UML – Unified Modeling Language

Introducere

Problema principală care apare în dezvoltarea programelor este complexitatea. Folosirea de modele poate înlesni abordarea de probleme complexe.

Un ***model*** este o simplificare unui anumit sistem, ce permite analizarea unora dintre proprietățile acestuia. Lucrul cu modelele poate facilita o mai bună înțelegere a sistemului modelat, datorită dificultății intrinseci de înțelegere a sistemului în întregul sau.

Folosirea tehnicii ” divide et impera” permite înțelegerea părților componente ale unui sistem, iar ansamblul sistemului ca o interacțiune între părțile acestuia. Un model reuşit reține caracteristicile importante ale obiectului modelat şi le ignoră pe cele irelevante. De remarcat că noțiunile de important / irelevant sunt relative, ele depinzând de scopul pentru care se face modelarea. Astfel apare plauzibilă construirea mai multor modele pentru un anumit obiect, fiecare surprinzând anumite aspecte ale acestuia.

Orice metodologie de dezvoltare a programelor abordează problema comunicării între membrii echipei. Este posibil să apară situații în care modelul construit de proiectant să nu fie înțeles exact de cel ce scrie cod, fie din cauza lipsei de precizie a modului în care este prezentat modelul, fie datorită lipsei unor elemente importante ale acestuia; adesea o serie de amănunte subtile ce sunt evidente pentru designer nu sunt transmise. O rezolvare a problemei comunicării ar fi ca aceeaşi persoană să fie implicată direct în toate fazele dezvoltării. Chiar şi aşa, apare des situația în care o persoană trebuie să continue munca alteia.

Se impune aşadar existența unui limbaj formal de comunicare al cerințelor.

Termenul formal este esențial. De obicei, chiar şi în proiecte de dimensiuni reduse, se face modelare, însă într-un limbaj specific celui care modelează, determinat de viziunea asupra problemei şi de pregătirea acestuia. Folosirea unui limbaj ”intern” nu trebuie privită ca negativă, ea având se pare un rol esențial în gândire în general şi în modelare în particular. Modelarea este posibilă, chiar şi fără folosirea explicită a unui limbaj.

Formalismul limbajului de modelare constă în stabilirea de elemente cu o semantică asupra căreia se convine şi cu ajutorul cărora să se poată transmite idei în mod cât mai eficient şi mai puțin ambiguu.

Limbajul de modelare UML îşi propune să definească o modalitate cât mai precisă, generală şi extensibilă de comunicare a modelelor. El a fost creat în primul rând pentru a facilita proiectarea programelor, însă datorită expresivității sale poate fi folosit şi în alte domenii (proiectare hardware, modelarea proceselor de afaceri etc.). Folosirea UML facilitează comunicarea modelelor, însă nu asigură crearea unui model bun. Limbajul de modelare nu specifică şi nu poate specifica ce modele trebuie create, în ce ordine şi cum trebuie ele utilizate pentru un sistem particular.

Începând cu mijlocul anilor 1970 şi continuând în anii 1980 au apărut diverse limbaje de modelare, odată cu creşterea experienței de lucru în cadrul paradigmei orientate obiect. Astfel, numărul de limbaje de modelare ajunge de la 10 până la mai mult de 50 în perioada 1989-1994.

Limbajele de modelare de succes din această perioadă sunt:

- ***Booch*** - potrivită mai ales pentru proiectare şi implementare, cu dezavantajul unor notații complicate;
- ***OOSE*** (Object-Oriented Software Engineering) - această metodă a propus aşa-numitele cazuri de utilizare, care ajutau la înțelegerea comportamentului întregului sistem;

- ***OMT*** - Object Modeling Technique - potrivită pentru analiză şi sisteme informaționale cu multe date.

La mijlocul anilor 1990, când este semnalată o nouă generație de limbaje de modelare, a început un proces de omogenizare, prin încorporarea în fiecare limbaj a caracteristicilor găsite în celelalte limbaje.

Cei trei autori ai limbajelor de modelare principale, Booch, Jacobson şi Rumbaugh au ajuns la concluzia ca ar fi mai bine să conducă evoluția limbajelor lor pe un acelaşi drum, pentru a elimina diferențele gratuite ce nu făceau decât sa încurce utilizatorii. Un al doilea motiv a fost unul pragmatic, reieşit din necesitatea industriei software de a avea o oarecare stabilitate pe piața limbajelor de modelare. Al treilea motiv a fost convingerea că prin unirea forțelor se pot aduce îmbunătățiri tehnicilor existente.

Dezvoltarea UML a început în mod oficial în octombrie 1994, când Rumbaugh s-a alăturat lui Booch în cadrul companiei Rational Software, ambii lucrând la unificarea limbajelor Booch şi OMT.

Versiunea preliminară 0.8 a Unified (Metoda Unificată - aşa cum era numită atunci) a fost publicată în octombrie 1995. Tot atunci, Jacobson s-a alăturat echipei de la Rational şi scopul UML a fost extins pentru a include şi OOSE. În iunie 1996 a fost publicată versiunea 0.9 a UML. Pe parcursul anului 1996, ca o consecință a reacțiilor comunității proiectanților de sistem, a devenit clar ca UML este văzut de către multe companii ca o opțiune strategică pentru dezvoltarea produselor lor.

A fost creat un consorțiu UML format din organizații doritoare să aloce resurse pentru a obține o definiție finală a UML. Dintre aceste companii care au contribuit la crearea UML 1.0 au făcut parte DEC, Hewlet-Packard, I-Logix, Intellicorp, IBM, MCI Systemhouse, Microsoft, Oracle, Rational, Texas Instruments etc.

UML 1.0 a fost propus spre standardizare în cadrul OMG în ianuarie 1997.

Până la sfârşitul anului 1997 echipa care lucra la UML s-a extins, urmând o perioadă în care UML a primit o specificare formală mai riguroasă. Versiunea UML 1.1 a fost adoptată ca standard de către OMG în noiembrie 1997.

Versiunea 1.2 a UML a fost o revizie editorială; versiunile 1.3 au fost publicate începând cu sfârşitul anului 1998.

În septembrie 2001 a fost publicată versiunea 1.4 şi în martie 2003 a fost versiunea 1.5.

În octombrie 2004 a fost introdusă versiunea 2.0.

UML este un limbaj de modelare bazat pe notații grafice folosit pentru a ***specifica, vizualiza, construi*** şi ***documenta*** componentele unui program.

UML este un limbaj cu ajutorul căruia se pot construi (descrie) modele. Un model surprinde un anumit aspect al unui program şi acelaşi model poate fi descris la diferite nivele de abstractizare. Fiecărui model îi corespunde o diagramă. Tipurile de diagrame existente în UML sunt:

- Diagrama cazurilor de utilizare (*Use Case Diagram*)
- Diagrama de clase (*Class Diagram*)
- Diagrame care descriu comportamentul:
  - Diagrame de interacțiuni (*Interactions Diagrams*)
    - Diagrama de secvență (*Sequence Diagram*)
    - Diagrama de colaborare (*Collaboration Diagram*)

- Diagrama de stări (*State chart Diagram*)
- Diagrama de activități (*Activity Diagram)*

➢ Diagrame de implementare:

- Diagrama de componente (*Component Diagram*)
- Diagrama de plasare (*Deployment Diagram*)

Fiecăreia din cele trei mari faze din dezvoltarea un proiect software îi corespund una sau mai multe diagrame UML şi anume:

- pentru faza de *analiza* se utilizează diagrama cazurilor de utilizare şi diagrama de activități;
- în faza de *proiectare* se folosesc: diagrama de clase pentru precizarea structurii sistemului şi diagramele de stări şi interacțiune pentru descrierea comportamentului acestuia;
- în faza de *implementare* se utilizează diagramele de implementare.

1. Diagrama cazurilor de utilizare (*Use Case Diagram*)

O ***diagramă use case*** este una din diagramele folosite în UML pentru a modela aspectele dinamice ale unui program alături de diagrama de activități, diagrama de stări, diagrama de secvență şi diagrama de colaborare. Altfel spus, diagramele use case se utilizează pentru:

- pentru a modela contextul unui sistem: determinarea granițelor sistemului şi a actorilor cu care acesta interacționează.
- pentru a modela cerințele unui sistem: *ce* trebuie să facă sistemul (dintr-un punct de vedere exterior sistemului) independent de *cum* trebuie să facă. Va rezulta specificarea comportamentului dorit. Sistemul apare ca o cutie neagră. Ceea ce se vede este cum reacționează el la acțiunile din exterior.

