalizează criteriile operaţionale şi performanţa cu privire la stocarea de informaţii, inclusiv redundanţă, back-up-ul, sistemul de baze de date, fişierele şi alte mecanisme de stocare persistentă.
- *Securitate:* cerinţe privind accesul la date (securitatea informaţiilor) şi de securitate fizică (de acces la servere şi alte componente hardware critice).
- *Scalabilitate:* definește parametrii de funcţionare în ceea ce priveşte dimensiunea sistemului, numărul de tranzacţii, capacitatea, numărul de utilizatori şi noduri distribuite.
- *Performanţa:* definește parametri, cum ar fi tranzacţii pe secundă, viteza de transmitere în reţea, forma de încărcare şi alte aspecte măsurabile ale sistemului care guvernează viteza de ansamblu şi reacţia sistemului.

***Cerinţe de produs***

*Interfaţa cu utilizatorul pentru sistemul tip bibliotecă trebuie să fie implementată ca pagini HTML simple fără cadre (frames) sau Java applets.*

***Cerinţe externe.*** *Sistemul nu trebuie să ofere alte informaţii personale legate de clienţi în afară de numele lor.*

***Cerințele sistemului*** reprezintă un document structurat stabilind descrierea detaliată a funcțiilor sistemului, serviciile oferite și constrângerile operaționale. Poate fi parte a contractului cu clientul.

**Cerinţe utilizator.** Cerințele utilizatorului trebuie să descrie cerințele funcționale și nefuncționale într-o manieră în care sunt pe înțelesul utilizatorilor sistemului, care nu dețin cunoștințe tehnice detaliate. Ele sunt definite utilizând limbajul natural, tabele și diagrame care pot fi înțelese de toți utilizatorii.

Arhitectura sistemului se proiectează în baza cerințelor. Sistemul poate interopera cu alte sisteme care generează la rândul lor alte cerințe.

**Documentul de specificare a cerinţelor**

1. Documentul de cerinţe este exprimarea oficială a ceea ce se cere de la dezvoltatorii sistemului.
2. Trebuie să includă atât definiţii ale cerinţelor utilizator, cât şi specificaţii ale cerinţelor de sistem.
3. Acesta NU este un document de proiectare. Pe cât este posibil, trebuie să exprime CE trebuie să facă sistemul şi nu CUM trebuie să facă.

Structura generică a documentului de cerinţe ce trebuie creat pentru un sistem specific trebuie să corespundă unui standard, de exemplu, standardul IEEE 830-1998:

1. Introducere.
2. Descriere generală.
3. Cerinţe specifice.

4. Anexe.
5. Index.

## Concluzii

1. Cerinţele exprimă ceea ce trebuie să facă sistemul şi defineşte constrângerile asupra operării sau implementării.
2. Cerinţele funcţionale exprimă serviciile pe care le va oferi sistemul.
3. Cerinţele nefuncţionale constrâng sistemul sau procesul de dezvoltare.
4. Cerinţele utilizator sunt exprimări la nivel înalt a ceea ce trebuie să facă sistemul. Cerinţele utilizatorului trebuie scrise în limbaj natural, eventual folosind tabele şi diagrame.
5. Cerinţele de sistem trebuie să descrie funcţiile oferite de sistem.
6. Un document de cerinţe software este o exprimare oficială a cerinţelor sistemului.
7. Standardul IEEE 830-1998 este un punct de start pentru definirea unor standarde mai detaliate de specificare a cerinţelor.

## Desfășurarea lucrării:

1. Descrierea generala a scopului produsului.
2. Descrierea mediului de operare: echipamentele (hardware) și sistemul de operare.
3. Mediul de dezvoltare care urmează a fi utilizat.
4. Dacă produsul este un sistem independent sau o parte a unui sistem mai mare. Caracteristicile esențiale ale acestui sistem.
5. Descrierea modelului logic al sistemului.
6. Lista cerințelor funcționale.
7. Lista cerințelor nefuncționale.
8. Elaborarea diagramei cerințelor.

## 4.2. Lucrarea de laborator nr. 2

**Tema:** Diagrama cazurilor de utilizare

**Scopul:**

1. *Identificarea actorilor.*
2. *Identificarea scenariilor.*
3. *Determinarea cazurilor de utilizare.*
4. *Elaborarea diagramei cazurilor de utilizare.*

### Consideraţii teoretice

Modelarea vizuală în UML poate fi reprezentată ca un oarecare proces de lansare pe niveluri: de la cel mai general şi abstract model conceptual al sistemului iniţial, spre modelul logic, apoi spre cel fizic, ce corespunde unui sistem de program. Pentru atingerea acestui scop, inițial se creează un model în formă de diagaramă a cazurilor de utilizare (*use case diagram*) care descrie destinaţia funcţională a sistemului sau, cu alte cuvinte, descrie ceea ce va executa sistemul în procesul de funcţionare. Diagrama cazurilor de utilizare reprezintă un model iniţial conceptual al unui sistem în procesul de proiectare şi exploatare.

Esenţa acestei diagrame constă în faptul că sistemul proiectat reprezintă o colecţie de entităţi şi actori care colaborează cu sistemul prin aşa-numitele cazuri de utilizare. Totodată, orice entitate care colaborează cu sistemul din afară poate fi numită *actor*. Aceasta poate fi o persoană, o instalaţie tehnică, un program sau oricare alt sistem care poate servi drept sursă de acţiune pentru sistemul proiectat aşa, cum îl determină colaboratorul. La rîndul său, use *case-ul* este creat pentru descrierea serviciilor pe care le oferă sistemul actorului. Cu alte cuvinte, fiecare caz de utilizare defineşte o colecţie de acţiuni executate de sistem în timpul dialogului cu actorul. Însă nu este explicat modul în care va fi realizată colaborarea dintre actori şi sistem.

În general, diagrama cazurilor de utilizare reprezintă un graf deosebit, care este o notaţie grafică pentru prezentarea cazurilor de utilizare concrete, a actorilor sau a unor interfeţe şi pentru

prezentarea legăturilor dintre aceste elemente. Totodată, componente aparte ale diagramei pot fi incluse într-un dreptunghi, care semnifică sistemul proiectat în întregime. Trebuie de specificat că legăturile acestui graf pot fi de anumite tipuri de interacţiuni între actori şi cazuri de utilizare, care împreună descriu servicii şi cerinţe funcţionale faţă de sistemul modelat.

O diagrama a cazurilor de utilizare (*use case diagram*) reprezintă o colecție de cazuri de utilizare și actori care:

- oferă o descriere generală a modului în care va fi utilizat sistemul;
- furnizează o privire de ansamblu a funcționalităților ce se doresc a fi oferite de sistem;
- arată cum interacționează sistemul cu unul sau mai mulți actori;
- asigură faptul că sistemul va produce ceea ce s-a dorit.

*Caz de utilizare*

Structura sau elementul standard al limbajului UML se foloseşte pentru specificarea particularităţilor comune ale comportării unui sistem sau a oricărei alte entităţi în domeniul de lucru fără cercetarea structurii interne a acestei entităţi. Fiecare caz de utilizare determină o succesiune de acţiuni care trebuie să fie executate de către sistemul proiectat la colaborarea lui cu actorul corespunzător. Diagrama cazurilor de utilizare poate fi completată cu um text explicativ, care desfăşoară sensul sau semantica componentelor ce o formează. Acest text se numeşte adnotare sau scenariu. Cazul de utilizare se notează cu o elipsă în interiorul căreia se introduce denumirea prescurtată sau numele în formă de verb cu explicație (Fig. 4.2.1).

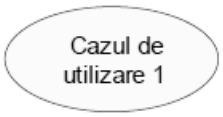



**Fig. 4.2.1.** Caz de utilizare

Scopul cazului de utilizare constă în determinarea aspectului terminal sau al fragmentului de comportare a unei entităţi fără desfăşurarea structurii interne a acestei entităţi. Drept astfel de entitate poate servi un sistem iniţial sau un element al modelului care dispune de um comportament propriu, precum este subsistemul sau clasa în modelul unui sistem.

Fiecare caz de utilizare corespunde unui serviciu aparte, care reprezintă o entitate modelată sau un sistem la cererea utilizatorului (actorului), mai precis determină metoda aplicată către o anumită entitate. Serviciul, care este iniţializat la cererea utilizatorului, reprezintă o succesiune terminată de acţiuni. Aceasta înseamnă că după ce sistemul va termina prelucrarea cererii utilizatorului el (sistemul) trebuie să revină la starea iniţială în care este gata pentru a indeplini cererile următoare.

Cazurile de utilizare descriu nu numai colaborarea dintre utilizatori şi entităţi, dar şi reacţia entităţii la primirea anumitor mesaje de la utilizatori şi asupra percepţiei acestor mesaje în afara entităţii. Cazurile de utilizare pot include descrierea specificaţiilor modurilor de realizare a serviciului şi a diferitor situaţii excepţionale, aşa cum este prelucrarea corectă a erorilor unui sistem. Mulţimea cazurilor de utilizare poate determina toate aspectele posibile ale comportării aşteptate a unui sistem. Pentru comoditate mulţimea cazurilor de utilizare poate fi considerată ca un pachet aparte.

Din punct de vedere sistemo-analitic, cazurile de utilizare pot fi folosite pentru specificarea cerinţelor externe către sistemul proiectat şi pentru specificarea comportării funcţionale a sistemului deja existent.

Exemple de cazuri de utilizare pot fi următoarele acţiuni: verificarea stării contului curent al clientului; întocmirea comenzii la procurarea marcii; obţinerea informaţiei suplimentare despre solvabilitatea clientului; reprezentarea formei grafice pe ecranul monitorului ş.a.

*Actorul.* Actorul reprezintă orice entitate externă a sistemului modelat, care colaborează cu sistemul şi utilizează

posibilităţile lui funcţionale pentru atingerea anumitor scopuri şi pentru rezolvarea problemelor particulare. Totodată, actorii sunt utilizaţi pentru notarea mulţimii rolurilor coordonate ale utilizatorilor în procesul de colaborare cu sistemul proiectat. Fiecare actor poate avea un rol aparte corespunzător unui caz de utilizare concret. Notaţia grafică standard a unui actor în diagramă este ”omuleţul” sub care se indică numele actorului (Fig. 4.2.2).

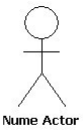



**Fig. 4.2.2**. Actor

*Identificarea actorilor* se face raspunzând la urmatoarele întrebări:

- Cine doreşte sau este interesat de informaţiile aflate în sistem?
- Cine modifică datele?
- Cine interacţionează cu sistemul?

În unele cazuri, actorul poate fi notat ca dreptunghiul clasei cu cuvântul-cheie ”actor” şi cu elementele comune ale clasei. Numele actorilor trebuie să fie scrise cu litere majuscule şi trebuie să respecte recomandările la utilizarea numelor pentru tipurile şi clasele modelului. Totodată, simbolul actorului aparte leagă descrierea corespunzătoare a actorului cu un anumit nume. Numele actorilor abstracţi, precum şi a altor elemente abstracte în limbajul UML, se recomandă a fi notate cu italic.

Exemplu de actori pot servi: clientul unei bănci; angajatul unei bănci; vânzătorul unui magazin; managerul secţiei de vânzare; pasagerul unui avion; conductorul unui auto; administratorul unui hotel şi alte entităţi, care au legătură cu modelul conceptual ce corespunde domeniului de lucru.

Actorii sunt utilizaţi pentru modelarea entităţilor externe ale sistemului, care acţionează reciproc cu sistemul şi pe care îl va utiliza în mod separat. În calitate de actori pot servi şi alte subsisteme ale sistemului proiectat sau clase aparte. Este important

să înţelegem că actorul defineşte o anumită mulţime de roluri coordonate, care pot fi utilizatorii unui sistem dat în procesul de colaborare. În fiecare moment de timp cu sistemul colaborează un anumit utilizator în unul din roluri date. Drept exemplu evident de actor poate servi un anumit utilizator al sistemului cu parametri de autentificare proprie.

*Legăturile în diagrama cazurilor de utilizare.* Între componentele diagramei cazurilor de utilizare pot să existe diferite legături care descriu colaborarea exemplarelor unor actori şi cazurilor de utilizare cu exemplarele altor actori şi cazuri. Un anumit actor poate să colaboreze cu mai multe cazuri de utilizare. În acest caz, actorul dat se adresează câtorva servicii ale sistemului dat. La rândul său, un anumit caz de utilizare poate să colaboreze cu mai mulţi actori, pentru care acordă serviciul său. Trebuie de observat că două cazuri de utilizare definite pentru aceeaşi entitate nu pot colabora unul cu altul, deoarece fiecare din ele descrie o variantă proprie terminală de utilizare a acestei entităţi. Mai mult, cazurile de utilizare întotdeauna presupun careva semnale şi mesaje pentru colaborarea cu actorii în afara limitelor sistemului. În acelaşi timp, pot fi definite alte metode de colaborare între elemente din interiorul sistemului.

În limbajul UML există câteva tipuri standard de relaţii între actori şi cazuri de utilizare:

- relaţia de asociere (association relationship);
- relaţia de extindere (extend relationship);
- relaţia de generalizare (generalization relationship);
- relaţia de cuplare (include relationship).

Un actor este un stereotip al unei clase. Actorii sunt reprezentaţi de utilizatori sau entităţi care pot interacţiona cu sistemul. Ei nu fac parte din sistem şi definesc mulţimi de roluri în comunicarea cu acesta.

*Relaţia de asociere.* Relaţia de asociere este o noţiune fundamentală în limbajul UML şi mai mult sau mai puţin se utilizează la crearea tuturor modelelor grafice în forma diagramelor canonice.

În ceea ce privește diagrama cazurilor de utilizare, relaţia de asociere specifică rolul deosebit al actorului în cazul de utilizare aparte. Cu alte cuvinte, asocierea specifică particularitatea semantică de colaborare a actorilor şi cazurilor de utilizare în modelul grafic. Astfel, această relaţie stabileşte ce rol joacă actorul la colaborarea cu exemplarul cazului de utilizare. În diagrama cazurilor de utilizare, precum şi în alte diagrame relaţia de asociere se notează cu o linie neîntreruptă între actor şi cazul de utilizare. Această linie poate avea condiţii suplementare, de exemplu, numele şi multiplicitatea (Fig. 4.2.3).


**Fig. 4.2.3.** Relaţie de asociere dintre acotr şi cazul de utilizare

*Relaţia de extindere.* Relaţia de extindere defineşte interconexiunea exemplarelor cazului de utilizare cu cazul general, proprietăţile căruia sunt definite pe baza modului de uniune a exemplarelor date. Dacă are loc relaţie de extindere de la cazul de utilizare A la cazul de utilizare B, acest lucru înseamnă că proprietăţile exemplarului cazului de utilizare B pot fi adăugate datorită proprietăţilor extinse a cazului de utilizare A.

Relaţia de extindere între cazurile de utilizare reprezintă o linie punctată cu săgeată (cazul relaţiei de dependenţă) directă de la acel caz de utilizare, care reprezintă extinderea cazului de utilizare iniţial. Această linie cu săgeată este marcată cu cuvântul „extend” („extinde”) (Fig. 4.2.4).

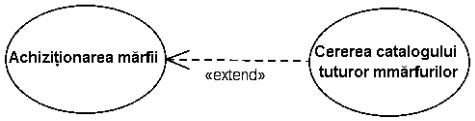



**Fig. 4.2.4.** Relaţia de extindere dintre cazurile de utilizare.

Relaţia de extindere denotă că la comportamentul unuia din cazurile de utilizare poate fi conectat un comportament adăugător, definit pentru un alt caz de utilizare. Relaţia dată cere o anumită condiţie indicată la punctele de extensie specificate în cazul de utilizare de bază. Pentru extindere este necesar de a se îndeplini condiţia specificată relaţiei date. Punctele de extensie definesc locul în care use case-ul specializat (B) extinde use case-ul de bază (A).

Unul din cazurile de utilizare poate fi extinderea pentru câteva cazuri de bază, având ca extinderi proprii alte cazuri. Cazul de utilizare de bază poate fi adăugător independent de alte extensii.

Semantica relaţiei de extensie este definită astfel: dacă exemplarul cazului de utilizare execută anumite acţiuni consecvente, care defineşte comportamentul lui şi în urma căruia există un punct de extensie la alt exemplar al cazului de utilizare, care este unul din primele puncte de extensie a cazului iniţial, verifându-se condiţia relaţiei date. Dacă condiţia este executată, consecvenţa iniţială de acţiuni extinde includerea acţiunii altui exemplar a cazului de utilizare.

*Relaţia de generalizare.* Relaţia de generalizare este folosită pentru indicarea faptului că un caz de utilizare A poate fi generalizat la cazul de utilizare B. În urma acestui fapt, cazul A va fi un caz special al cazului B, iar cazul B se va numi părinte relativ A, cazul A – descendent relativ cazului de utilizare B.

De menţionat că descendentul moşteneşte toate proprietăţile şi comportamentul părintelui său şi poate avea proprietăţile şi comportamentul său adăugător. Grafic relaţia dată este reprezentată prin linie continuă cu săgeată în forma de triunghi nehaşurat, care indică cazul de utilizare părinte (Fig. 4.2.5). Această linie cu săgeată are un nume specific – săgeata „generalizare”.