Elementele componente ale unei diagrame use case sunt:

- *use case*-uri;

- *actori;*

- *relațiile* care se stabilesc între use case-uri, între actori şi între use case-uri şi actori.

Use case-uri

Un *use case* (caz de utilizare) reprezintă cerințe ale utilizatorilor. Este o descriere a unei mulțimi de secvențe de acțiuni (incluzând variante) pe care un program le execută atunci când interacționează cu entitățile din afara lui (*actori*) şi care conduc la obținerea unui rezultat observabil şi de folos actorului. Un use case descrie ce face un program sau subprogram, dar nu precizează nimic despre cum este realizată (implementată) o anumită funcționalitate.

Fiecare use case are un *nume* prin care se deosebeşte de celelalte use case-uri. Acesta poate fi un şir arbitrar de caractere, însă de regulă numele sunt scurte fraze verbale care denumesc un comportament ce există în vocabularul sistemului ce trebuie modelat.

Figura 1.1 prezintă notația grafică pentru use case.

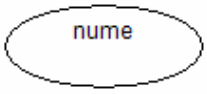


Figura 1.1: Notația grafică pentru use case

Comportamentul unui use case poate fi specificat descriind un flux de evenimente într-un text suficient de clar ca să poată fi înțeles de cineva din exterior (de exemplu utilizatorul). Acest flux de evenimente trebuie să includă cum începe şi se termină use case-ul atunci când acesta interacționează cu actori, ce obiecte sunt interschimbate, precum şi fluxuri alternative ale acestui comportament. Aceste fluxuri de evenimente reprezintă scenarii posibile de utilizare a sistemului.

Identificarea use case-urilor se face pornind de la cerințele utilizatorului şi analizând descrierea problemei.

Actori

Un *actor* reprezintă idealizarea unei peroane, proces sau obiect exterior care interacționează cu un sistem, subsistem sau o clasă. Actorii sunt entități exterioare sistemului. Ei pot fi utilizatori (persoane), echipamente hardware sau alte programe. Fiecare actor are un *nume* care indică rolul pe care acesta îl joacă în interacțiunea cu programul.

Notație grafică pentru un actor este ilustrată în figura 1.2.

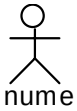


Figura 1.2: Notația grafică pentru actor

Există două moduri în care actorii pot interacționa cu un sistem:

- folosind sistemul, adică inițiază execuția unor use case-uri;
- sunt folosiți de către sistem, adică oferă funcționalitate pentru realizarea unor use case-uri.

Fiecare actor trebuie să comunice cu cel puțin un use case.

Pentru identificarea actorilor ar trebui să răspundem la următoarele întrebări:

- Cine foloseşte programul?
- De cine are nevoie programul pentru a-şi îndeplini sarcinile?
- Cine este responsabil cu administrarea sistemului?
- Cu ce echipamente externe trebuie să comunice programul?
- Cu ce sisteme software externe trebuie să comunice programul?
- Cine are nevoie de rezultatele (răspunsurile) oferite de program?

Relații

După cum am mai precizat, relațiile exprimă interacțiuni între use case-uri, între actori şi între use case-uri şi actori. Relațiile pot fi de mai multe tipuri: asociere, dependență şi generalizare.

Relația de asociere se definește între actori şi use case-uri, sau între use case-uri. Este folosită pentru a exprima interacțiunea (comunicarea) între elementele pe care le unește. Relația de asociere se reprezintă grafic printr-o linie şi este evidențiată în exemplele din figurile 1.3 şi 1.4.

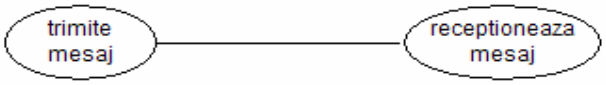


Fig. 1.3. Exemplu de asociere între use case-uri

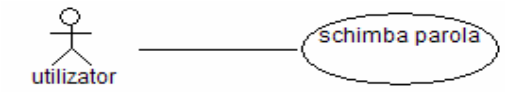


Fig. 1.4. Exemplu de asociere între actor şi use case

Relația de dependență se poate stabili numai între use case-uri. Acest tip de relație modelează două situații:

- cazul în care un use case foloseşte funcționalitatea oferită de un alt use case - dependența de tip *include*;
- există variante ale aceluiaşi use case – dependența de tip *extend*.

Dependența de tip ***include***. Notația grafică este dată în figura 1.5.

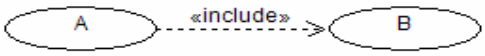


Fig. 1.5. Dependență de tip *include*

În acest caz comportamentul use case-ului B este inclus în use case-ul A. B este de sine stătător, însă este necesar pentru a asigura funcționalitatea use case-ului de bază A. În exemplul din figura 1.6, use case-ul `Stabileste grupul care lucreaza la campanie` are o relație de dependență de tip *include* cu use case-ul `Gaseste campanie`. Aceasta înseamnă că atunci când actorul `Manager campanie` utilizează `Stabileste grupul care lucreaza la campanie`, comportamentul use case-ului `Gaseste campanie` va fi inclus pentru a putea selecta o campanie relevantă.

Fig. 1.6. Diagramă use case cu dependență de tip *include*

Dependența de tip *include* se foloseşte şi pentru a scoate în evidență un comportament comun (B poate fi inclus în mai multe use case-uri de bază – vezi figura 1.7).

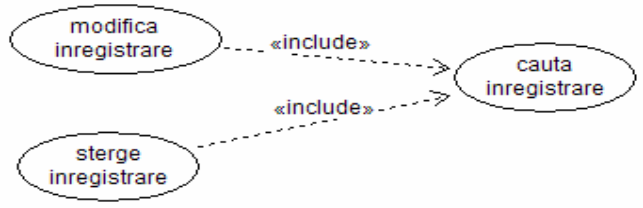


Fig. 1.7. Dependență de tip *include* în care un use case este inclus în mai multe use case-uri de bază

Dependența de tip ***extend***. Notație grafică se poate vedea în figura 1.8.

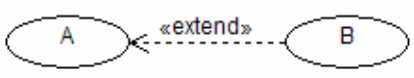


Fig. 1.8. Dependență de tip *extend*

În acest caz comportamentul use case-ului B poate fi înglobat în use case-ul A. A şi B sunt de sine stătătoare. A controlează dacă B va fi executat sau nu (vezi exemplul din figura 1.9).

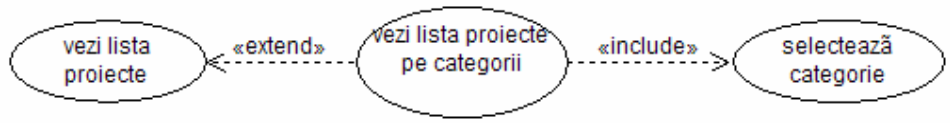


Fig. 1.9. Exemplu de dependență de tip *extend*

Într-o dependență de tip *extend* pot apărea aşa numitele *puncte de extensie* care specifică locul în care use case-ul specializat (B) extinde use case-ul de bază (A). Pentru fiecare use case pot fi specificate mai multe puncte de extensie. Fiecare dintre aceste puncte trebuie să aibă un nume. Aceste nume trebuie să fie unice, însă nu este obligatoriu ca ele să coincidă cu numele use case-urilor specializate. De asemenea, trebuie precizată condiția de care depinde apelul use case-ului specializat. Acest tip de relație se foloseşte pentru a modela alternative. În figura 1.10 este prezentată o diagramă use case cu dependență *extend* care are puncte de extensie.

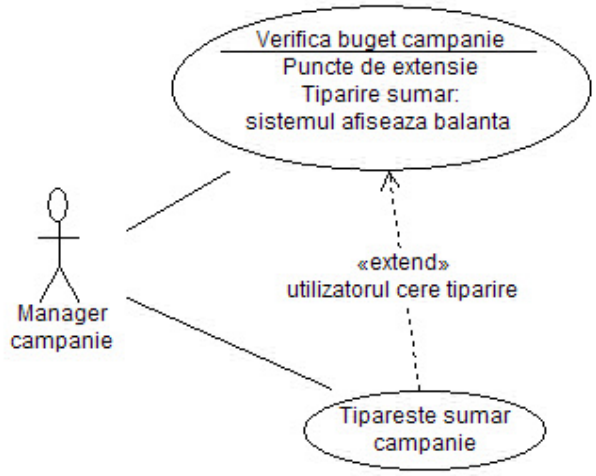


Fig. 1.10. Diagramă use case cu dependență de tip *extend* cu punct de extensie

Relația de generalizare se stabileşte între elemente de acelaşi tip (doi actori, sau doua use case-uri).

Este similară relației de generalizare (moştenire) care se stabileşte între clase. Figura 1.11 ilustrează notația grafică pentru relația de generalizare între use case-uri. Elementul derivat B moşteneşte comportamentul şi relațiile elementului de bază A. Use case-ul B este o specializare a use case-ului A.

În cazul unei relații de generalizare între use case-uri comportamentul poate fi modificat sau extins; use case-ul derivat B poate înlocui în anumite situații use case-ul de bază A. Case-ul derivat B controlează ce anume se execută şi ce se modifică din use case-ul de bază A.

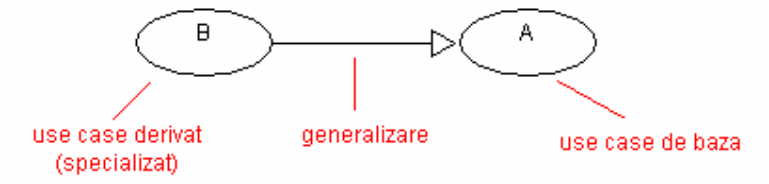


Fig. 1.11. Notația grafică pentru relația de generalizare între use case-uri

În figura 1.12 este prezentat un exemplu de relație de generalizare între use case-uri.

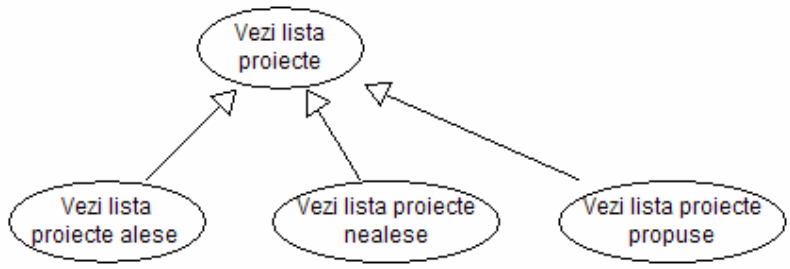


Fig. 1.12. Exemplu de relație de generalizare între use case-uri

După cum am precizat mai sus, relația de generalizare se poate aplica şi între actori. În exemplul din figura 1.13 este prezentată o relație de generalizare între actori.

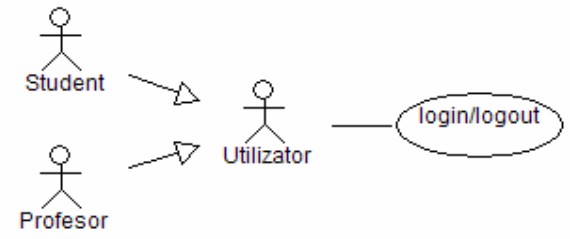


Fig. 1.13. Relație de generalizare între actori

În exemplul din figura 1.14a actorii A şi B joacă acelaşi rol R atunci când interacționează cu use case-ul UC şi joaca roluri diferite în interacțiunea cu use case-ul UC în figura 1.14b.

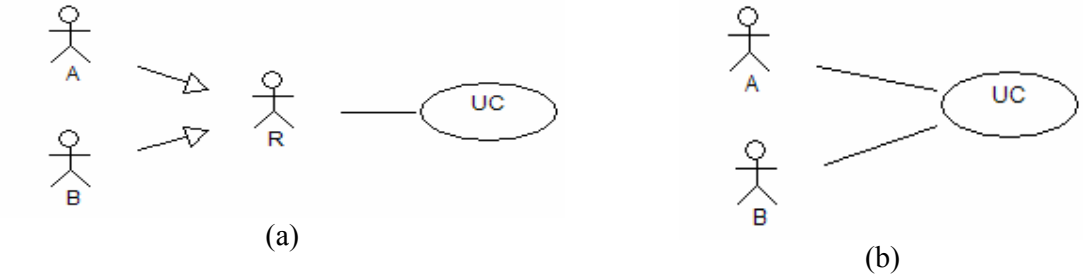


Fig. 1.14. Tipuri de roluri jucate de actori în interacțiunea cu use case-ul

În figura 1.15 este prezentată o diagramă use case care utilizează toate tipurile de relații definite anterior.

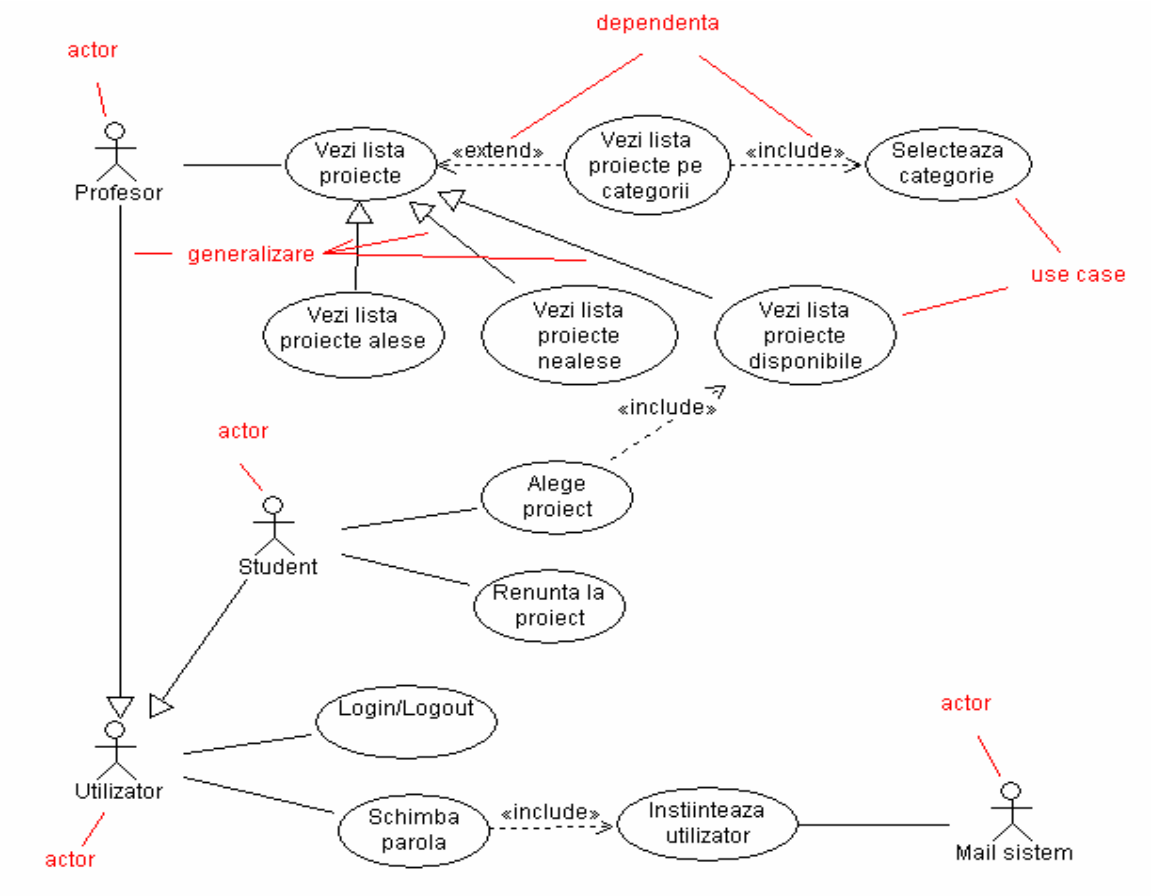


Fig. 1.15. Exemplu de diagramă use case

2. Diagrama de clase (*Class Diagram*)

Diagrama de clase este folosită pentru a modela structura (viziunea statică asupra) unui sistem. O astfel de diagramă conține clase / interfețe, obiecte şi relații care se stabilesc între acestea. Relațiile pot fi de tipul:

- asociere;
- agregare;
- generalizare;
- dependență;
- realizare.

Clasele sunt folosite pentru a surprinde vocabularul sistemului ce trebuie dezvoltat. Ele pot include:

- abstracții care fac parte din domeniul problemei;
- clase necesare la momentul implementării.

O clasă poate reprezenta entități software, entități hardware sau concepte. Modelarea unui sistem presupune identificarea elementelor importante din punctul de vedere al celui care modelează. Aceste elemente formează vocabularul sistemului. Fiecare dintre aceste elemente are o mulțime de proprietăți.

Clase

Elementele unei clase sunt:

Nume - prin care se distinge de alte clase - o clasă poate fi desenată arătându-i numai numele;

Atribute - reprezintă numele unor proprietăți ale clasei;

Operații (metode) - reprezintă implementarea unor servicii care pot fi cerute oricărui obiect al clasei.