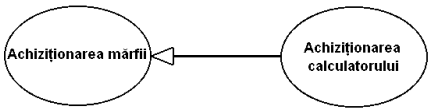



**Fig. 4.2.5.** Relaţia de generalizare dintre cazurile de utilizare

Relaţia de generalizare dintre cazurile de utilizare este folosită în cazurile când este necesar a indica că acest caz de utilizare derivat conţine toate atributele şi particularităţile comportamentului cazurilor de bază. Datorită căruia cazurile de utilizare derivate pot participa în relaţii cu cazurile de bază. În urma acestui fapt cazurile derivate pot obține proprietăţi noi de comportament, pe care nu le au cazurile de utilizare de bază, dar şi de a preciza sau a modifica proprietăţile de comportament moştenite.

Între actori aparte, de asemenea, poate exista relaţia de generalizare. Această relaţie este directă şi denotă că specializarea unor actori este relativă față de alţii. De exemplu, relaţia de generalizare de la actorul A la actorul B indică faptul că fiecare exemplar al actorului A este concomitent cu exemplarul actorului B şi are toate proprietăţile lui. În acest caz, actorul B este părinte relativ al actorului A, iar actorul A este descendentul actorului B, de aceea actorul A are posibilitatea de joc a aceleiaşi mulţimi de roluri ca şi actorul B, adică dacă un actor moștenește un alt actor, atunci el poate sa comunice cu aceleași cazuri de utilizare ale sistemului ca şi parintele său. Grafic, relaţia dată este reprezentată prin săgeata de generalizare, adică prin linie continuă cu săgeată în formă de triunghi nehaşurat, care indică părintele actorului (Fig. 4.2.6).

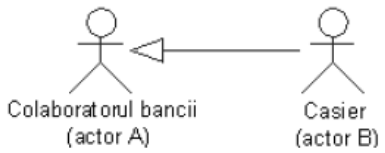



**Fig. 4.2.6.** Relaţia de generalizare dintre actori.

*Relaţia de tip include,* Relaţia de tip *include* între două cazuri de utilizare indică un comportament stabilit pentru un caz de

utilizare ce este inclus drept component compus în consecutivitatea comporatamentului altui caz de utilizare.

Semantica acestei relaţii este definită astfel: când exemplarul primului caz de utilizare în timpul executării ajunge la punctul de includere în consecutivitatea comporatamentului exemplarului al doilea al cazului de utilizare, exemplarul primului caz de utilizare execută consecutivitatea acţiunilor, care definesc comportamentul exemplarului al doilea al cazului de utilizare, după ce continuă executarea acţiunilor comportamentului său.

Cazul de utilizare ce este inclus poate fi independent de cazul de bază. Anume el îi impune ultimului un comportament incapsulat, detaliile de realizare a căruia sunt ascunse şi pot fi liber împărţite între cîte-va cazuri de utilizare incluse.

Relaţia de tip include orientată de la cazul de utilizare A la cazul de utilizare B denotă că fiecare exemplar al cazului A include proprietăţi funcţionale stabilite pentru cazul B. Aceste proprietăţi specializează comportamentul cazului respectiv A în diagrama dată.

Grafic, relaţiile date sunt indicate prin linie punctată cu săgeată (cazul relaţiei de dependenţă), îndreptată de la cazul de utilizare de bază la cazul ce este inclus, iar linia cu săgeată este indicată prin cuvântul-cheie „include”, Fig. 4.2.7).

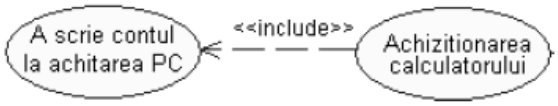



**Fig. 4.2.7.** Relaţie de tip include între cazurile de utilizare

*Exemplu.* Drept exemplu va servi procesul de modelare a sistemului de vindere a mărfurilor după catalog. În calitate de actori ai sistemului dat pot fi două subiecte, unul dintre care este vânzătorul, iar altul cumpărătorul. Fiecare din actori interacţionează cu sistemul de vindere a mărfurilor după catalog şi este utilizatorul lui, adică ambele se adresează la serviciului respectiv „A perfecta comanda de cumpărare a mărfii”. Din cerinţele adresate sistemului, este evident că serviciul reprezintă cazul de utilizare a diagramei,

deoarece strutura de bază poate include numai doi actori şi un singur caz de utilizare (Fig. 4.2.8).

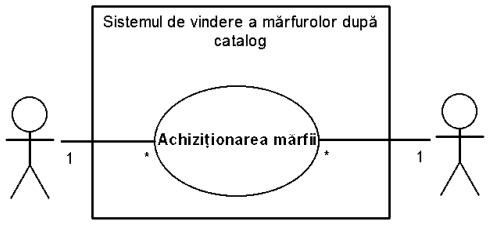



**Fig. 4.2.8.** Diagrama cazurilor de utilizare

Conform cerinţelor, la vinderea mărfurilor, un vânzător poate participa la formarea câtorva comenzi. În acelaşi timp, fiecare comandă poate fi formată numai de un singur vânzător, care este responsabili pentru corectitudinea formării şi respectiv va fi recompensat pentru aceasta. Pe de altă parte, fiecare cumpărător poate forma câteva comenzi şi în acelaşi timp fiecare comandă trebuie să fie formată de un singur cumpărător, care va avea drepturile de proprietate asupra mărfii după achiziţionarea ei.

Etapa următoare în diagrama dată este „A perfecta comanda de cumpărare a mărfii" care poate fi precizată pe baza întroducerii a patru cazuri de utilizare adăugătoare. Acest lucru este evident din analiza mai detaliată a procesului de vindere a mărfii, ce permite a alege ca servicii separate acţiuni ca oferirea cumpărătorului a informaţiei despre marfă, coordonarea condiţiilor de plată şi comandarea mărfii de la depozit. Evident, acţiunile indicate desfăşoară comportamentul cazului de utilizare iniţial, şi anume, concretizarea lui, de aceea între ele va exista relaţia de tip include.

În cazul nostru, catalogul cu mărfuri poate fi comandat de cumpărător sau vânzător când apare necesitatea de a alege marfa şi precizarea detaliilor de vindere. Reprezentarea serviciului „Cererea catalogului cu mărfuri" drept caz de utilizare independent, este mai corect.

Detalizarea poate fi executată pe baza indicării relaţiilor adăugătoare de tipul relaţiei „generalizare” pentru componentele diagramei deja existente. În sistemul de vindere a mărfurilor pot avea valoare independentă şi proprietăţile specifice. O categorie independentă de mărfuri sunt calculatoarele. În acest caz, în diagramă poate fi adăugat un caz de utilizare „Perfectarea comenzii de achiziţionare a calculatorului” cu actorii „Cumpărător de calculatoare” şi „Vânzător de calculatoare”, care sunt legate cu componentele corespunzătoare ale diagramei relaţiei de generalizare (Fig. 4.2.9).

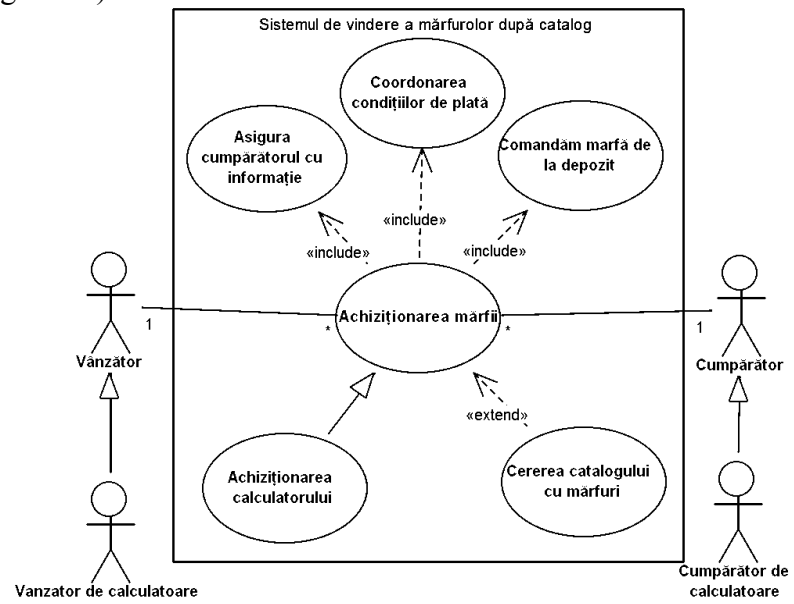



**Fig. 4.2.9.** Diagrama cazurilor de utilizare

Construirea diagramei cazurilor de utilizare este prima etapă în procesul analizei obiect orientate şi proiectare, scopul căreia este reprezentarea totalităţii de cereri la comportamentul sistemului proiectat. Specificaţia de cereri la sistemul proiectat în formă de diagramă a cazurilor de utilizare reprezintă un model independent,

care se numeşte modelul cazurilor de utilizare în limbajul UML şi are un nume propriu stanadard sau steriotip „UseCaseModel”.

**Desfășurarea lucrării:**

1. Să se identifice actorii și relațiile dintre ei.
2. Să se specifice toate scenariile tipice și atipice.
3. Să se determine cazurile de utilizare.
4. Să se stabilească relațiile dintre cazurile de utilizare, cazurile de utilizare și actori.
5. Elaborarea diagramei cazurilor de utilizare.

### 4.3. Lucrarea de laborator nr. 3

**Tema:** Diagrama de clase.

**Scopul:**

1. *Identificarea conceptelor domeniului (analiza domeniului.*
2. *Selectarea claselor și determinarea tipului ei (boundary, entity, controler).*
3. *Identificarea relațiilor dintre clase (dependențe, asociere, generalizare.*
4. *Elaborarea diagramei de clase.*

*Clase, obiecte şi relaţiile lor*

În modelarea orientată spre obiect elementele primare de modelare sunt *clasele*, *obiectele* şi *relaţiile* dintre ele. Când încercăm să descriem un sistem, folosim *clase* şi *obiecte*. Un sistem este format din entităţi utilizând *relaţiile* putem preciza modul în care acestea sunt structurate.

*Clase şi obiecte*

Un *obiect* este o entitate despre care putem vorbi şi pe care o putem gestiona. O *clasă* este o descriere a proprietăţilor şi comportamentului unui tip obiect. Toate obiectele sunt instanţe ale

claselor; un obiect joacă un rol similar cu al unei variabile de un anumit tip într-un limbaj de programare.

*Diagrama de clase.* Pentru a descrie static un sistem vom construi *diagrama claselor,* care constituie punctul de plecare în construirea altor tipuri de diagrame, construite pentru a surprinde alte aspecte ale sistemului, cum ar fi stările obiectelor sau colaborările dintre obiecte.

Modelul de domeniu este creat cu scopul de a reprezenta vocabularul şi conceptele-cheie ale domeniului problemei. Modelul de domeniu identifică, de asemenea, relaţiile dintre toate entităţile în sfera de aplicare a domeniului problemei, precum și atributele lor.

***Identificarea conceptelor domeniului***

*Exemplu*: să ne referim la cazurile de utilizare specificat în figura 4.3.1.

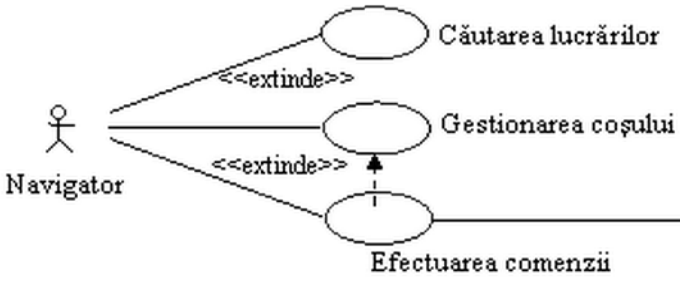



**Fig. 4.3.1.** Diagrama cazurilor de utilizare

*Căutarea lucrărilor*

Pentru acest caz, vom identifica următoarele concepte fundamentale: *lucrarea, autorul, editorul.*

*Gestiunea coşului*

În ”Gestiunea coşului*”* există următoarele concepte fundamentale: *coşul, cartea*.

*Efectuarea comenzii*

"Efectuarea comenzii" are următoarele concepte fundamentale: *comanda, coşul, clientul, cartea de credit*.

***Observaţie***

Modelarea nu tolerează utilizarea mai multor nume pentru același concept. În cazul nostru, "lucrare" este sinonim cu "carte".

**Definiții:**

1. *Clasă* - reprezinta descrierea abstractă a unui ansamblu de obiecte având aceleași caracteristici.

2. *Obiect* - este o entitate cu limite bine definite, având o identitate și înglobând o stare și un comportament. Un obiect este o instanță a unei clase.

3. *Atribut* - reprezintă un tip de informație conținut într-o clasa. *Exemplu*: viteza, numarul de înmatriculare, sunt atribute ale clasei "Autoturism".

4. *Asociere* - este o relație semantică durabilă între două clase. *Exemplu*: o persoană poate avea un autoturism. Asocierea între concepte este implicit bidirecțională (un automobil poate fi posedat de o persoana).

5. *Operație*- reprezinta un element de comportament (un serviciu) conținut într-o clasa.

*Elementele unei clase sunt:*

- ***Numele*** - prin care se distinge de alte clase - o clasă poate fi desenată arătându-i numai numele;
- ***Atributele*** - reprezintă numele unor proprietăţi ale clasei;
- ***Operaţiile (metodele)*** - reprezintă implementarea unor servicii care pot fi cerute oricărui obiect al clasei.

*Notaţia grafică* a clasei poate fi observată în figura 4.3.2.

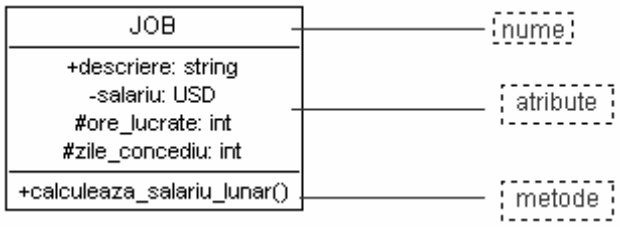


**Fig. 4.3.2.** Notaţia grafică a clasei

*Specificatorii de vizibilitate* au următoarea semnificaţie:

• + public (dreptul de utilizare se acordă şi celorlalte clase);

• - private (numai clasa dată poate folosi atributul sau operaţia);

• # protected (posibilitatea de utilizare este accesibilă numai claselor succesoare).

***Selectarea claselor.*** Selectarea claselor necesită anumite deprinderi ce ţin de filtrarea substantivelor care nu vor reprezenta clase, ci poate doar atribute sau operaţii ale unor clase. Această filtrare presupune eliminarea claselor candidat care au una din următoarele caracteristici:

- *este redundantă*: există deja un sinonim în modelul domeniului;
- *este irelevantă*: substantivul nu este relevant pentru sistemul dat;
- *este un atribut*: încapsulează o singură valoare şi nu interesează în studiul respectiv nici un comportament specific;
- *este o operaţie*: substantivul corespunde unui atribut calculat (ex. Valoarea_totala în clasa Comandă);
- *este un eveniment*: (ex. Generarea rapoartelor are loc lunar).

***Observaţie***

Cea mai frecventa eroare la selectarea claselor este aceia de a reprezenta ceva ca pe un atribut în loc de concept. Dacă nu putem cere unei entităţi decât valoarea sa, atunci acea entitate este un atribut; dacă îi putem pune mai multe întrebări, atunci avem de-a face cu un concept care are la rândul său mai multe atribute şi legături cu alte obiecte.

Există trei tipuri de clase (marcate în UML ca stereotipuri):

• ***Clase entităţi*** (sau clase de domeniu) care reprezintă nucleul unei aplicaţii, reţin informaţile legate de entităţile persistente şi capturează serviciile ce conduc majoritatea interacţiunilor în aplicaţie. De obicei clasele entitate trebuie:

-să înmagazineze şi să redea valori de atribute;

-să creeze şi să şteargă entităţi;

-să furnizeze un comportament dependent de modificarea stării entității.

- ***Clase de interfață*** reprezintă granița dintre actorii care doresc să interacționeze cu aplicația și clasele entitate. Majoritatea sunt componente ale interfeței utilizator (fiecare cutie de dialog este o clasa interfață), modelând comunicarea cu alte aplicații sau reprezentând clase wrapper peste anumite componente soft. Se determină studiind:
  - modul în care actorii doresc să creeze entități;
  - interfața dintre aplicație și alte sisteme;
  - modalitatea de vizualizare a informațiilor (rapoarte).
- ***Clase de control*** (controller) coordonează activitatea în interiorul aplicației. Se ceează câte o clasă controller pentru fiecare caz de utilizare. Pot juca unul din următoarele roluri:
  - modelarea unui comportament tranzacțional;
  - secvență de control specifică unuia sau mai multor cazuri de utilizare;
  - serviciu ce separă obiectele-entitate de obiectele de interfață.