Notația grafică pentru clasă poate fi observată în figura 2.1.

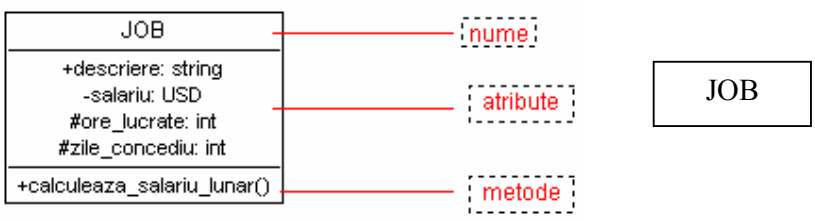


Figura 2.1. Notația grafică pentru clasă

Specificatorii de vizibilitate au următoarea semnificație:

- \+ public (dreptul de utilizare se acordă şi celorlalte clase);
- \- private (numai clasa dată poate folosi atributul sau operația);
- \# protected (posibilitatea de utilizare este accesibilă numai claselor succesoare)

O clasă poate avea oricâte atribute şi operații sau poate să nu aibă nici un atribut sau nici o operație. Modelarea vocabularului unui sistem presupune identificarea elementelor pe care utilizatorul sau programatorul le foloseşte pentru a descrie soluția problemei. Pentru fiecare element se identifică o mulțime de responsabilități (ce trebuie să facă acesta), după care se definesc atributele şi operațiile necesare îndeplinirii acestor responsabilități.

Există trei tipuri particulare de clase (***stereotipuri***):

- *clasa limită* – realizează interacțiunea cu utilizatorii sau cu alte sisteme (notație - figura 2.2a); permit interfațarea cu alte sisteme software şi cu dispozitive fizice ca imprimante, motoare, senzori etc.
- *clasa entitate* – modelează obiecte din interiorul domeniului aplicației, dar externe sistemului software, despre care sistemul trebuie să stocheze câteva informații (notație - figura 2.2b);
- *clasa de control* – modelează coordonarea, secvențierea, controlul altor obiecte (notație - figura 2.2c).; recomandarea este să existe câte o clasă de control pentru fiecare use-case.

a. b. c.

Figura 2.2. Notații grafice pentru stereotipuri

Obiecte

Obiectele sunt instanțe ale claselor. Obiectele au identitate şi valori ale atributelor. Notația grafică pentru obiect se poate observa în figura 2.3.

Figura 2.3 Notații grafice pentru obiecte

Interfețe

Interfața specifică un grup de operații caracteristice unei comportări determinate a unei clase. Se modelează cu acelaşi simbol ca şi clasele. Interfața are numai operații. Pentru a le putea deosebi de clase se plasează stereotipul <<interface>> sau caracterul "I" la începutul numelui interfeței respective.

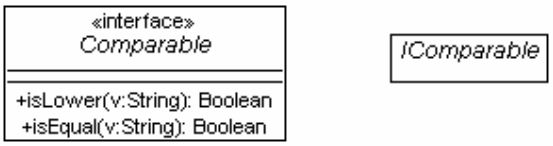


Figura 2.4. Notații grafice pentru interfețe

Clasă parametrizată (template)

Clasa parametrizată are unul sau mai mulți parametri formali. Prin intermediul acestei clase se poate defini o familie de clase dând valori parametrilor formali. De obicei parametrii reprezintă tipuri ale atributelor. Notația grafică se poate vedea în figura 2.5.

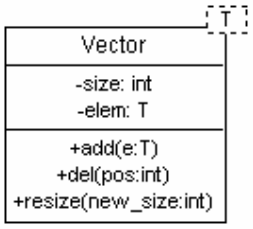


Figura 2.5. Exemplu de clasă parametrizată

Relații

O relație modelează o conexiune semantică sau o interacțiune între elementele pe care le conectează. În modelarea orientată obiect cele mai importante relații sunt relațiile de asociere, dependența, generalizare şi realizare. Un caz particular al relației de asociere îl constituie relația de agregare. Între clase se pot stabili relații de generalizare, dependență şi realizare. Relațiile de asociere şi agregare se stabilesc între instanțe ale claselor.

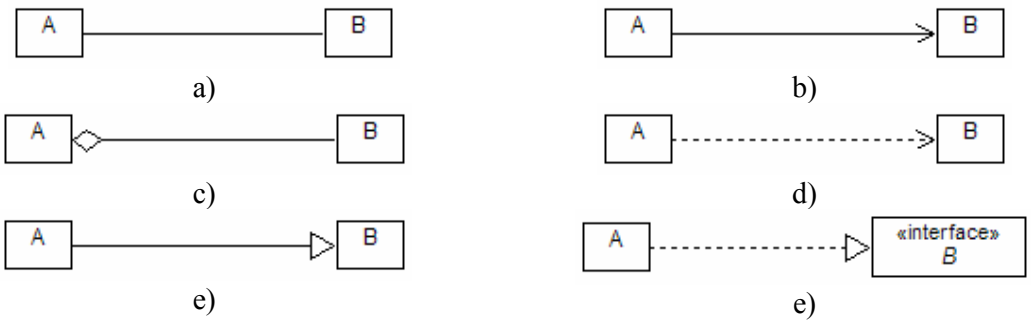


Figura 2.6. Notații grafice pentru diferite tipuri de relații: a)asociere bidirecțională; b)asociere unidirecțională; c) agregare; d) dependență; e) generalizare, f) realizare

Relația de asociere

Relația de asociere exprimă o conexiune semantică sau o interacțiune între obiecte aparținând unor anumite clase. Asocierea poartă informații despre legăturile dintre obiectele unui sistem. Pe măsură ce sistemul evoluează, pot fi create legături noi între obiecte sau pot fi distruse legăturile existente. Relația de asociere oferă baza arhitecturală pentru modelarea conlucrării şi interacțiunii dintre clase.

O asociere interacționează 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 înspre respectivul capăt de asociere. Cea mai importantă caracteristică a lor este multiplicitatea, ce specifică câte instanțe ale unei clase corespund unei singure instanțe a altei clase. De obicei multiplicitatea este folosită pentru asociațiile binare.

Reprezentarea grafică a asocierii este o linie (sau drum) ce conectează clasele ce participă în asociere. Numele asocierii este plasat pe linie, iar multiplicitatea şi rolurile sunt plasate la extremitățile sale (figura 2.7).

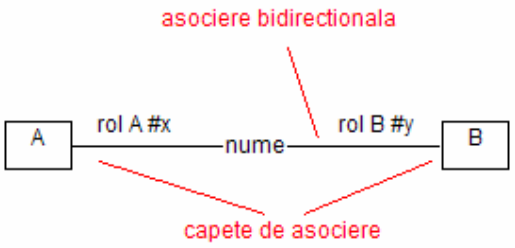


Figura 2.7: Notația grafică detaliată a relației de asociere

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

Este posibilă specificarea direcției unei asocieri, pentru a elimina legături redundante sau irelevante pentru un anumit model. În această situație notația grafică pentru relația de asociere este o linie cu o săgeată la unul din capete indicând direcția asocierii (figura 2.6b).

Relația de agregare

Relația de agregare este o asociere ce modelează o relație *parte-întreg*. Este reprezentată ca un romb gol atașat la capătul asocierii de lângă clasa agregat (figura 2.6c). În figură o instanță a clasei A conține o instanță a clasei B (altfel spus un obiect B este o parte unui obiect A). Relația de agregare este deci un caz particular al relației de asociere. Ea poate avea toate elementele unei relații de asociere, însă în general se specifică numai multiplicitatea. Se foloseşte pentru a modela situațiile în care un obiect este format din mai multe componente.

Compoziția este o formă mai puternică a agregării. *Partea* are „timpul de viață” al *întregului*. *Întregul* poate avea responsabilitatea directă pentru crearea sau distrugerea *părții* sau poate accepta o altă *parte* creată şi mai apoi să paseze „responsabilitatea” altui *întreg*.

Obs. Instanțele nu pot avea relații de agregare ciclice (o *parte* nu poate conține *întregul*)

În figura 2.8 este prezentat un exemplu în care se foloseşte agregarea şi compoziția.