**Identificarea relațiilor dintre clase.** La identificarea claselor se observă că puține din ele au sens singure. Majoritatea claselor colaborează unele cu altele în moduri diferite, astfel încât la modelarea unui sistem, pe lângă identificarea claselor, trebuie identificate și relațiile dintre acestea.

Indentificarea claselor nu poate fi separată, ca moment în timp, de identificarea relațiilor dintre acestea, deoarece înțelegerea relațiilor ajută la rafinarea claselor.

**Relaţii.** O relaţie modelează o conexiune semantică sau o interacţiune între elementele pe care le conectează. În modelarea orientată obiect cele mai importante relaţii sunt relaţiile de asociere, dependenţă, generalizare şi realizare. Un caz particular al relaţiei de asociere îl constituie relaţia de agregare. Între clase se pot stabili relaţii de generalizare, dependenţă şi realizare. Relaţiile de asociere şi agregare se stabilesc între instanţele claselor (Fig. 4.3.3).

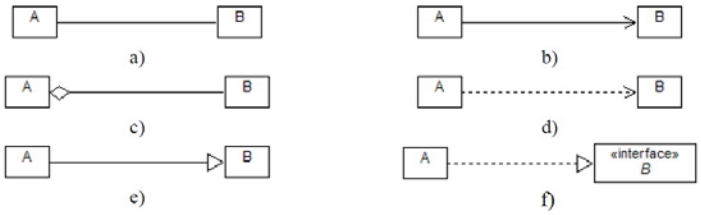



**Fig. 4.3.3.** Notaţii grafice ale diferitelor tipuri de relaţii:
a) asociere bidirecţională; b) asociere unidirecţională;
c) agregare; d) dependenţă; e) generalizare; f) realizare.

***Relaţia de asociere.*** Relaţia de asociere exprimă o conexiune semantică sau o interacţiune între obiecte aparţinând unor anumite clase. Asocierea poartă informaţii despre legăturile dintre obiectele unui sistem. Pe măsură ce sistemul evoluează, pot fi create legături noi între obiecte sau pot fi distruse legăturile existente. Relaţia de asociere oferă baza arhitecturală pentru modelarea conlucrării şi interacţiunii dintre clase.

O asociere interacţionează cu obiectele sale prin intermediul *capetelor de asociere*. Capetele de asociere pot avea *nume*, cunoscute sub denumirea de roluri, şi *vizibilitate*, ce specifică modul în care se poate naviga spre respectivul capăt de asociere. Cea mai importantă caracteristică a lor este multiplicitatea, ce specifică câte instanţe ale unei clase corespund unei singure instanţe ale altei clase. De obicei, multiplicitatea este folosită pentru asociaţiile binare. Reprezentarea grafică a asocierii este o linie ce conectează clasele ce participă la asociere. Numele asocierii este plasat pe linie, iar multiplicitatea şi rolurile sunt plasate la extremităţile sale (Fig. 4.3.4).

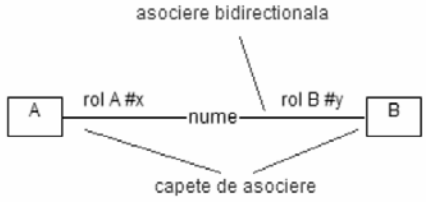



**Fig. 4.3.4.** Notaţia grafică detaliată a relaţiei de asociere

*Observaţie*. Numele rolurilor pot fi omise (eventual şi numele asocierii)

Este posibilă specificarea direcţiei unei asocieri, pentru a elimina legăturile redundante sau irelevante pentru un anumit model. În această situaţie, notaţia grafică pentru relaţia de asociere este o linie cu o săgeată la unul din capete indicând direcţia asocierii (Fig.4.3.3b).

**Relaţia de agregare**

Relaţia de agregare este o asociere ce modelează o relaţie *parte-întreg*. Este reprezentată printr-un romb gol plasat la capătul asocierii de lângă clasa agregat (Fig. 4.3.3c). În figură o instanţă a clasei A conţine o instanţă a clasei B (altfel spus un obiect B este o parte a unui obiect A). Relaţia de agregare este deci un caz particular al relaţiei de asociere. Ea poate avea toate elementele unei relaţii de asociere, însă în general se specifică numai multiplicitatea. Se foloseşte pentru a modela situaţiile în care un obiect este format din mai multe componente.

***Compoziţia*** este o formă mai puternică a agregării. *Partea* are „timpul de viaţă" al *întregului*. *Întregul* poate avea responsabilitatea directă pentru crearea sau distrugerea *părţii* sau poate accepta o altă *parte* creată, apoi să paseze „responsabilitatea" altui *întreg*. În figura 4.3.5 este prezentat un exemplu în care se foloseşte agregarea şi compoziţia.

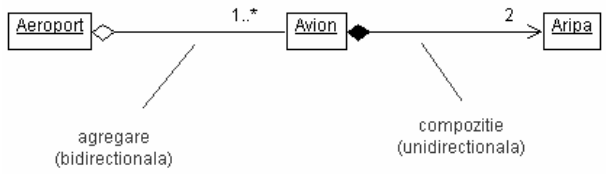



**Fig. 4.3.5.** Exemplu de relaţii de agregare şi compoziţie

**Relaţia de dependenţă.** O dependenţă (Fig. 4.3.3d) indică o relaţie semantică între două elemente ale modelului. Se foloseşte în situaţia în care o schimbare în elementul destinaţie (B) poate atrage după sine o schimbare în elementul sursă (A). Această relaţie modelează interdependenţele ce apar la implementare (vezi ex. din Fig. 4.3.6).

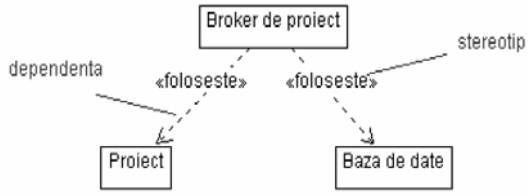



**Fig. 4.3.6** Exemplu de relaţii de dependenţă

**Relaţia de generalizare.** Relaţia de generalizare (Fig. 4.3.3e) se foloseşte pentru a modela conceptul de moştenire între clase. În figura 4.33 e clasa A este derivată din clasa B. Considerăm că A este clasa derivată (subclasa sau clasa copil), iar B este clasa de bază (superclasa, sau clasa părinte). Relaţia de generalizare mai poartă denumirea de relaţie de tip *is a* (*este un fel de*), în sensul că o instanţă a clasei derivate A este în acelaşi timp o instanţă a clasei de bază B (clasa A este un caz particular al clasei B sau, altfel spus, clasa B este o generalizare a clasei A). Clasa A moşteneşte toate atributele şi metodele clasei B. Ea poate adăuga noi atribute sau metode sau le poate redefini pe cele existente (Fig. 4.3.7).

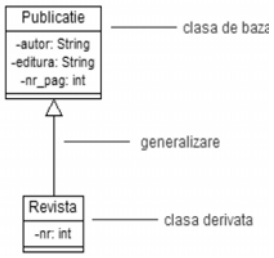



**Fig. 4.3.7.** Exemplu de relaţie de generalizare

**Relaţia de realizare.** Relaţia de realizare (Fig. 4.3.3f) se foloseşte în cazul în care o clasă (A) implementează o interfaţă (B). Se spune că A realizează interfaţa specificată de B. O interfaţă este o specificare comportamentală fără o implementare asociată. O clasă include o implementare. Una sau mai multe clase pot realiza o interfaţă prin implementarea metodelor definite de acea interfaţă (Fig. 4.3.8).

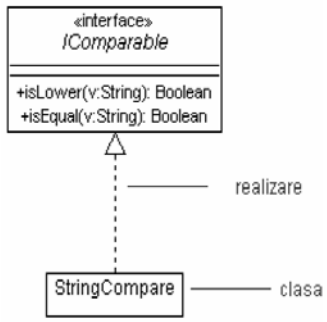
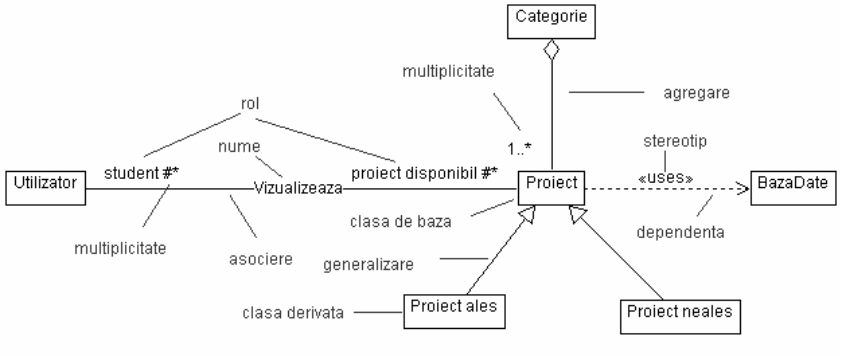



**Fig. 4.3.8.** Exemplu de relaţie de realizare

În figura 4.3.9 sunt prezentate două diagrame de clase în care sunt utilizate tipurile de relaţii descrise anterior.



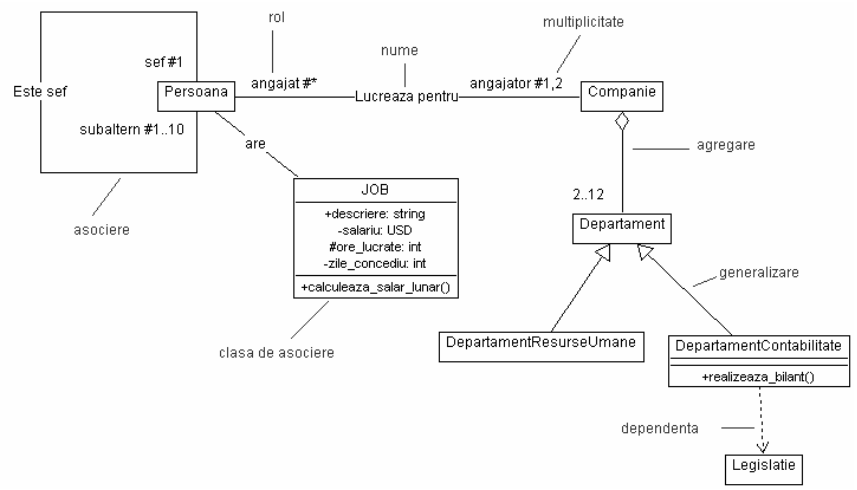



**Fig. 4.3.9.** Diagrama de clase

## Desfășurarea lucrării:

1. Să se elaboreze vocabularul domeniului (analiza domeniului).
2. Să se selecteze clasele și să se determine tipul lor (boundary, entity, controler).
3. Să se determine atributele și metodele claselor.
4. Să se identifice relațiile dintre clase (dependențe, asociere, generalizare) și multiplicitățile lor.
5. Să se elaboreze diagrama de clase.

## 4.4. Lucrarea de laborator nr. 4

**Tema:** Diagrame de interacțiuni (diagrama de secvență/colaborare)

**Scopul:**

1. *Determinarea scenariilor.*
2. *Obiectele ce interacționează.*
3. *Mesajele (sincrone, asincrone).*
4. *Elaborarea diagramei de secvență.*
5. *Utilizarea fragmentlor de reprezentare în diagrama de secvență.*
6. *Elaborarea diagramei de colaborare.*

După înțelegerea domeniului problemei (prin realizarea modelului domeniului) și a cerințelor sistemului (prin realizarea modelului cazurilor de utilizare), se poate trece la etapa de proiectare a sistemului, respectiv realizarea *modelului proiectării*, prin introducerea de concepte specifice programării.

Modelul proiectării descrie atât structura internă a sistemului (clase, obiecte și relații), cât și interacțiunile care intevin când obiectele transmit mesaje pentru a realiza funcționalitatea dorită. Structura statică este descrisă în diagrame de clase, iar dinamica sistemului este descrisă prin diagrame de interacțiune. Aceste grupuri de diagrame au ca elemente comune metodele claselor. Prin urmare, o modalitate de verificare a modelului proiectării este acea de a se asigura corespondența metodelor descrise în diferite diagrame.

Diagramele de interacțiune arată cum conlucrează obiectele prin transmiterea de mesaje pentru a satisface cerințele sistemului. Există două tipuri de diagrame de interacțiune: diagrama de colaborare și diagrama de secvență. Între cele două tipuri de diagrame nu există diferențe majore: diagramele de secvență scot în evidență ordinea în timp a mesajelor, iar diagramele de colaborare scot în evidență legăturile dintre obiecte.

Diagrama de interacțiuni ilustrează interacțiunea obiectelor între ele cu ajutorul *mesajelor*, folosită pentru a modela comportamentul unei mulțimi de *obiecte* dintr-un anumit *context,* care interacționează în vederea îndeplinirii unui anumit *scop*.

*Scop*: specifică modul în care se realizează o operație sau un caz de utilizare.

Diagrama de interacțiuni poate conține:

- obiecte, actori, clase;
- relații;
- mesaje.

*Obiecte:*

- pot fi lucruri concrete sau prototipuri;
- între ele se pot stabili conexiuni semantice (legături);
- comunică între ele prin schimburi de mesaje.

*Mesaj:*

- specifică o comunicare între obiecte;
- îi este asociată o *acțiune* care poate avea ca efect schimbarea stării actuale a obiectului;
- forma generală a unui mesaj:

[cond gardă] acțiune (lista parametrilor)

**Tipuri de acțiuni în UML:**

- call: invocă o operație a unui obiect;
- return: returnează o valoare apelantului;
- send: trimite un semnal;
- create: creează un obiect;
- destroy: distruge un obiect.

Există două tipuri de diagrame de interacțiuni: diagrama de secvență și diagrama de colaborare.

**Diagrama de secvență** pune accentul pe aspectul temporal (ordonarea în timp a mesajelor, figura 4.4.1)

*Notație grafică:*

1. Dimensiunea orizontală (axa ox): obiecte
2. Dimensiunea vertical (axa oy):
   - linia vieții obiectelor (grafic: linie punctată);

- perioada de activare în care un obiect preia controlul execuției (grafic: dreptunghi pe linia vieții)
- mesaje ordonate în timp (grafic: săgeți)

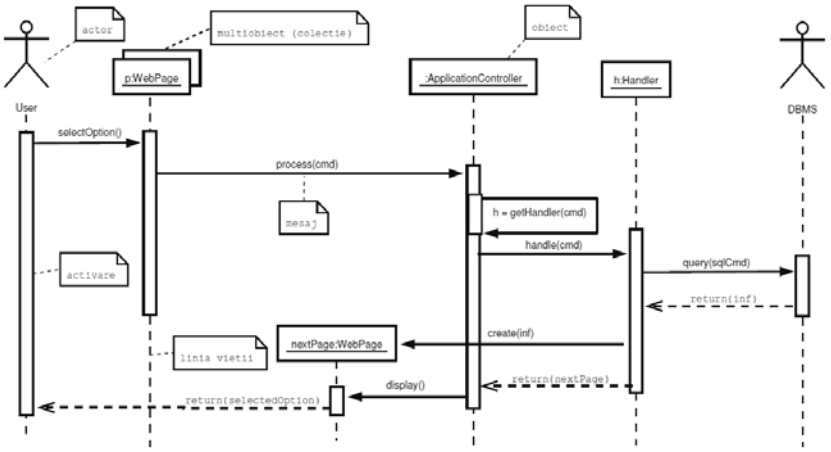



**Fig. 4.4.1.** Diagrama de secvență

**Primitive de comunicare** (Fig. 4.4.2):

*Comunicare sincronă*. Controlul execuției trece de la A la B și revine la A după ce B își termină execuția. *Exemplu*: apel de funcție.

*Comunicare asincronă*. A trimite un semnal după care își continuă execuția mai departe. *Exemplu*: aruncarea unei excepții.

*Return.* Întoarcere dintr-o funcție (procedură). În cazul în care este omisă se consideră implicit că revenirea se face la sfârșitul activării.

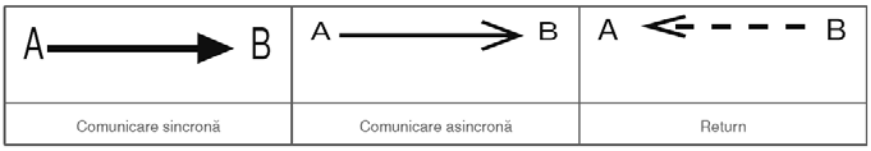



**Fig. 4.4.2.** Pimitive de comunicare

***Ramificații*** – este reprezentarea mai multor mesaje care pleacă din același punct și sunt etichetate cu o condiție (Fig. 4.4.3):

- condiții mutual exclusive → condiționalitate (if, switch);
- condiții care se suprapun → concurență.