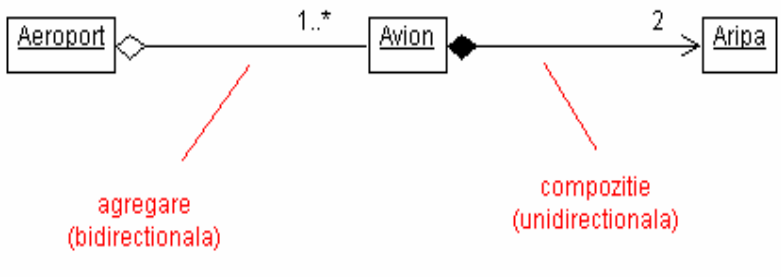


Figura 2.8. Exemplu de relații de agregare şi compoziție

Relația de dependență

O dependență (figura 2.6d) indică o relație semantică între două elemente ale modelului. Se foloseşte în situația în care o schimbare în elementul destinație (B) poate atrage după sine o schimbare în elementul sursă (A). Această relație modelează interdependențele ce apar la implementare.

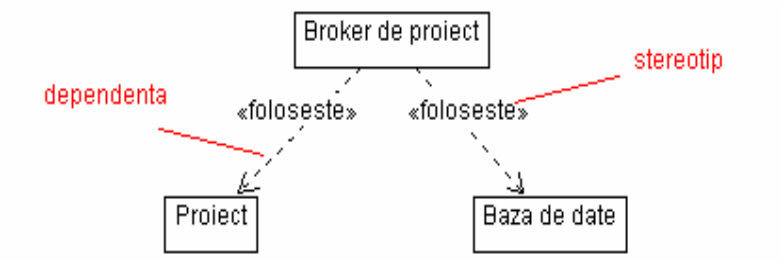


Figura 2.9. Exemplu de relații de dependență

Dependențele pot să apară şi la structurarea unui sistem în pachete, definind arhitectura sistemului.

Relația de generalizare

Relația de generalizare (figura 2.6e) se foloseşte pentru a modela conceptul de moştenire între clase. În figură clasa A este derivată din clasa B. Spunem că A este clasa derivată (sau subclasa, sau clasa copil), iar B este clasa de baza (sau superclasă, sau clasă părinte).

Relația de generalizare mai poartă denumirea de relație de tip *is a* (*este un fel de*), în sensul că o instanță a clasei derivate A este în acelaşi timp o instanță 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.

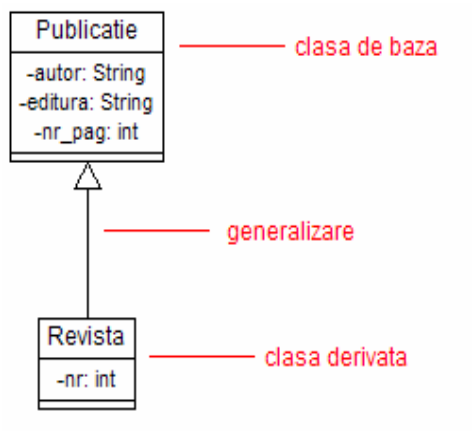


Figura 2.10. Exemplu de relație de generalizare

Relația de realizare

Relația de realizare (figura 2.6f) se foloseşte în cazul în care o clasă (A) implementează o interfață (B). Se spune că A realizează interfața specificată de B. O interfață este o specificare comportamentală fără o implementare asociată. O clasă include o implementare. Una sau mai multe clase pot realiza o interfață prin implementarea metodelor definite de acea interfață.

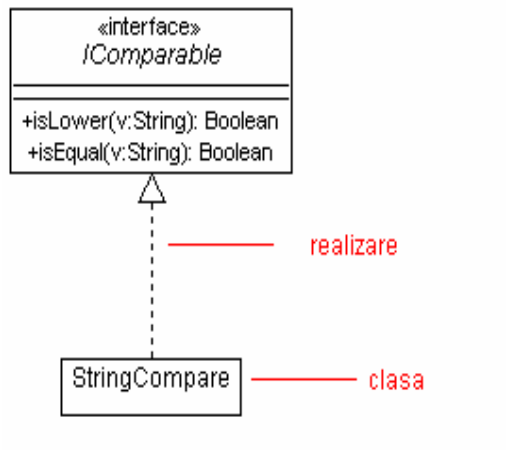


Figura 2.11. Exemplu de relație de realizare.

În figura 2.11 sunt prezentate două diagrame de clase în care se folosesc toate tipurile de relații prezentate mai sus.

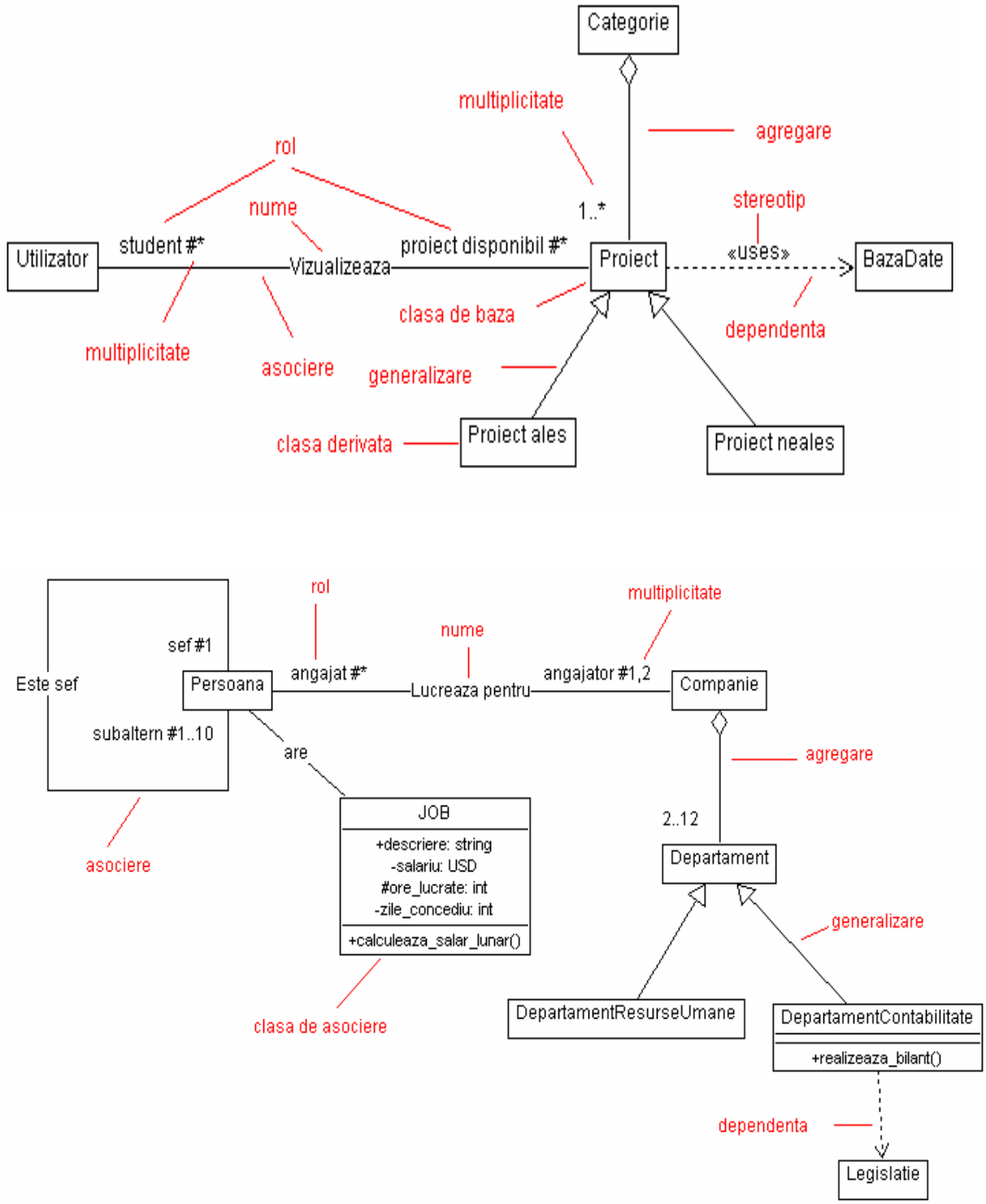


Figura 2.12. Exemple de diagrame de clase

3. Diagrama de stări (*State chart Diagram*)

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

- diagrama de stări
- diagrama de activități
- diagrama de interacțiuni:
  - diagrama de secvențe
  - diagrama de colaborare

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

Noțiuni preliminare

Un ***eveniment*** reprezintă ceva (atomic) ce se întâmplă la un moment dat şi care are ataşată o locație în timp şi spațiu. Evenimentele modelează apariția unui stimul care poate conduce la efectuarea unei tranziții 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 execuției 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 recepționat de altul (ex: excepții).
- ***apeluri de operații*** (de obicei sincrone): un obiect invoca o operație pe un alt obiect; controlul este preluat de obiectul apelat, se efectuează operația, obiectul apelat poate trece într-o nouă stare, după care se redă controlul obiectului apelant.
- ***trecerea timpului***
- ***o schimbare a rezultatului evaluării unei condiții***

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

Prin ***activitate*** se înțelege execuția neatomică a unor acțiuni. Activitățile au durată în timp, pot persista pe toată durata stării şi poate fi întreruptă (ex.: execuția unei funcții).