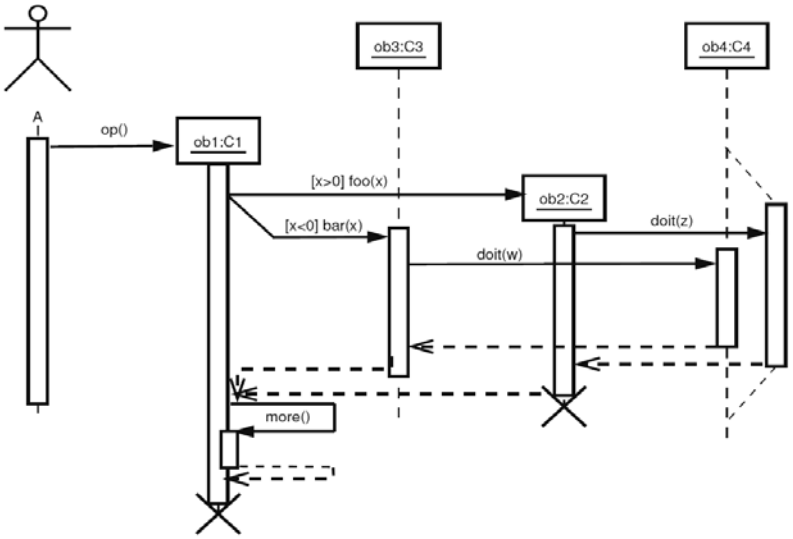



**Fig. 4.4.3** Diagrama de secvență cu ramificații

***Iterații*** (Fig. 4.4.4):

- indică faptul că un mesaj (o mulțime de mesaje) se repetă;
- mesajul este etichetat cu o *condiție-gardă* de forma:

*[cond] acțiune(lista parametrilor);

- dacă sunt mai multe mesaje acestea vor fi înconjurate cu un chenar, în interiorul chenarului va fi specificată condiția (*[cond]).

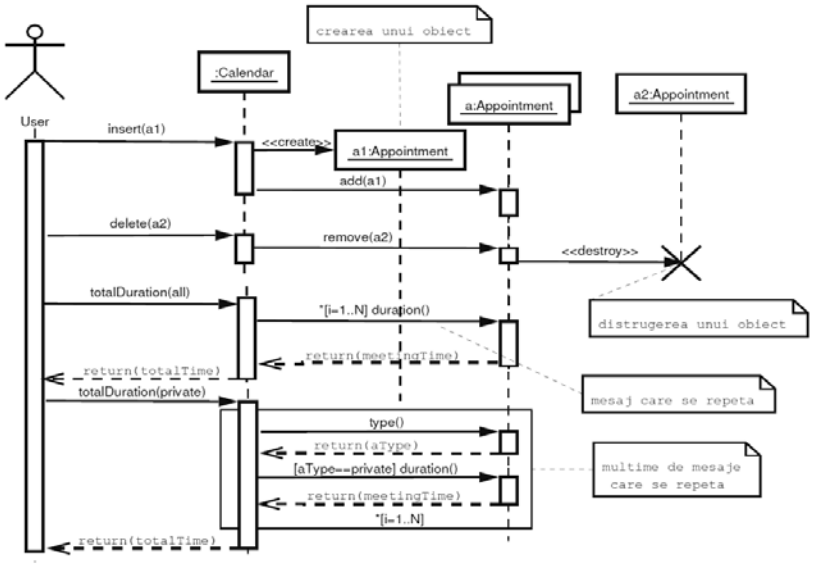



**Fig. 4.4.4.** Diagrama de secvenţă cu iteraţii

**Fragmente combinate** (Fig. 4.4.5):

Fragmentele combinate (combined fragment) sunt elementele modelului, prevăzute pentru reprezentarea structurii logice interne a interacţiunii dintre fragmente.

Operandul de interacţiune (*interaction operand*) este un fragment special al interacţiunii, destinat utilizării ca parte internă a fragmentului combinat.

Constrângerile de interacţiune (*interaction constraint*) este o expresie logică, care acţionează ca o *condiţie-gardă* a unor operanzi într-un fragment combinat.

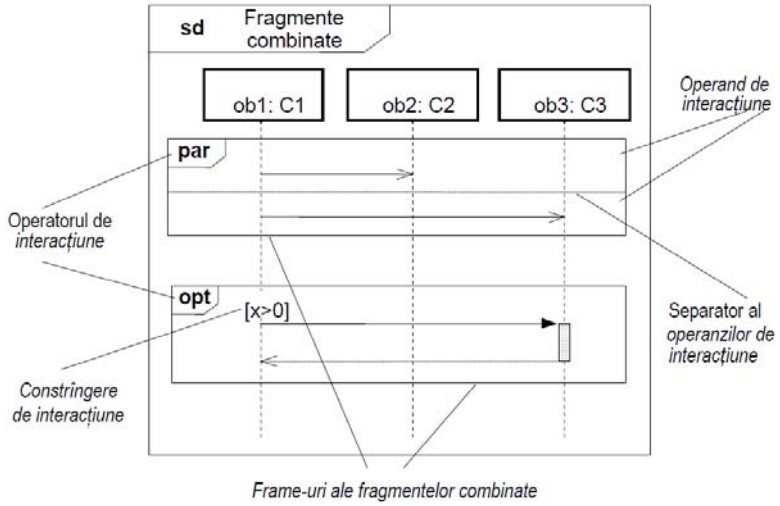



**Fig. 4.4.5.** Diagrama de secvență utilizând fragmente combinate

**Operatorul de interacţiune** (*interaction operator*) determină tipul fragmentului combinat. Se desting 12 operatori:

| | |
|---|---|
| 1) **alt** | 7) **assert** |
| 2) **break** | 8) **critical** |
| **3) ignore** | 9) **consider** |
| 4) **loop** | 10) **neg** |
| **5) opt** | 11) **par** |
| 6) **seq** | 12) **strict** |

Vom examina câțiva dintre ei:

- Operatorul de interacţiune **alt** al fragmentului combinat specifică alternative de interacţiune (alternatives), ce reprezintă o alegere a comportamentului. La alegere participă numai unul dintre operanzi. Operandul selectat trebuie să aibă o condiţie-gardă explicită sau implicită, care la acest punct de interacţiune trebuie să ia valoarea **true**. În cazul în care operandul nu are nici o condiţie-

gardă, implicit se presupune că condiţia de gardă are valoarea **true**. Operandul marcat cu condiţia de gardă [else], denotă negarea disjuncţiei celorlalte condiţii de gardă ale acestui fragment combinat (Fig. 4.4.6).

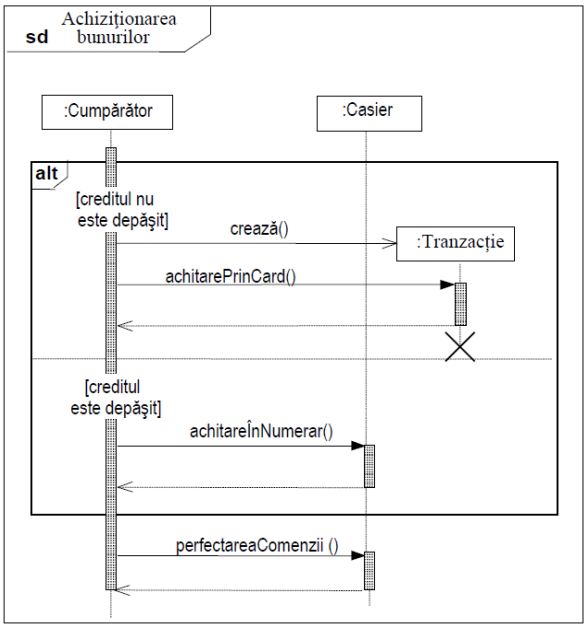



**Fig. 4.4.6.** Diagrama de secvență cu fragment combinat **alt**, (alternative de interacţiune)

Operatorul de interacţiune **break** specifică fragmentul combinat *sfârşit* (**break**), ceea ce reprezintă un scenariu de sfârșit (Fig. 4.4.7). Acest scenariu se execută în locul fragmentului de interacţiune rămas, care conţine acest operand corespunzător. De obicei, operatorul **break** conține o condiţie de gardă. Dacă aceasta condiţie de gardă ia valoarea **true,** atunci se execută fragmentul combinat **break**, iar restul fragmentului de interacţiune ce conţine acest operand este ignorat. În cazul în care condiţia de gardă ia

valoarea **false,** operandul *sfârşit* este ignorat şi se execută restul fragmentului de interacţiune care conţine acest operand.

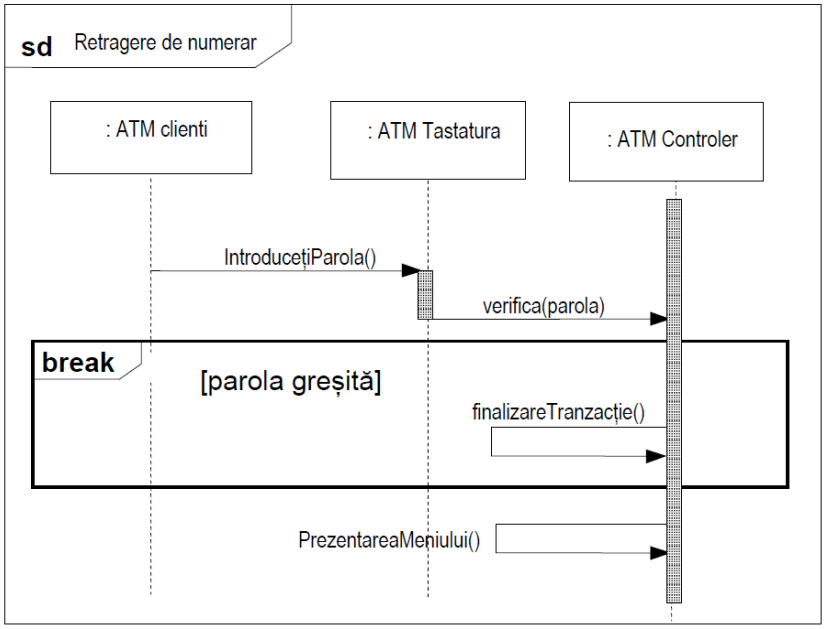



**Fig. 4.4.7.** Diagrama de secvenţă cu fragment combinat **break**

• Operatorul de interacţiune **critical** specifică fragmentul combinat al regiunii critice (critical region), traiectoriile cărora nu pot alterna cu alte specificaţii de evenimente pe acele linii de viaţă, care acoperă această regiune (Fig. 4.4.8). Regiunea critică este tratată ca indivizibilă în determinarea setului de traiectorii posibile ale diagramei sau regiunii pe care aceasta o conţine.

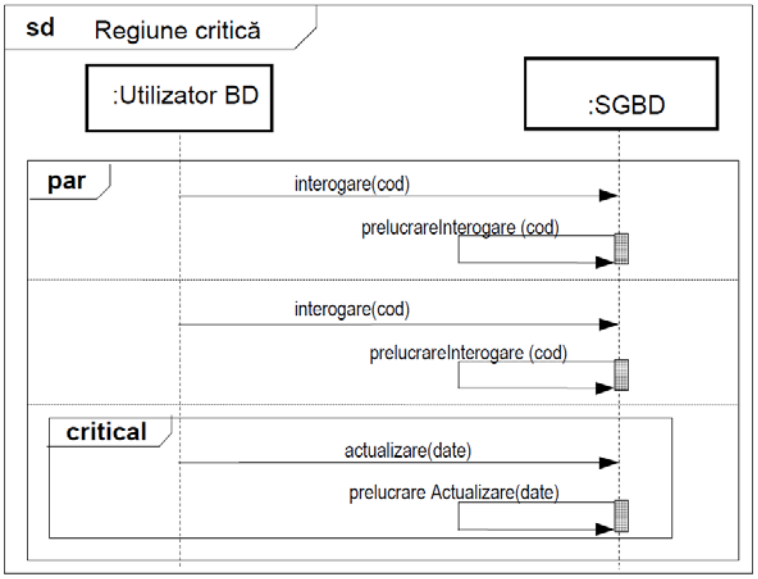



**Fig. 4.4.8.** Diagrama de secvență cu fragment combinat **par** și **critical**

• Setul de traiectorii ale regiunii critice nu poate fi întrerupt de alte evenimente, care au loc în afara acestei regiuni. În practică regiunea critica este utilizată, de obicei, împreună cu operatorul de paralelism **par**.

**Diagrama de colaborare** pune accentul pe organizarea structurală a obiectelor care participă la interacțiune. Aceasta ilustrează mai bine ramificări complexe, iterații și comportament concurent (Fig. 4.4.9).

Poate conține: obiecte, clase, actori; legături între acestea; mesaje.

*Exemple de mesaje*

- **Simple**

*Exemplu:* 2: display(x,y)

- **Subapeluri, inclusiv valoarea de retur**

*Exemplu:* 1.3.1: p=find(specs)

- **Condiționale**

*Exemplu:* 4 [x<0]: invert(x,color)

- **Iterații**

*Exemplu:* 1 *[i=1..n]: update()

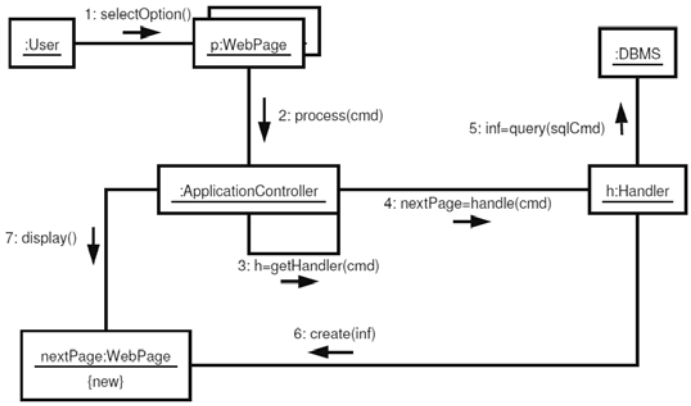



**Fig. 4.4.9.** Exemple de mesaje în diagrama de colaborare

Exemplu diagramă de colaborare (Fig. 4.4.10):

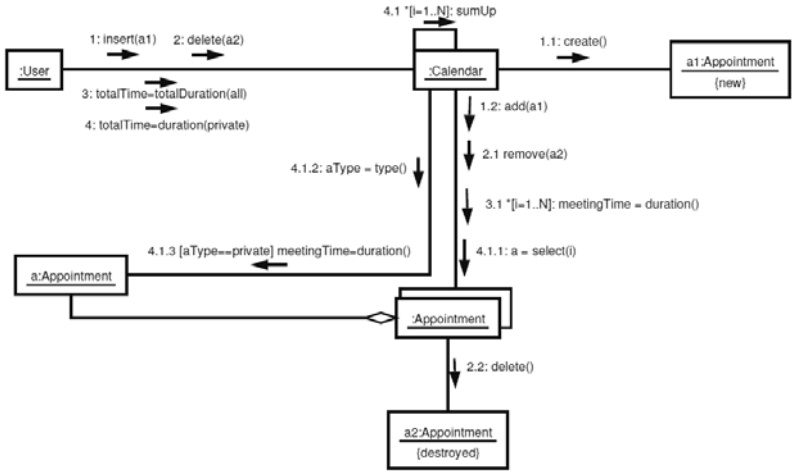



**Fig. 4.4.10.** Diagrama de colaborare cu mesaje condiționate și subapeluri

**Desfășurarea lucrării:**

1. În baza scenariilor elaborate, să se determine obiectele ce participă la interacțiune.
2. Să se specifice comunicarea obiectelor prin intermediul mesajelor.
3. Să se elaboreze diagrama de secvență.
4. Să se utilizeze elemente de control cu fragmente combinate.
5. În baza diagramelor de secvență, să se elaboreze diagrama de colaborare cu specificarea mesajelor dintre obiecte.

## 4.5. Lucrarea de laborator nr. 5

**Tema:** Diagrama de activități și diagrama de stare a obiectelor

**Scopul:**

1. *Descrierea fluxului de activităţi specifice funcţionalităţii sistemului.*
2. *Descrieria stărilor prin care trec unele obiecte.*

Pentru modelarea procesului de executare a operaţiilor, în limbajul UML se utilizează aşa-numitele diagrame de activităţi. Grafic, diagrama de activităţi se reprezintă în forma unui graf de activitate cu noduri – stări activitate şi muchile – tranziţii de la o stare la altă.

Astfel, diagramele de activităţi pot fi considerate cazuri particulare ale diagramelor de stări. Anume aceste diagrame permit realizarea administrării procedurale şi sincrone care depinde de terminarea activităţii interne în limbajul UML. Sensul principal al utilizării acestor diagrame constă în vizualizarea particularităţilor operaţiilor unor clase pentru reprezentarea algoritmilor executării lor. Totodată, fiecare stare realizează operaţiile unei clase anumite şi permite utilizarea diagramei de activităţi pentru descrierea reacţiilor la evenimentele interne ale acestui sistem.

În contextul limbajului UML, activitatea (activity) reprezintă o totalitate de calcule executate automat. Totodată, calculele

elementare pot duce la un anumit rezultat sau la unele acţiuni (action). În diagrama de activităţi se reflectă logica sau consecutivitatea tranziţiilor de la o acţiune la alta, totodată se evidenţiază rezultatul activităţii. Rezultatul, la rândul său, poate duce la schimbarea stării sistemului dat sau la returnarea unei valori.

**Diagrama de activităţi** este folosită pentru a modela dinamica unui proces sau a unei operaţii şi scote în evidenţă controlul execuţiei de la o activitate la alta. Diagramele de activităţi pot conţine:

- stări activităţi şi stări acţiuni care sunt stări ale sistemului;
- tranziţii;
- obiecte;
- bare de sincronizare;
- ramificaţii.

***Stările activitate*** (activity states) - pot fi descompuse, activitatea lor putând fi reprezentată cu ajutorul altor diagrame de activităţi.

Stările activitate nu sunt atomice (pot fi întrerupte de apariţia unui eveniment) şi au durată (îndeplinirea lor se face într-un interval de timp).

***Stările acţiuni*** (action states) - modelează ceva care se întâmplă (de exemplu, evaluarea unei expresii, apelul unei operaţii, a unui obiect, crearea/distrugerea unui obiect).

O stare acţiune reprezintă execuţia unei acţiuni. Ea nu poate fi descompusă. Stările acţiuni sunt atomice (nu pot fi întrerupte chiar dacă se produc evenimente) şi au o durată nesemnificativă (foarte mică).

Notaţia grafică a stărilor activitate/acţiune se poate observa în figura 4.5.1.

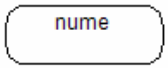
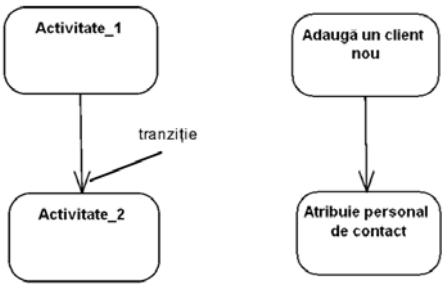



**Fig. 4.5.1.** Notaţia grafică a stărilor activitate/acţiune

***Tranziţiile*** – reprezintă relaţiile dintre două activităţi. Tranziţia este iniţiată de terminarea primei activităţi şi are ca efect preluarea controlului execuţiei de către a doua activitate.




a) b)

**Fig. 4.5.2.** Exemplu de activităţi unite prin tranziţie

În exemplul din figura 4.5.2 b), prima activitate este cea în care se adaugă un client nou. Tranziţia la a doua activitate (şi anume cea de a atribui un personal de contact pentru o eventuală campanie de promovare) implică faptul ca odată ce prima activitate s-a terminat, a doua activitate este startată.

Dacă din starea dată se poate ieşi numai printr-o tranziţie, atunci ea poate să nu fie marcată (indicată), dar dacă tranziţiile de ieşire sunt mai multe, atunci poate să acţioneze numai una din ele. Anume în acest caz, pentru fiecare din tranziţii trebuie să fie indicată în paranteze pătrate o condiţie de supraveghere. Totodată, pentru toate stările de ieşire trebuie să fie executată numai o cerinţă de veridicitate a unei tranziţii. Asta are loc când activitatea consecvent executată trebuie să fie divizată în ramuri alternative independente de valoarea unui rezultat intermediar. Această situaţie se numeşte ramificaţie. Grafic, ramificaţie se reprezintă printr-un romb gol. În acest romb, numai o săgeată de la o acţiune corespunzătoare poate să intre. Săgeata de intrare se uneşte cu vârful de sus sau cu cel din stânga al simbolului de ramificaţie. Pot fi mai multe săgeţi de ieşire, dar pentru fiecare din ele trebuie să fie indicată condiţia de supraveghere sub formă de expresie booleană. Fiecare tranziţie de ieşire trebuie să aibă o condiţie gardă. Condiţiile

gardă trebuie să fie disjuncte (să nu se suprapună) şi să acopere toate posibilităţile de continuare a execuţiei (figura 4.5.3), altfel fluxul de control al execuţiei va fi ambiguu (nu se ştie pe care cale se va continua execuţia). Condiţiile trebuie însă să acopere toate posibilităţile, altfel sistemul se poate bloca.

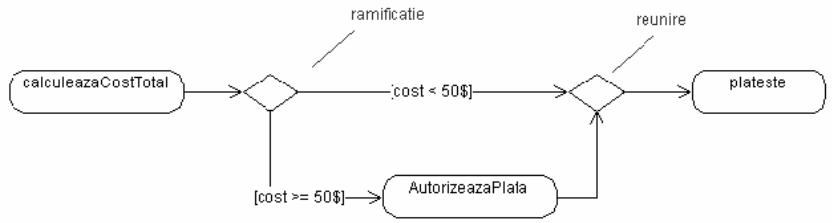
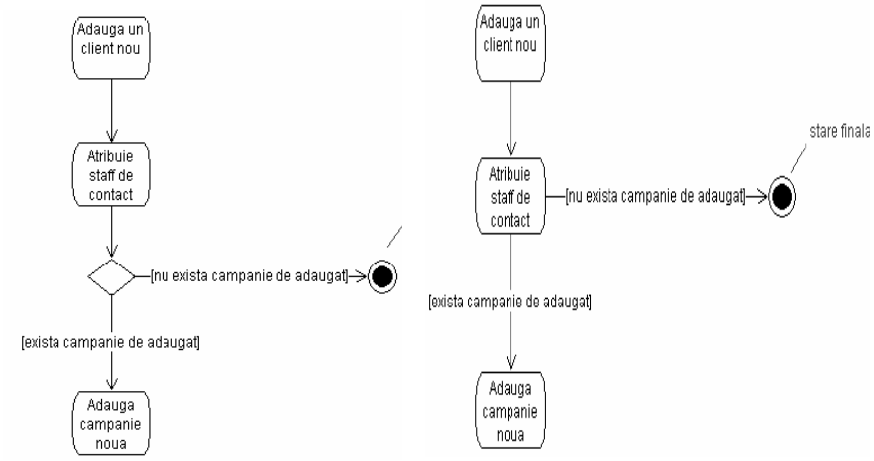



**Fig. 4.5.3.** Activităţi cu puncte de ramificaţie




a) b)

**Fig. 4.5.4.** Activităţi cu punct de ramificaţie și fără punct de ramificaţie explicit

Uneori nu este necesară precizarea explicită a unui punct de decizie, pentru a simplifica diagrama (figura 4.5.4b).

În figura 4.5.4 apare un alt element al diagramelor de activităţi, şi anume, *starea finală*. În general, după încheierea

ultimei activităţi dintr-o diagramă, trebuie marcată tranziţia spre starea finală. De asemenea, după cum se poate observa din figura 4.5.6, fiecare diagramă de activităţi trebuie să înceapă cu *starea iniţială*.

***Bare de sincronizare***

Există posibilitatea ca mai multe activităţi să se execute în paralel. Pentru sincronizarea acestora se folosesc aşa-numitele bare de sincronizare (figura 4.5.5). Acestea pot fi de două feluri:

• ***fork*** - poate avea loc o tranziţie de intrare şi două sau mai multe tranziţii de ieşire, fiecare tranziţie de ieşire reprezentând un flux de control independent. Activităţile de sub *fork* sunt concurente.

• ***join*** - reprezintă sincronizarea a două sau mai multor fluxuri de control. La *join* fiecare flux de control aşteaptă până când toate celelalte fluxuri de intrare ajung în acel punct. Pot avea loc două sau mai multe tranziţii de intrare şi o singură tranziţie de ieşire.

**Fig. 4.5.5.** Notaţia grafică pentru barele de sincronizare

***Obiecte.*** Acţiunile sunt realizate de către obiecte sau operează asupra unor obiecte. Un obiect poate interveni într-o diagramă de activităţi în două moduri:

• o operaţie a unui obiect poate fi folosită drept nume al unei activităţi (Fig. 4.5.6);

• un obiect poate fi considerat intrare sau ieşire a unei activităţi (Fig. 4.5.7).

Obiectele pot fi conectate de acţiuni prin linii punctate cu o săgeată la unul din capete (orientarea săgeţii indică tipul parametrului - intrare sau ieşire). Un obiect poate apărea de mai multe ori în cadrul aceleiaşi diagrame de activităţi. Fiecare apariţie indică un alt punct (stare) în viaţa obiectului. Pentru a distinge

apariţiile, numele stării obiectului poate fi adăugat la sfârşitul numelui obiectului.

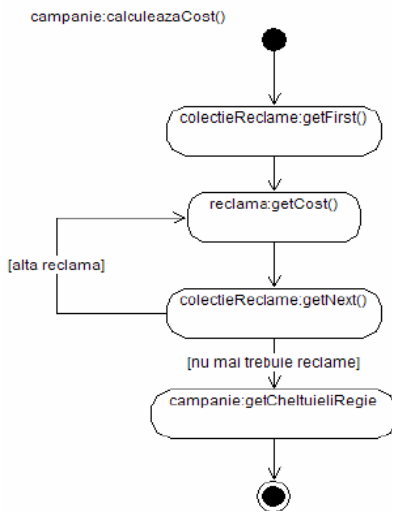



**Fig. 4.5.6.** Diagrama de activităţi cu operaţia obiectului ca activitate

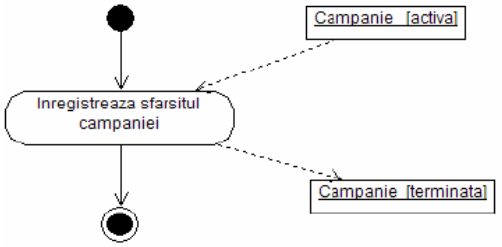



**Fig. 4.5.7.** Diagrama de activităţi cu fluxuri de obiecte

**Conceptul de „swimlanes”** modelează activităţile care au loc în interiorul unui sistem. Denumirele subdiviziunelor sunt indicate în partea de sus a partiţiei. A întretăia linia partiţiei pot numai tranzacţiile, care în acest caz indică ieşirea sau intrarea fluxului de control în subdiviziunea respectivă. Ordinea trecerii partiţiilor nu are importanţă semantică şi este definită după motivele de comfortabilitate.

Drept exemplu vom considera fragmentul diagramei de activitate a companiei de vindere, care servesc clienţii prin telefoni. Subdiviziunele companiei sunt secţia de acceptare şi perfectare a cerinţelor, secţia de vindere şi depozitul. Acestor subdiviziuni vor corespunde trei partiţii în diagrama de activitate, fiecare din care specifică zona de responsabilitate a subdiviziunei.

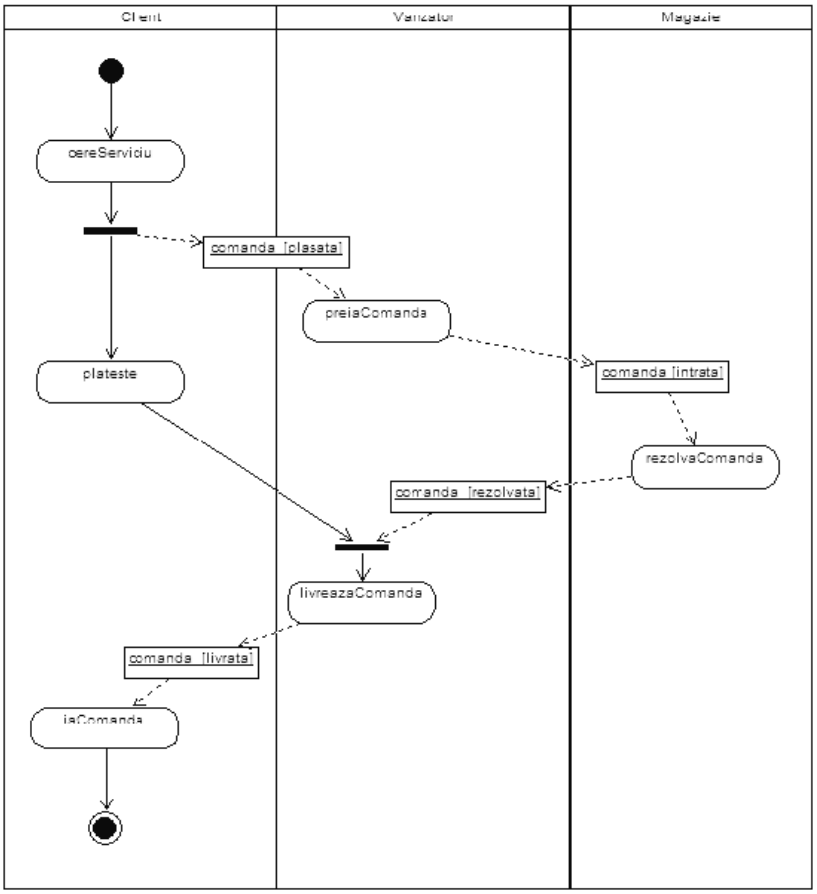



**Fig. 4.5.8.** Diagramă de activităţi cu obiecte şi „swimlanes”

În diagrama de activitate cu partiţii, deplasarea obiectelor poate avea un sens adăugător. Şi anume, dacă obiectul este amplasat la hotarul ambilor partiţii, aceasta înseamnă că trecerea la starea de acţiune următoare în partiţia vecină este asociată cu un document finit (obiectul în careva stare). Dar dacă obiectul este amplasat înăuntrul partiţiei, atunci starea acestui obiect este definită de acţiunile partiţiei date.

## Diagrama de stări (State chart Diagram)

Comportamentul unui program poate fi descris prin următoarele tipuri de diagrame:

- diagrama de stări;
- diagrama de activităţi;
- diagrama de interacţiuni: diagrama de secvenţe, diagrama de colaborare.

**Diagrama de stări** este folosită pentru a modela comportamentul unui singur obiect. Diagrama de stări specifică o secvenţă de stări prin care trece un obiect de-a lungul vieţii sale ca răspuns la evenimente împreună cu răspunsul la aceste evenimente.

## Noţiuni generale

Un ***eveniment*** reprezintă ceva (atomic) ce se întâmplă la un moment dat şi care are ataşată o locaţie în timp şi spaţiu. Evenimentele modelează apariţia unui stimul care poate conduce la efectuarea unei tranziţii între stări. Evenimentele nu au durată în timp.

Evenimentele pot fi clasificate în felul următor:

- sincrone sau asincrone;
- externe sau interne.

Evenimentele *externe* se produc între sistem şi actori (de exemplu, apăsarea unui buton pentru întreruperea execuţiei programului).

Evenimentele *interne* se produc între obiectele ce alcătuiesc un sistem (de exemplu, *overflow exception*).

Evenimentele pot include:

- ***semnale***; semnalul este un stimul asincron care are un nume şi care este trimis de un obiect şi recepţionat de altul (ex: excepţii);
- ***apeluri de operaţii*** (de obicei sincrone): un obiect invocă o operaţie pe un alt obiect; controlul este preluat de obiectul apelat, se efectuează operaţia, obiectul apelat poate trece într-o nouă stare, după care se redă controlul obiectului apelant;
- ***trecerea timpului;***
- ***o schimbare a rezultatului evaluării unei condiţii.***

O ***acţiune*** reprezintă execuţia atomică a unui calcul care are ca efect schimbarea stării sau returnarea unei valori. Acţiunile au o durată mică în timp, fiind tranzitorii (ex.: i++).

Prin ***activitate*** se înţelege execuţia neatomică a unor acţiuni. Activităţile au durată în timp, pot persista pe toată durata stării şi pot fi întrerupte (ex.: execuţia unei funcţii).

O diagramă de stări poate conţine *stări* şi *tranziţii*.

**Starea**

Prin *stare* se înţelege o condiţie sau situaţie din viaţa unui obiect în timpul căreia acesta:

- satisface anumite condiţii;
- efectuează o activitate;
- aşteaptă apariţia unui eveniment.

Notaţia grafică pentru stare este prezentată în figura 4.5.9.

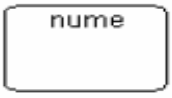



**Fig. 4.5.9.** Notaţia grafică a stării

Există două stări particulare, şi anume, *starea iniţială* şi *starea finală*.

***Starea iniţială*** (Fig. 4.5.10 a) este starea din care pleacă entitatea modelată.

***Starea finală*** (Fig. 4.5.10 b) este starea în care entitatea modelată îşi încheie existenţa.


**Fig. 4.5.10.** Notaţii grafice ale stării iniţiale (a) şi stării finale (b)

Elementele ce caracterizează o stare sunt:

- ***Numele*** - identifică în mod unic o stare; numele este reprezentat de o succesiune de şiruri de caractere.
- ***Acţiunile de intrare/ieşire*** - sunt acţiuni ce se produc la intrarea, respectiv ieşirea din starea respectivă.
- ***Substările*** care pot fi:
  - *concurente* (simultan active) – figura 4.5.11 a;
  - *disjuncte* (secvenţial active) – figura 4.5.11 b.