O diagramă de stări poate conține *stări* şi *tranziții*.

Starea

Prin *stare* se înțelege o condiție sau situație din viața unui obiect în timpul căreia acesta:

- satisface anumite condiții;
- efectuează o activitate;
- aşteaptă apariția unui eveniment.

Notația grafică pentru stare este prezentată în figura 3.1.

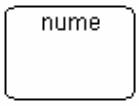


Figura 3.1: Notația grafică pentru stare

Există două stări particulare şi anume *starea inițială* şi *starea finală*.

Starea inițială (figura 3.2a) este starea din care pleacă entitatea modelată.

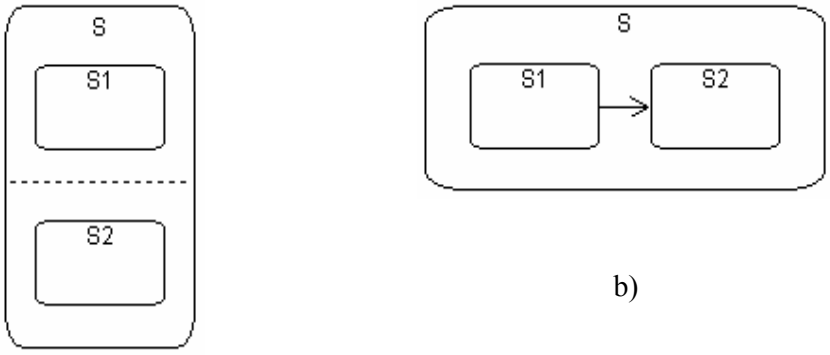
Starea finală (figura 3.2b) este starea în care entitatea modelată îşi încheie existența.

a) b)

Figura 3.2: Notații grafice pentru starea inițială (a) şi starea finală (b)

Elementele care caracterizează o stare sunt:

- ***Nume*** - identifică în mod unic o stare; numele este reprezentat de o succesiune de şiruri de caractere.
- ***Acțiuni de intrare/ieşire*** - sunt acțiuni ce se produc la intrarea, respectiv ieşirea din starea respectivă.
- ***Substări*** care pot fi
  - *concurente* (simultan active) – figura 3.3a
  - *disjuncte* (secvențial active) –figura 3.3b



a) b)

Figura 3.3. Notații grafice pentru substări concurente (a) şi disjuncte (b)

- ***Tranziții interne*** - sunt acțiuni şi activități pe care obiectul le execută cât timp se află în acea stare; se produc între substări şi nu produc schimbarea stării obiectului.

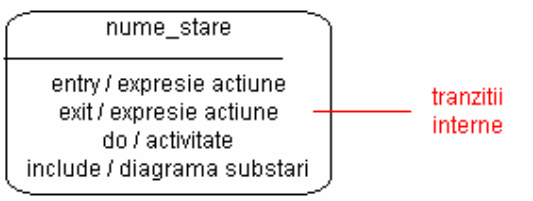


Figura 3.4. Notația completă pentru stare

nume_eveniment – identifică circumstanțele în care acțiunea specificată se execută;

condiție_gardă – condiție booleană care se evaluează la fiecare apariție a evenimentului specificat; acțiunea se execută doar când rezultatul evaluării este `true;`

acțiunea – poate folosi atribute şi legături care sunt vizibile entității modelate.

După cum se poate observa din figura 3.4, două evenimente au notații speciale: `entry` şi `exit`. Aceste evenimente nu pot avea condiții gardă deoarece se invocă implicit la intrarea, respectiv ieşirea din starea respectivă.

Activitățile sunt precedate de cuvântul cheie `do`.

Pentru a arăta că o stare conține substări, se foloseşte cuvântul cheie `include`, urmat de numele diagramei substărilor.

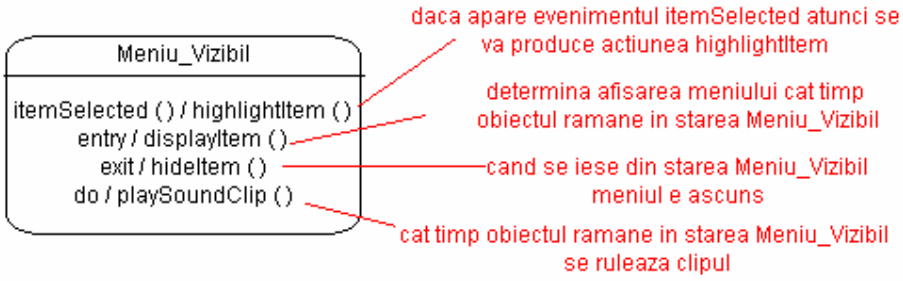


Figura 3.5. Exemplu de stare

Tranziția

O *tranziție* reprezintă o relație între două stări indicând faptul că un obiect aflat în prima stare va efectua nişte acțiuni şi apoi va intra în starea a doua atunci când un anumit eveniment se petrece. Notația grafică pentru tranziție se poate observa în figura 3.6.

Starea sursă reprezintă starea din care se pleacă.

Eveniment este evenimentul care declanşează tranziția.

Condiție gardă (guard condition) este o expresie booleană. Aceasta se evaluează la producerea evenimentului care declanşează tranziția. Tranziția poate avea loc numai dacă condiția este satisfăcută.

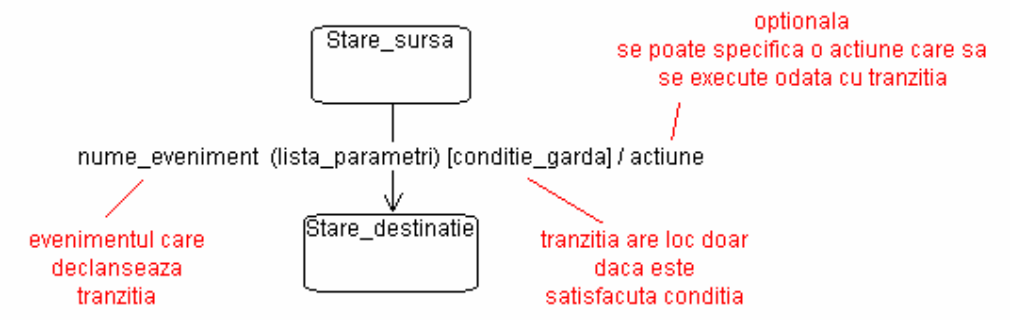


Figura 3.6. Notația grafică pentru tranziție

Acțiune - opțional se poate specifica o acțiune care să se execute odată cu efectuarea tranziției.

Starea destinație reprezintă starea în care ajunge obiectul după efectuarea tranziției.

În figurile 3.7 şi 3.8 se pot observa exemple de stări cu substări disjuncte, respectiv concurente.