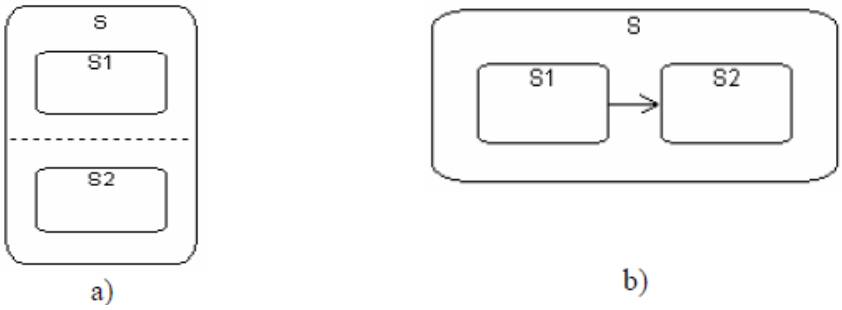



**Fig. 4.5.11.** Notaţii grafice ale substării concurente (a) şi disjuncte (b)

• ***Tranziţii interne*** - sunt acţiuni şi activităţi pe care obiectul le execută cât timp se află în acea stare; se efectuează între substări şi nu produc schimbarea stării obiectului.

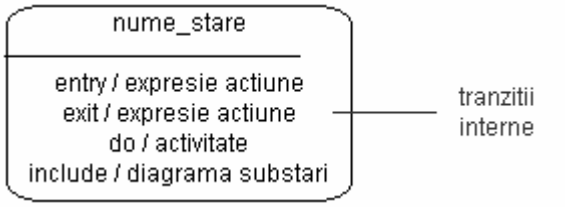



**Fig. 4.5.12.** Notaţia completă a stării

Forma generală a unei tranziţii interne este:

*nume_eveniment(lista_parametri)[condiţie_gardă]/acţiune*

unde:

*nume_eveniment* – identifică circumstanţele în care se execută acţiunea specificată;

*condiţie_gardă* – condiţie booleană care se evaluează la fiecare apariţie a evenimentului specificat; acţiunea se execută doar când rezultatul evaluării este true;

*acţiunea* – poate folosi atribute şi legături care sunt vizibile entităţii modelate.

După cum se poate observa din figura 4.5.12, două evenimente au notaţii speciale: *entry* şi *exit*. Aceste evenimente nu pot avea condiţii gardă, deoarece se invocă implicit la intrarea sau ieşirea din starea respectivă.

Activităţile sunt precedate de cuvântul-cheie *do*.

Pentru a arăta că o stare conţine substări, se foloseşte cuvântul-cheie *include*, urmat de numele diagramei substărilor (Fig. 4.5.13).

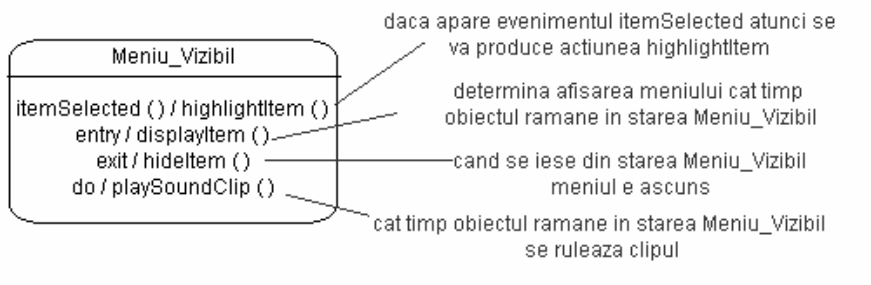



**Fig. 4.5.13.** Exemplu de stare

**Tranziţia.** O *tranziţie* reprezintă o relaţie între două stări, indicând faptul că un obiect aflat în prima stare va efectua nişte acţiuni, apoi va intra în starea a doua atunci când se petrece un anumit eveniment.

Notaţia grafică a tranziţiei se poate observa în figura 4.5.14.

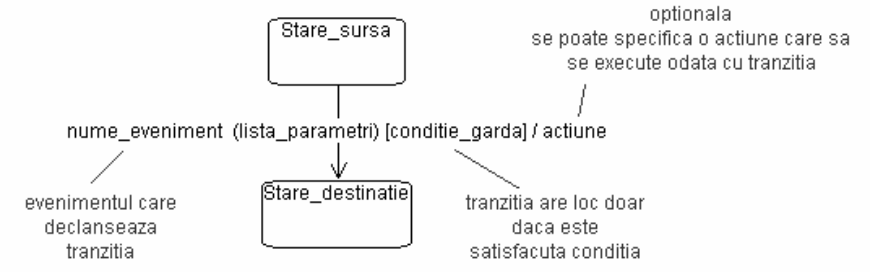



**Fig. 4.5.14.** Notaţia grafică a tranziţiei

*Starea-sursă* reprezintă starea din care se pleacă.

*Eveniment* este evenimentul care declanşează tranziţia.

*Condiţie gardă* (guard condition) este o expresie booleană. Aceasta se evaluează la producerea evenimentului care declanşează tranziţia. Tranziţia poate avea loc numai dacă condiţia este satisfăcută.

*Acţiune* - opţional se poate specifica o acţiune care să se execute odată cu efectuarea tranziţiei.

*Starea-destinaţie* reprezintă starea în care ajunge obiectul după efectuarea tranziţiei.

În figurile 4.5.15 şi 4.5.16 se pot observa exemple de stări cu substări disjuncte, respectiv concurente.

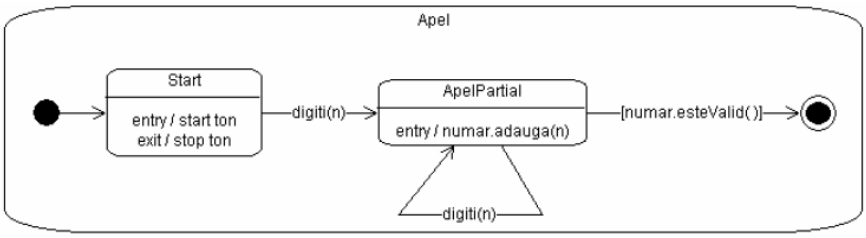



**Fig. 4.5.15.** Exemplu de stare cu substări disjuncte

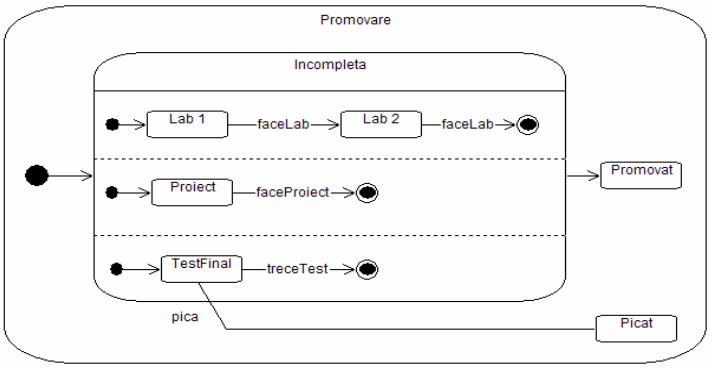



**Fig. 4.5.16.** Exemplu de stare cu substări concurente

În figura 4.5.17 este prezentat un exemplu de diagramă de stare pentru un proiect, propus de un student.

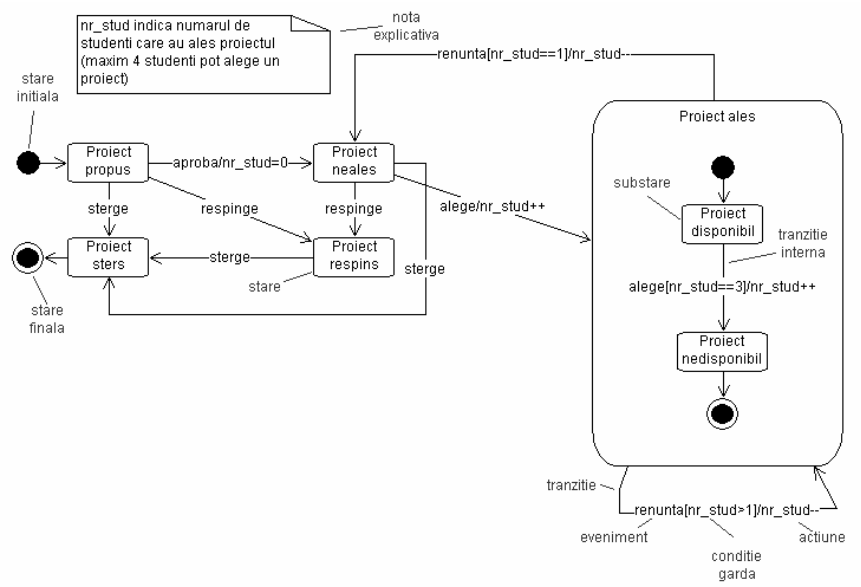



**Fig. 4.5.17.** Diagramă de stare

## Desfășurarea lucrării:

1. În baza scenariilor elaborate, să se descrie fluxul de activități specifice controlului execuţiei de la o activitate la alta.
2. Utilizând conceptul de „swimlanes”, să se modeleze activităţile care au loc în interiorul unui sistem.
3. Să se realizeze diagrama activități.
4. Să se modeleze comportamentul obiectelor prin specificarea secvenţelor de stări prin care trece un obiect de-a lungul vieţii sale.
6. Să se realizeze diagrama de stare.

## 4.6. Lucrarea de laborator nr. 6

**Tema:** Diagrama de componente

**Scopul:**

1. *Determinarea componentelor aplicaţiei.*
2. *Stabilirea dependenţelor dintre componente.*
3. *Determinarea artefactelor*

Toate diagramele analizate mai sus reflectau aspectele conceptuale de proiectare a unui model de sistem şi se referiau la nivelul logic de reprezentare. Specificul reprezentării logice constă în faptul că el utilizează noţiuni care nu au personificare materială proprie. Cu alte cuvinte, nu există în mod material sau fizic elementele reprezentării logice cum ar fi (clase, asocieri, stări, mesaje). Ele numai reflectă înţelegerea noastră despre sistemul fizic sau despre aspectele comportamentului acestui sistem.

Pentru crearea unui sistem fizic real este necesar a transforma toate elementele reprezentării logice în entităţi materiale. Pentru descrierea acestor entităţi este destinat aspectul reprezentării modelare–fizice.

Sistemul program poate fi considerat realizat numai în cazul când va putea executa funcţiile destinaţiei sale. Aceasta este posibil dacă codul-sursă al unui sistem va fi realizat în forma de module executate, biblioteci ale claselor şi procedurilor, interfeţelor grafice standarde, fişierelor bazelor de date. Anume aceste componente sunt necesare pentru reprezentarea fizică a unui sistem.

Proiectul complet al unui sistem al programului reprezintă o totalitate de modele ale reprezentării logice şi fizice care sunt coordonate între ele. În limbajul UML, pentru reprezentarea fizică a unui model al sistemului sunt utilizate diagramele de realizare (implementation diagrams) care includ două diagrame canonice: diagrama *de componente şi diagrama de desfăşurare*.

Pentru reprezentarea entităţilor fizice, în limbajul UML se utilizează componenta (component). Componenta realizează un set de interfeţe şi desemnează elementele reprezentării fizice ale unui model. Grafic, componenta se reprezintă printr-un dreptunghi cu anexe (fig. 4.6.1). În interiorul acestui dreptunghi se indică numele componentei şi posibil informaţia suplementară. Reprezentarea acestui simbol variază în dependenţă de informaţia asociată cu componenta dată.

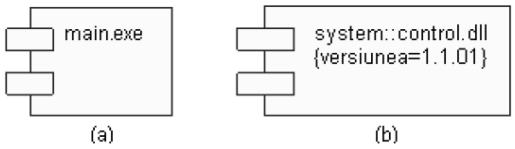



**Fig. 4.6.1.** Componentă

Următorul element al diagramei componentelor sunt interfeţele. În caz general, interfaţa este reprezentată în formă de circumferinţă, care este legată de component prin linia fără săgeată (Fig. 4.6.2, a), iar numele interfeţei, care obligatoriu trebuie să fie scrisă cu majusculă ”I”, este notată alături de circumferinţă. Semantic, linia înseamnă interfaţa, iar prezenţa interfeţelor în cadrul componentelor înseamnă că componentul dat realizează trusă de interfeţe respective.

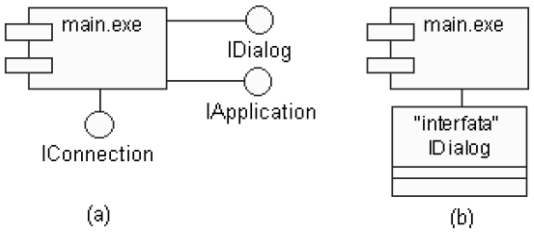



**Fig. 4.6.2.** Reprezentarea grafică a interfeţelor în diagrama de componente

Un alt mod de reprezentare a interfeţelor în diagrama de componente este reprezentarea în forma de dreptunghi a clasei cu stereotipul «interfaţa» şi cu secţiuni posibile ale atributelor şi operaţiilor (Fig. 4.6.2, b). De regulă, acest caz de notare este utilizat pentru reprezentarea structurii interne a interfeţei, care poate fi importantă pentru realizare.

Există două feluri de legături ale interfeţei şi componentului. Dacă componentul realizează o anumită interfaţă, atunci această interfaţă este numită de export (provided), deoarece acest component prezintă în el modul de serviciu pentru altel componente. Dacă componentul utilizează o anumită interfaţă, care este realizată de un alt component, atunci acea interfaţă pentru primul component este numită de import (required). Particularităţile interfeţei de import constau în faptul că în diagrama de componente această relaţie este reprezentată cu ajutorul dependenţei.

Relaţia de dependenţă în diagrama de componente reprezintă o linie întreruptă cu săgeată orientată de la client (element dependent) spre sursă (element independent).

Relaţia poate indica legăturile modulelor programului la etapa de compilare şi generare a codului. În alt caz, dependenţa poate indica existenţa în componentul independent a descrierii clasei, care sunt utilizate în componentul dependent pentru crearea obiectelor respective. În diagrama de componente dependenţele pot conecta componentele şi interfeţele de import de component, dar şi diferite tipuri de componente între sine.

În primul caz se deseanează săgeata de la component–client la interfaţa de import (Fig. 4.6.3).

Prezenţa săgeţii înseamnă că componetul nu realizează interfaţa respectivă, dar utilizează procesul său de executare. În această diagramă mai poate exista şi un alt component care realizează această interfaţă. De exepmplu, fragmentul diagramei de componente prezentat mai jos reprezintă o informaţie despre componentul cu numele «main.exe» dependent de interfaţa de import IDialog, care la rândul său este realizată de componentul cu

numele «image.java». Pentru al doilea component aceeaşi interfaţă este de export.

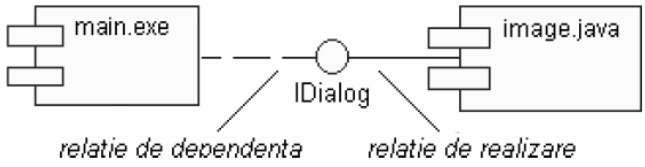



**Fig. 4.6.3.** Fragmentul diagramei de componente cu relaţie de dependenţa.

Dacă unele componente realizează anumite clase, atunci pentru indicarea componentului este utilizat simbolul în formă de dreptunghi, dreptunghiul componentului divizându-se în două secţiuni prin linia orizontală. Secţia de sus este utilizată pentru notarea numelui componentului, iar cea de jos – pentru indicarea informaţiei adăugătoare (Fig. 4.6.4).

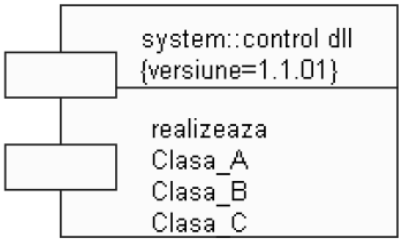



**Fig. 4.6.4.** Reprezentarea grafică a componentului cu informaţie adăugătoare despre clasele realizate

În cadrul simbolului componentului sunt indicate alte elemente ale notaţiei grafice, cum ar fi componentele nivelului de tip sau componentele nivelului de exemplare (obiectele). În acest caz simbolul componentului este reprezentat în aşa fel ca să introducă aceste simboluri adăugătoare. De exemplu, componentul realizat mai jos (Fig. 4.6.5 ) este exemplar şi realizează trei obiecte.

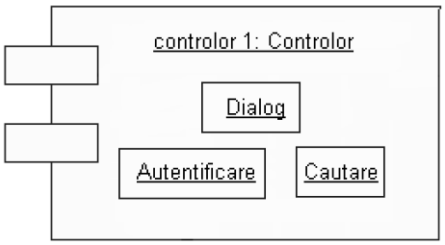



**Fig. 4.6.5.** Reprezentarea grafică a componentului nivelului de exemplar ce realizează obiectele.

Obiectele, care se află în cadrul componentului–exemplar sunt reprezentate într-un fel de elementele depuse în simbolul componentului dat. Astfel de depunere înseamnă că efectuarea componentului duce la executarea obiectelor respective.

Instrumentul Rational Rose introduce o serie de stereotipuri predefinite pentru componente (Fig. 4.6.6 ):

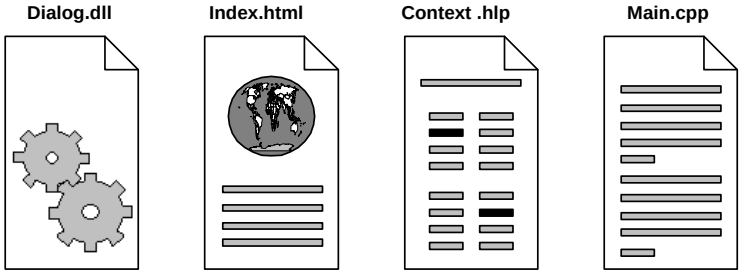



**Fig. 4.6.6.** Variantele reprezentării grafice a componentelor

Aceste elemente sunt uneori numite artefacte, subliniind astfel conţinutul lor informaţional finit, dependent de tehnologia de realizare concretă a componentelor respective. Mai mult decât atât, elaboratorii pot utiliza un acest scop notaţii independente, deoarece în limbajul UML nu există notare strictă pentru reprezentarea grafică a notaţiilor.

*Recomandări privind construirea diagramei de componente*

Elaborarea diagramei de componente presupune utilizarea informaţiei despre reprezentarea logică a modelului sistemului, dar şi despre particularităţile realizării ei fizice. Înainte de elaborare este necesar a lua o decizie privind alegerea programei de calcul şi a sistemului operaţional, după care să realizăm sistemul, dar şi alegerea bazelor de date concrete şi limbajelor de programare.

Astfel ajungem la structurizarea generală a diagramei de componente. În primul rând, este necesar a hotărî din care părţi (fişiere) fizice va fi compus sistemul de programare.

Etapa finală de construire a diagramei de componente este legată de stabilirea şi depunerea în diagramă a înteracţiunilor dintre componente şi relaţiile de realizare. Aceste relaţii trebuie desfăşurate în toate aspectele importante ale realizării fizice a sistemului, începând cu particularităţile compilării textelor iniţiale ale programului şi terminând cu îndeplinirea unor părţi ale programului la etapa de executare. Pentru aceast scop pot fi utilizate diferite tipuri de reprezentare grafică a componentelor. Dar dacă proiectul conţine anumite elemente fizice, descrierea cărora lipseşte în limbajul UML, trebuie utilizat mecanismul de extindere. Şi anume, utilizarea stereotipurilor adăugătoare pentru unele componente netipice sau valorile marcate pentru precizarea unor caracteristici ale lor.

În sfârşit, trebuie atrasă atenţia că diagrama de componente este, de regulă, elaborată împreună cu diagrama de plasare, care reprezintă informaţia despre deplasarea fizică a componentelor sistemului programei după nodurile ei. Particularităţile construirii diagramei de plasare vor fi definite în capitolul următor.

Etapa de implementare a sistemului presupune un efort de modelare prin realizarea *modelului componentelor.* Modulele de cod de diferite tipuri și fișierele binare pot fi descrise prin prisma claselor conținute și a dependențelor dintre ele în *diagramele componentelor.*

Etapa de instalare a sistemului presupune realizarea modelului de desfășurare, prin prezentarea nodurilor folosite, a modurilor în

care acestea sunt conectate, precum și a componentelor ce se vor executa pe fiecare nod (de exemplu, ce program sau serviciu este executat pe fiecare calculator). Acest model a1 sistemului prezintă interes pentru dezvoltatori și pentru cei care realizeaza testarea sistemului, fiind realizată cu ajutorul diagramelor de desfășurare.

*Diagramele de componente* sunt diagrame ce descriu arhitectura codului, prin prezentarea componentelor unei aplicații și a dependențelor dintre acestea (Fig. 4.6.7). Termenul de componentă reda orice unitate de program (cod-sursă, fișier executabil, bibliotecă de funcții, machete realizate în generatoare de rapoarte) sau documente (Word, Excel), care îndeplinesc funcții în cadrul sistemului informatic.

Diagramele de componenete sunt importante, deoarece:

- modelează sistemul software real în mediul de implementare;
- evidențiază probleme de configurare prin relațiile de dependență;
- reprezintă o imagine a sistemului existent, înainte de a fi modificat;
- pot evidenția probleme de implementare fără a fi necesar să se citească tot codul-sursa.

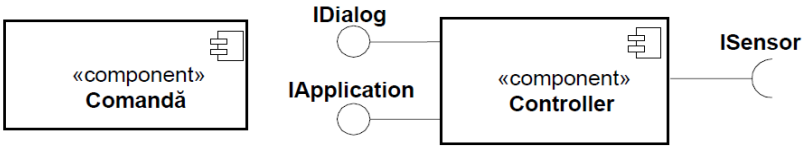



**Fig. 4.6.7.** Diagramele de componente

Există două tipuri de interfețe:

- *Provided interfaces*: aceste interfeţe descru servicii pe care instanţele unui clasificator (furnizor) le oferă clienţilor.

- *Required interfaces*: aceste interfețe specifică serviciile de care au nevoie un clasificator pentru a-și îndeplini funcţiile şi obligaţiile sale față de clienţi.

Aceste relaţii de dependenţă arată cum sunt îndeplinite cerinţele funcţionale ale componentelor de către alte componente, precum şi modul în care sistemul în ansamblu este capabil să execute lucrările necesare.

Dependențele dintre componente se reprezinta grafic prin linii întrerupte între o componentă-client şi o componentă-furnizor de servicii, orientate spre componenta-furnizor. Relatia de dependeță semnifică faptul ca clasele incluse în componenta-client pot moşteni, instanția sau utiliza clase incluse în componenta-furnizor (sau server).

De asemenea, pot exista relații de dependență între componente şi interfețe ale altor componente, relații care semnifică faptul ca clientul utilizează operații ale componentei server.

**Desfășurarea lucrării:**

1. Să se descrie toate componentele şi interfețele aplicației.
2. Să se specifice legăturile dintre interfaţă şi component.
3. Să se specifice artefactele sistemului.
4. Să se realizeze diagrama de componente.

## 4.7. Lucrarea de laborator nr. 7

**Tema:** Diagrama de desfășurare.

**Scopul:**

1. *Determinarea nodurilor.*
2. *Stabilirea relațiilor dintre noduri.*
3. *Distribuirea componentelor în noduri.*

Diagrama de desfăşurare descrie structura sistemului în momentul execuţiei. Ea prezintă dispunerea fizică a diferitelor elemente hardware, numite noduri, care intră în componenţa unui sistem, şi repartizarea programelor executabile pe aceste elemente.

În diagrama de desfăşurare, se indică nodurile şi conexiunile unui model.

Un nod este resursa care este disponibilă în timpul execuţiei unui sistem software şi reprezintă un procesor sau un dispozitiv, pe care vor fi desfăşurate şi executate componentele sistemului.

Reprezentarea fizică a sistemului nu poate fi deplină, dacă lipseşte informaţia despre programele şi metodele de realizare a calculului. Dacă este elaborat un simplu program, care poate fi executat local la calculatorul utilizatorului fără introducearea echipamentelor periferice şi a resurselor, atunci nu este necesară elaborarea diagramelor adăugătoare.

Însă sistemele de programare compuse pot fi realizate în formă de reţea în diferite programe de calcul şi tehnologii de accesare la bazele de date. Prezenţa reţelei locale corporative necesită rezolvarea totalităţii de probleme adăugătoare despre amplasarea raţională a componentelor după nodurile reţelei, ce definesc producerea generală a sistemului.

Integrarea sistemului (aplicaţie) în Internet, necesită asigurarea securităţii, şi accesul stabil la informaţie pentru clienţii corporativi. Aceste aspecte depind mult de realizarea proiectului în formă de noduri a sistemului, care există fizic, precum serverele, canalele de conectare şi păstrările de date.

Diagrama de componente este prima diagrama a reprezentării fizice. A doua formă de reprezentarea fizică a sistemului este diagrama de desfăşurare. Ea este utilizată pentru reprezentarea generală a configurării şi topologiei sistemului şi conţine repartizarea componentelor după nodurile sistemului.

Scopurile urmărite în timpul elaborării diagramei de desfăşurare sunt:

- a definit distribuirea componentelor sistemului după nodurile fizice;
- a prezenta legăturile fizice între toate nodurile de realizare a sistemului la etapa de executare;

- a găsi locurile înguste ale sistemului şi a reconfigura topologia acestuia pentru atingerea productivităţii necesare;
- pentru garantarea cerinţelor, diagramei de desfăşurare se elaborează împreună cu analiştii sistemului, inginerii de reţea şi alţii.

Nodul (node) reprezintă un element fizic al sitemului, care are o anumită resursă.

Reprezentaea grafică a nodului în diagrama de desfăşurare este în formă de cub. Nodul are un nume propriu care este indicat în interiorul acestui simbol grafic. Nodurile pot fi prezentate în formă de tipuri (fig. 4.7.1 a), dar şi în calitate de exemplare (fig. 4.7.1, b).

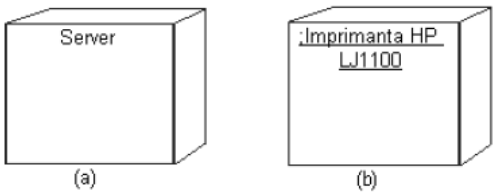



**Fig. 4.7.1.** Reprezentarea grafică a nodului în diagrama de desfăşurare

În primul caz, numele nodului este scris fără subliniere şi începe cu majusculă. În al doilea caz, numele nodului–exemplar este scris în formă de <numele nodului ':' numele tipului nodului>. În desenul din figura 4.7.1, a, nodul cu numele ”Server” se referă la tipul general şi nu se concretizează. Al doilea nod (Fig. 4.7.1, b) este nodul – exemplar anonim al modelului concret al imprimantei.

La fel ca şi în diagrama de componente, reprezentarea nodurilor poate fi extinsă pentru includerea informaţiei adăugătoare despre specificarea nodului. Dacă informaţia adăugătoare este legată de numele nodului, atunci ea este scrisă mai jos de nume ca valoare marcată (Fig. 4.7.2).

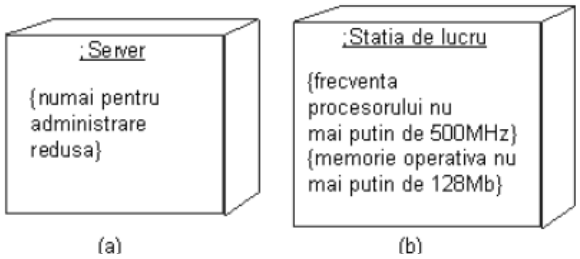



**Fig. 4.7.2.** Reprezentarea grafică a nodului–exemplar cu informaţie adăugătoare ca valoare marcată

Dacă este necesară indicarea componentelor care sunt plasate în noduri separate, aceasta se face în două moduri. Primul dă posibilitatea de a împărţi simbolul grafic în două secţii cu linie orizontală. În secţiunea de sus este scris numele nodului, iar în cea de jos sunt plasate componentele nodului dat (Fig. 4.7.3 a).

Al doilea mod permite a plasa în diagrama de desfăşurare nodurile cu componentele depuse (Fig. 4.7.3b). Drept componente depuse pot fi numai componentele executabile.

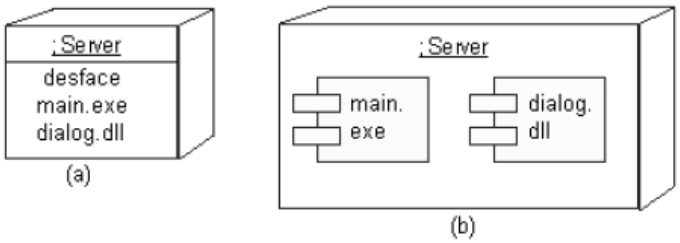



**Fig. 4.7.3.** Reprezentarea grafică a nodurilor–exemplare cu componentele deplasate

Drept adăugare la numele nodului pot fi utilizate diferite stereotipuri care specifică destinaţia nodului. Deşi în limbajul UML stereotipurile pentru noduri nu sunt specificate, în literatură există următoarele variante: ”procesor”, ”modem”, ”reţea” şi altele, care pot fi specificate de elaborator. Mai mult decât atât, în diagramele

de desfăşurare pot fi indicaţii speciale pentru diferite echipamente fizice, iar reprezentarea grafică clarifică destinaţia sau funcţiile lor.

Pe lângă reprezentarea nodurilor, în diagrama de desfăşurare sunt indicate şi relaţiile dintre ele. În calitate de relaţii servesc conectările fizice dintre noduri şi dependenţa dintre noduri şi componente.

Conectările sunt un fel de asociere, fiind prezentate în formă de linii fără săgeţi. Prezenţa liniei indică necesitatea organizării canalului fizic pentru schimbarea informaţiei dintre nodurile respective. Caracterul de conectare poate fi adăugător specificat de adnotare, indicată de valoare sau restricţie. În fragmentul prezentat mai jos, în diagrama de desfăşurare (Fig. 4.7.4) sunt specificate nu numai cererea la viteza de transfer a datelor în reţeaua locală cu ajutorul valorii indicate, dar şi recomandarea privind tehnologia fizică a realizării conexelor în formă de adnotare.

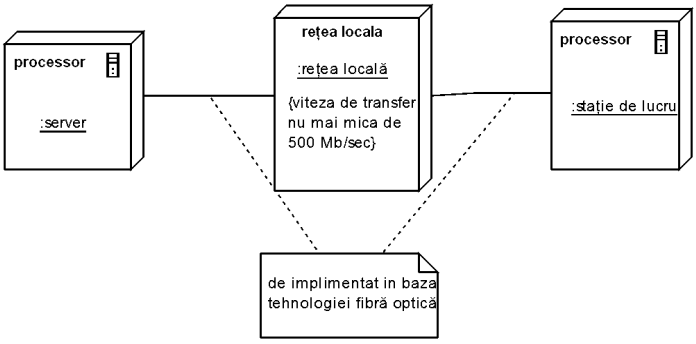



**Fig. 4.7.4** Diagrama de desfăşurare cu adnotare

În afară de conectări, în diagrama de desfăşurare pot exista relaţii de dependenţă între nod şi componentele lui. Acest caz este alternativ pentru reprezentarea componentelor introdu-se în simbolul nodului, ce nu este întodeauna comod, deoarece simbolul devine valoros. De aceea când există mulţimea de componente desfăşurate în nod, o informaţie respectivă poate fi reprezentată în fel de relaţie de dependenţă (Fig. 4.7.5).

Diagrama de desfăşurare poate avea o structură mai compusă, care include componentele, interfeţele şi alte echipamente. În diagrama de desfăşurare din figura 4.7.6 putem observa fragmentul reprezentării fizice a sistemului serviciului destinat clienţilor băncii. Nodurile acestui sistem sunt terminalul depărtat (nodul-tip) şi serverul băncii (nodul-exemplar).

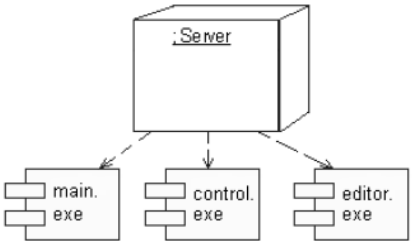



**Fig. 4.7.5.** Diagrama de desfăşurare cu relaţia de dependenţă dintre nod şi componente

În această diagramă de desfăşurare este indicată dependenţa componentului de realizare a dialogului ”dialog.exe” în terminalul depărtat de interfaţa IAuthorise, realizat de componentele ”main.exe”, care la rândul său se desfăşoară la nodul–exemplar anonim ”Serverul băncii”. Ultimul depinde de componentul bazei de date ”Clienţii băncii” care, de asemenea, este desfăşurat la acest nod.

Adnotarea indică necesitatea utilizării liniei protejate de comunicare pentru schimbarea datelor în sistemul dat.

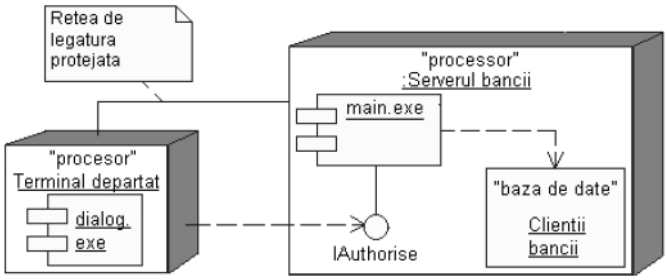



**Fig. 4.7.6.** Diagrama de desfăşurare pentru sistemul serviciului destinat clienţilor băncii

Elaborarea diagramei de desfăşurare începe cu identificarea tuturor tipurilor de echipamente, care sunt necesare pentru executarea de către sistem a funcţiilor sale. În primul rând, sunt specificate nodurile sistemului ce au memorie şi/sau procesorul, în urma căruia sunt utilizate steriotipuri ale limbajului UML, iar în cazul lipsei lor, elaboratorii pot specifica stereotipuri noi. Etapa următoare constă în plasarea tuturor componentelor executabile ale diagramei în nodurile sistemului.

De regulă, elaborarea diagramei de desfăşurare este realizată la etapa finală, ceea ce caracterizează sfârşitul fazei de proiectare a reprezentării fizice. Pe de altă parte, diagrama de desfăşurare poate fi construită pentru analiza sistemului respectiv cu scopul analizei şi modificării ulterioare, în urma căreia analiza presupune elaborarea acestei diagrame la etapa ei iniţială, ceea ce caracterizează direcţia generală a analizei de la reprezentarea fizică la cea logică.