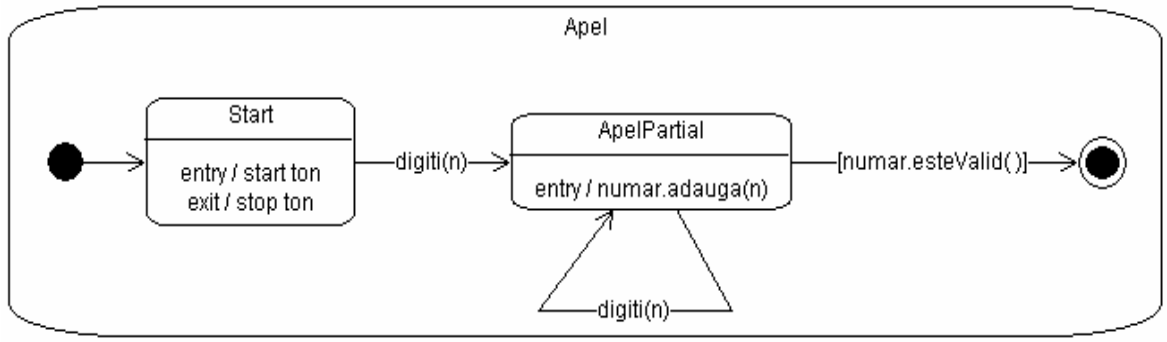


Figura 3.7. Exemplu de stare cu substări disjuncte

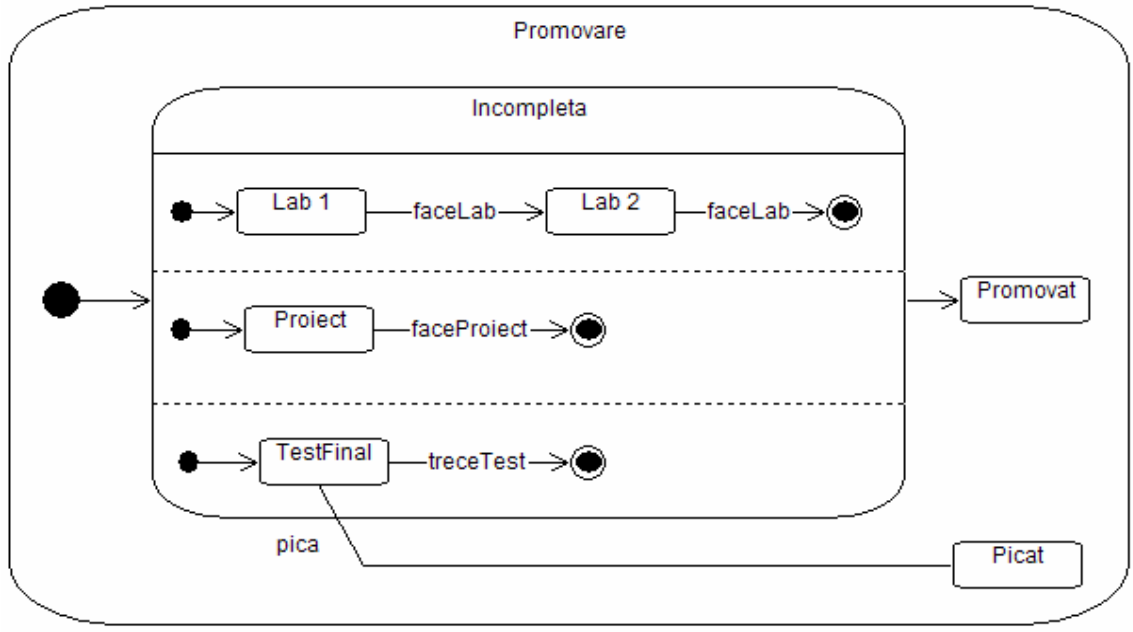


Figura 3.8. Exemplu de stare cu substări concurente

În figura 3.9 este prezentat un exemplu de diagramă de stare pentru un proiect propus de un student

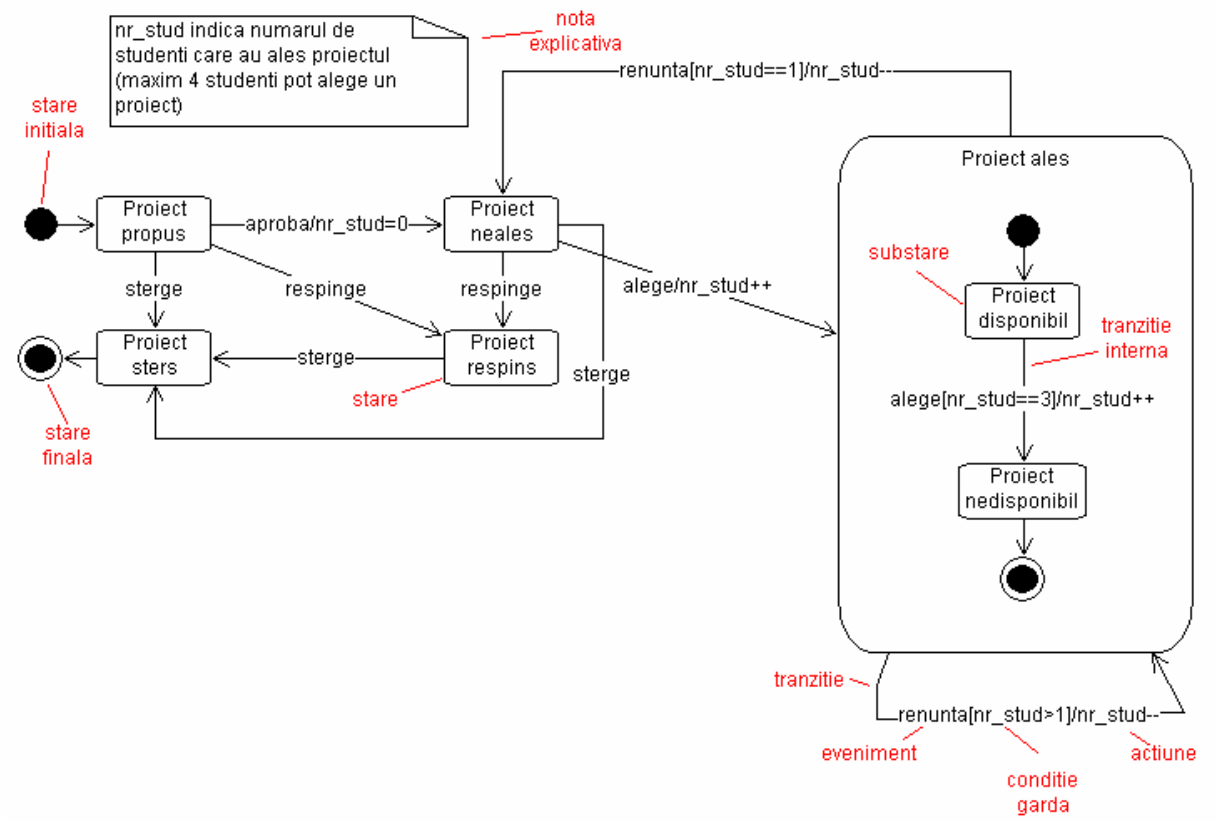


Figura 3.9. Exemplu de diagramă de stare pentru un proiect propus de un student

4. Diagrama de activități (*Activity Diagram*)

Diagrama de activități este o variantă a diagramei de stare şi este folosită pentru a modela dinamica unui proces sau a unei operații.

Diagramele de activități scot în evidență controlul execuției de la o activitate la alta. Diagramele de activități pot conține:

- stări activități şi stări acțiuni, care sunt stări ale sistemului;
- tranziții;
- obiecte;
- bare de sincronizare;
- ramificații.

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

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

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

O stare acțiune reprezintă execuția unei acțiuni.

Ea nu poate fi descompusă.

Stările acțiuni sunt atomice (nu pot fi întrerupte chiar dacă se produc evenimente) şi au o durată nesemnificativă (foarte mică).

Notația grafică a stărilor activitate/acțiune se poate observa în figura 4.1.

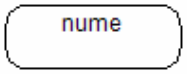


Figura 4.1. Notația grafică a stărilor activitate/acțiune

Tranzițiile – reprezintă relații între două activități.

Tranziția este inițiată de terminarea primei activități şi are ca efect preluarea controlului execuției de către a doua activitate.

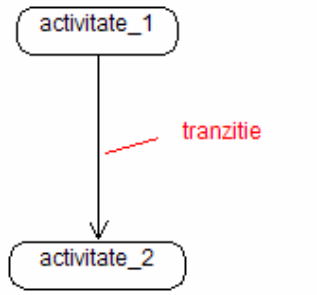


Figura 4.2. Notația grafică a tranziției

În exemplul din figura 4.3, prima activitate este aceea în care se adaugă un client nou. Tranziția la a doua activitate (şi anume aceea de a atribui un staff de contact pentru o eventuală campanie de promovare) implică faptul ca odată ce prima activitate s-a terminat, a doua activitate este pornită.