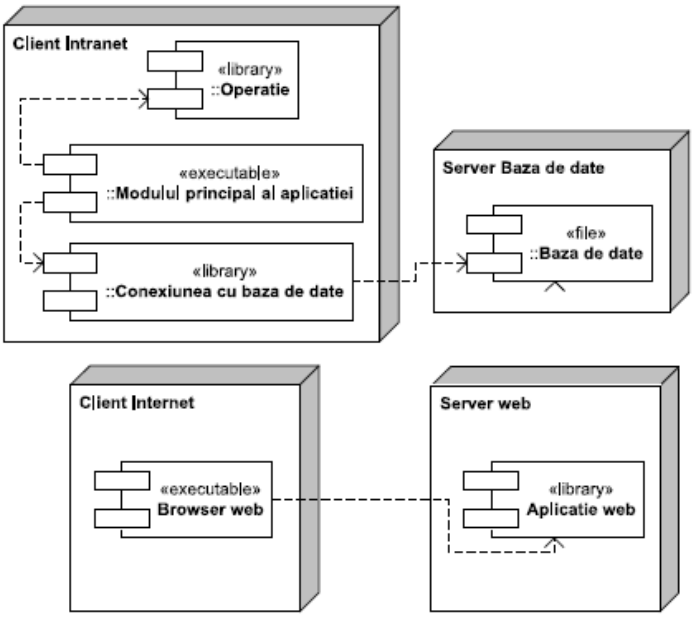



**Fig. 4.7.7.** Diagrama de desfăşurare

Diagrama de desfășurare din figura 4.7.7, simbolizează existenta, în cadrul unei aplicații-client, a unui modul principal, ce se conecteaza la o baza de date aflată pe un alt calculator, prin intermediul unei biblioteci specifice, pentru a răspunde cererilor formulate de anumite biblioteci ce implementează diverse operații. Conform aceleiași diagrame, la acest server de baze de date se conectează o aplicație web, care poate fi apelată de utilizatori doar prin intermediul unui browser.

**Desfășurarea lucrării:**

1. Să se determine elementele fizice (hardware) care intră în componenţa unui sistem.
2. Să se repartizeze componentele pe echipamentele hardware (noduri).
3. Să se determine relațiile dintre noduri.
4. Să se realizeze diagrama de desfășurare.

**Bibliografie**


1. http://www.uml.org/
2. http://inf.ucv.ro/~giurca/courses/CB3105/resources/
   Introducere in UML.pdf
3. http://www.itzone.ro/articolDisplay.php?id=38&categorie_id=0
4. http://www.scribd.com/fullscreen/47113906
5. http://www.racai.ro/Referate/referat_2_Bizoi_web.pdf
6. http://discipline.elcom.pub.ro/isc/Curs_ISC_2011_2_v01.pdf
7. http://www.bazededate.org/ModelulUnificat.pdf
8. http://software.ucv.ro/~eganea/OOP_index.html
9. https://sites.google.com/site/codrutaistin/laborator-fis-5
10. http://www.scritube.com/stiinta/informatica/INGINERIA-PROGRAMARII-_Instant-10423211617.php
11. http://oop.xhost.ro/ip/cursuri.html
12. http://www.scribd.com/doc/46265644/Ingineria-programarii-Faza-de-analiza
13. http://users.cs.tuiasi.ro/~fleon/Curs_IP/IP09_SC1.pdf
14. http://www.geocities.ws/luciweb/c/ip_toc.html
15. http://www.cartigratis.eu/carte/250441/ingineria-programarii-diagrame-uml

# Anexe

## Anexa 1. Teme pentru laborator

1. Buletin de stiri (newsletter)
2. Bursa locurilor de muncă
3. Motor de testare și evaluare
4. Sistem de gestiune a învățării
5. Magazin virtual (de un anumit tip)
6. Rețea socială profesională
7. Aplicație de gestiune a evenimentelor culturale
8. Sistem de gestiune a cunoștințelor (de un anumit tip)
9. Aplicație de localizare a serviciilor de pe un dispozitiv mobil
10. Aplicație de gestiune a unui program muzical (playlist)
11. Aplicație de planificare a călătoriilor
12. Sistem rezervare bilete avion
13. Sistem de management pentru o firmă de curierat rapid
14. Agenda electronică
15. Sistem de management într-un SEL
16. Sistem pentru gestiunea cheltuielilor personale
17. Librărie online
18. Sistem de management pentru un centru medical
19. Piața auto
20. Sistem de management al proiectelor într-o companie

Fiecare student va participa în cadrul unei echipe la elaborarea unui proiect din lista temelor propuse (numarul de membri ai echipei este de 3 persoane).

*Raportul trebuie să conțină:*

1. Foaia de titlu
2. Scopul lucrării
3. Noțiuni generale
4. Desfășurarea lucrării
4. Diagrama UML
5. Concluzii

**Anexa 2. Ghid de utilizare a instrumentului Enterprise Architect**

*Enterprise Architect* prevede modelarea completă a ciclului de viață pentru:

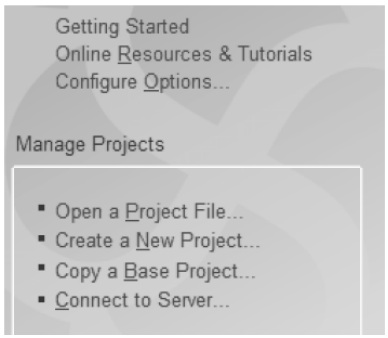
- Afaceri și sisteme IT;
- Software și ingineria sistemelor;
- Integrarea dezvoltării în timp real.

Integrând capacitățile de gestionare a cerințelor, Enterprise Architect ajută la urmărirea specificațiilor la nivel înalt pentru analiza, proiectarea, implementarea, testarea și întreținerea modelelor folosind UML.

Enterprise Architect este un instrument grafic, proiectat pentru a ajuta echipele, construind sisteme robuste și întreținute. Utilizând calitatea înaltă, integrarea raportării și documentării poate oferi o viziune împărtășită cu ușurință și precizie.

## Crearea unui proiect nou

Când lansaţi Enterprise Architect, se deschide pagina de start:



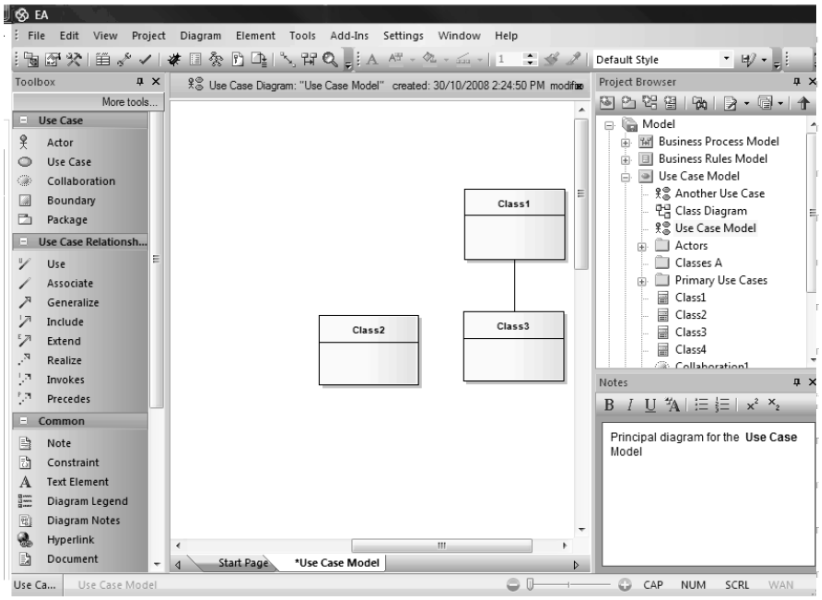

1.Selectați opţiunea *Create a New Project*. Se afişează fereastra de dialog *New Project*.

2. În câmpul *File name*, introduceți un nume semnificativ pentru proiect şi tastați butonul *Save* pentru a crea fişierul de proiect. Apoi se afişează fereastra de dialog *Select Model(s)*.

3. Selectăm unul sau mai multe template-uri model, prin selectarea checkbox-ului fiecărui model care vă interesează.

4. Tastaţi butonul OK. Enterprise Architect creează un pachet de model pentru fiecare şablon selectat şi îl afişează în browser-ul proiectului, în partea dreaptă a ecranului.

**View-ul Diagramelor** este fereastra principală a spaţiului de lucru, care ne permite să creăm şi să vizualizăm diagrame.

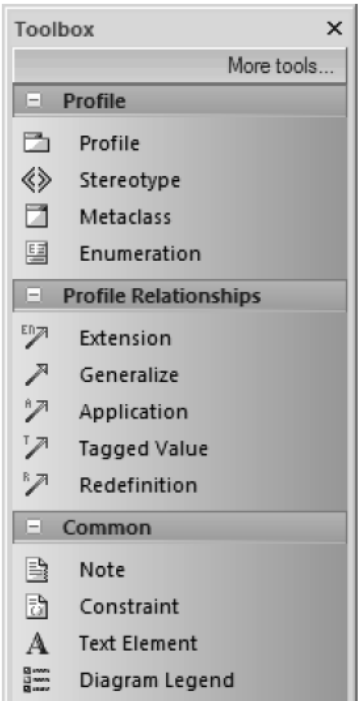



Putem deschide mai multe diagrame, dar putem vizualiza doar una la un moment dat. Diagrama se deschide prin dublu-clic pe numele diagramei în Project Browser. În acelaşi mod putem deschide şi celelalte diagrame, tastând pe hyperlink dintr-o diagramă deschisă, sau tastând pe elementele care conţin alte diagrame.

Utilizaţi View-ul Diagramelor pentru a construi relaţiile şi elementele modelului. În cadrul diagramei, avem posibilitatea să creăm elemente noi, să mişcăm elementele existente şi în general

putem să organizăm elementele şi relaţiile. Înţelegerea modului de lucru şi organizare a elementelor este esenţială, deoarece cel mai mult în View-ul Diagramelor se operează cu elementele. Utilizaţi exemplele furnizate de Enterprise Architect pentru a explora capacităţile şi comportamentul View-ul Diagramelor.

**Ce este Toolbox.** Enterprise Architect UML Toolbox este un panou de pictograme pe care le utilizăm pentru a crea elemente şi legături într-o diagramă. În cadrul Toolbox, elementele UML şi tipurile de legături sunt organizate în pagini, fiecare pagină conţinând elemente sau legături utilizate pentru un anumit tip de diagramă. Diagramele includ diagrame UML standard, diagrame Enterprise Arhitect extinse. Când deschideţi o diagramă, Toolbox de instrumente generează în mod automat elementele şi relaţiile corespunzătoare tipului de diagramă. Acest lucru nu ne împiedică să utilizăm alte elemente şi legături de pe alte pagini în diagrama dată, deşi unele asocieri sar putea să nu fie valide în UML.




**Crearea Elementelor şi legăturilor:**

1. În Project Browser, selectăm diagrama necesară. Diagrama se va deschide cu Toolbox-ul corespunzător tipului de diagramă selectat. (Dacă doriţi un set diferit de elemente şi legături, Selectaţi *More tools* şi selectaţi tipul de diagramă corespunzătoare).

2. Selectăm elementul necesar, de exemplu, elementul tip *clasă* sau relaţie de *Associere*.

3. Pentru crearea elementelor, tastăm cu mouse-ul în câmpul diagramei pentru a plasa noul element.

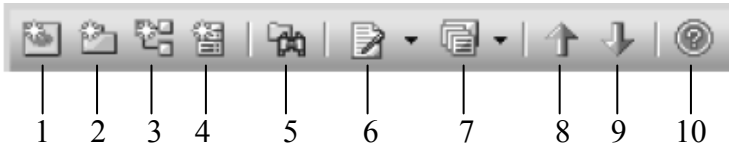
4. Pentru crearea legăturilor, tragem cursorul între elementul sursă şi elementul-destinaţie al diagramei. Pentru a schimba direcţia-legăturii, tastaţi [Shift], în timp ce schimbaţi direcţia cursorului. Alternativ, dacă tragem de la elementul sursă în spaţiul liber al diagramei, Quick-linker-ul vă permite să creaţi elementul destinaţie.

5. Editǎm proprietăţile elementului sau proprietăţile legăturilor, după dorinţă.

**Project Browser**

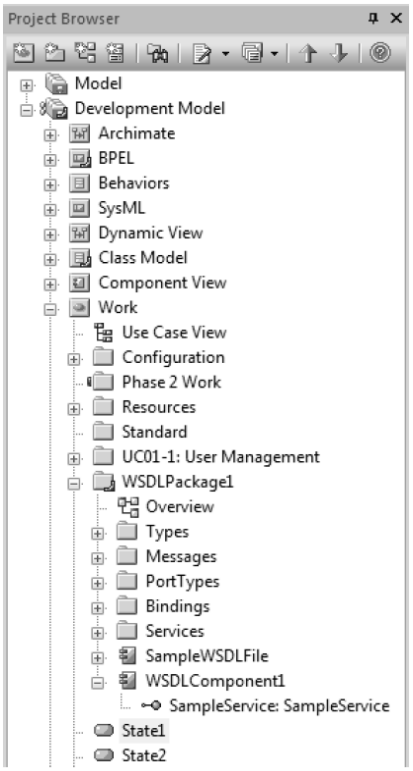
Project Browser vă permite să navigaţi prin spaţiul proiectului Enterprise Arhitect. El afişează pachete, diagrame, elemente şi proprietăţi ale elementelor.

Dacă tastăm clic dreapta pe un element în Project Browser, aveţi posibilitatea să efectuaţi acţiuni suplimentare, cum ar fi adăugarea de pachete noi, crearea de diagrame, redenumirea, crearea de documente şi de alte rapoarte şi ştergerea elementelor modelului. De asemenea, puteţi edita numele oricărui element din Proiect Browser prin selectarea elementului şi apăsând pe tasta [F2].




1. Creează un model nou în proiect, de la un model UML predefinit sau utilizând tehnologia şabloanelor.
2. Creează un pachet nou sub pachetul selectat.
3. Creează o diagramă nouă sub elementul sau pachetul selectat.
4. Creează un element nou sub elementul sau pachetul selectat.
5. Efectuează o simplă cautare după un text în Proiect Browser.

6. Oferă opţiuni pentru a genera un raport tip RTF, HTML sau doar un raport Diagramă a pachetului selectat în Proiect Browser.
7. Oferă opţiuni pentru generarea codului-sursă sau DDL, importul directorului sursă, modul binar sau schema bazei de date, generează conţinutul pachetului pentru a fi sincronizat cu codul pachetului sau resetarea codului sursă al limbajului, toate pentru pachetul selectat.
8. Plasează elementul selectat sau pachetul cu o poziție mai sus în Project Browser, în cadrul pachetului părinte.
9. Plasează elementul selectat sau pachetul cu o poziție mai jos în Project Browser, în cadrul pachetului-părinte.
10. Deschide Ajutorul Enterprise Architect în browserul proiectului.




Browser de proiect pote fi împărţit în vederi (views). Fiecare din aceste vederi conţine diagrame, pachete şi alte elemente. Mai jos este descrisă ierarhia unei vederi implicite, însă noi avem posibilitatea să creăm diferite vederi care se potriviesc cerinţelor proiectului:

| Vederi (View) |
| --- |
| Use Case View |
| Dynamic View |
| Logical View |
| Component View |
| Deployment View |
| Custom View |

*Tab-ul* Diagramei este situat în partea de jos sau în partea de sus a zonei diagramei, deasupra barei de stare.

Locaţia implicită este în partea de jos a zonei diagramei. De fiecare dată când deschideţi o diagramă, numele diagramei este afişat în *tab* pentru a o identifica şi accesa mai uşoar.


Observaţi că tab-ul AICollections este bold, acest lucru înseamnă că diagrama curentă este diagrama AICollections. De asemenea, observaţi că diagrama *Class Model* are un asterisc. Aceasta înseamnă că există modificări care nu au fost salvate în diagramă.

**Anexa 3. Titlul referatului lucrării de laborator**

**UNIVERSITATEA TEHNICĂ A MOLDOVEI**

**Catedra Calculatoare**

**Ingineria programării**

# Referat
# la lucrarea de laborator nr.1

# Tema: Elaborarea documentului de specificare a cerințelor


**Elaborat** **studentul gr. C-091**
**Ion Moraru**

**Verificat** **lect. univ. Diana Palii**


**Chişinău, 2012**

## Cuprins

**INGINERIA PROGRAMĂRII**
*Prezentare teoretică și aplicații*

Autori: Emilian Guțuleac
Diana Palii
Iurii Țurcanu

Redactor: Eugenia Balan

| | |
|---|---|
| Bun de tipar | Formatul hârtiei 60x84 1/16 |
| Hîrtie ofset. Tipar RISO | Tirajul 75 ex. |
| Coli de tipar 7,0 | Comanda nr. |

U.T.M. 2011, Chişinău, bd. Ştefan cel Mare, 168.
Secţia Redactare şi Editare a U.T.M.
2068, Chişinău, str. Studenţilor, 9/9.