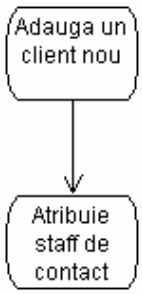


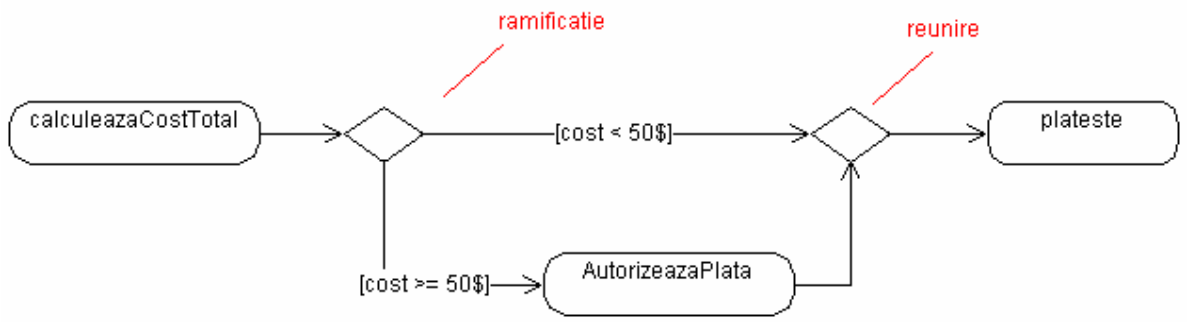
Figura 4.3. Exemplu de două activități unite de o tranziție

Ramificațiile - se folosesc pentru a modela alternative (decizii) a căror alegere depinde de o expresie booleană Au o tranziție de intrare şi două sau mai multe tranziții de ieşire

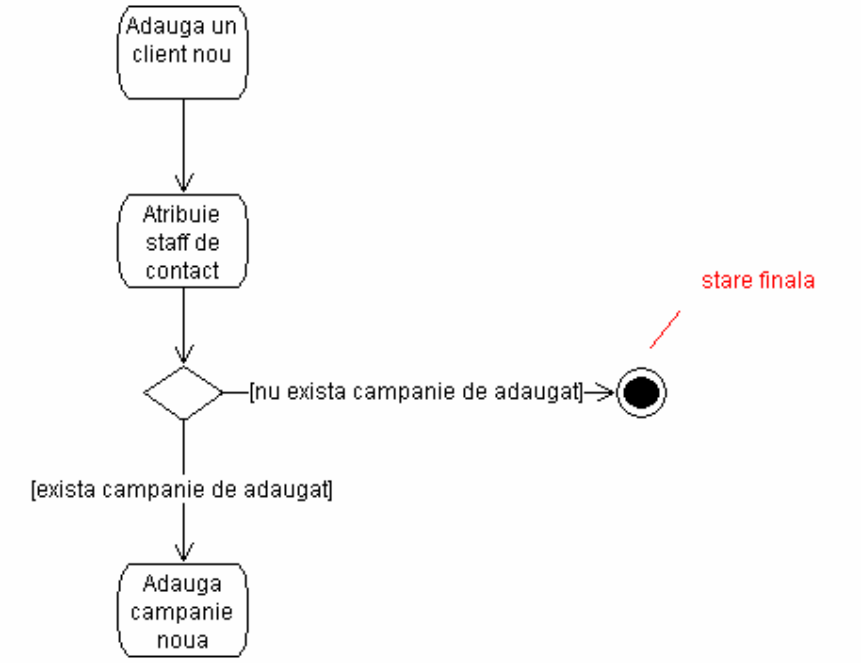
Fiecare tranziție de ieşire trebuie să aibă o condiție gardă

Condițiile gardă trebuie să fie disjuncte (să nu se suprapună) şi să acopere toate posibilitățile de continuare a execuției (vezi exemplele din figura 4.4), altfel fluxul de control al execuției va fi ambiguu (nu se ştie pe care cale se va continua execuția).

Condițiile trebuie însă să acopere toate posibilitățile, altfel sistemul se poate bloca.



a)



b)

Figura 4.4. Exemple de activități cu puncte de ramificație

Uneori nu este necesară precizarea explicită a unui punct de decizie, pentru a simplifica diagrama (vezi exemplul din figura 4.5).

În figurile 4.4 şi 4.5 apare un alt element al diagramelor de activități şi anume *starea finală*.

În general, odată încheiată ultima activitate dintr-o diagramă, trebuie marcată tranziția spre starea finală. De asemenea, după cum se poate observa în figura 4.6, fiecare diagramă de activități trebuie să înceapă cu *starea inițială*.

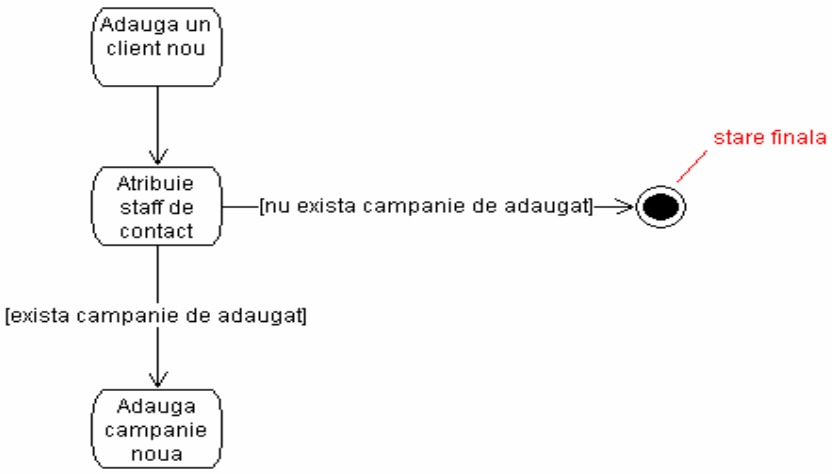


Figura 4.5. Exemplu de alegere reprezentată fără un punct de ramificație explicit

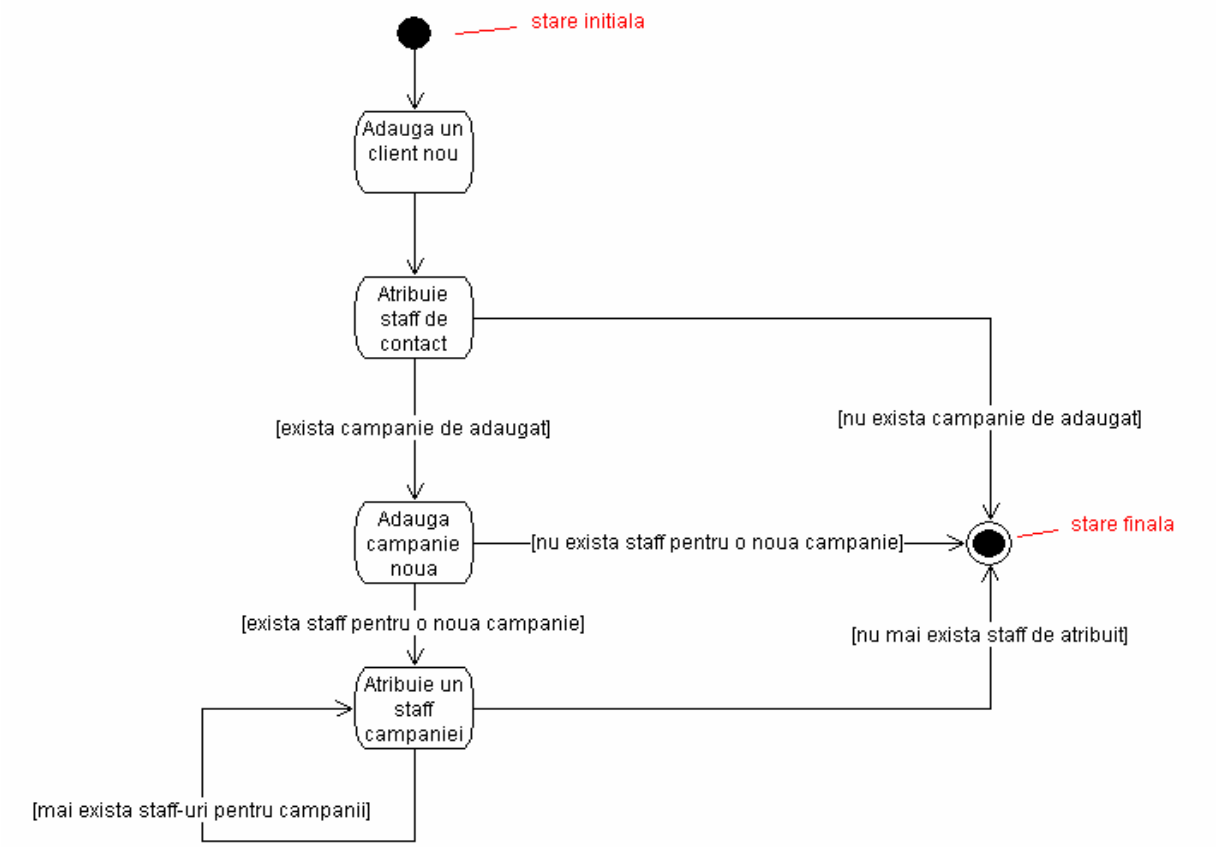


Figura 4.6. Diagramă de activități cu stare inițială şi finală

Bare de sincronizare

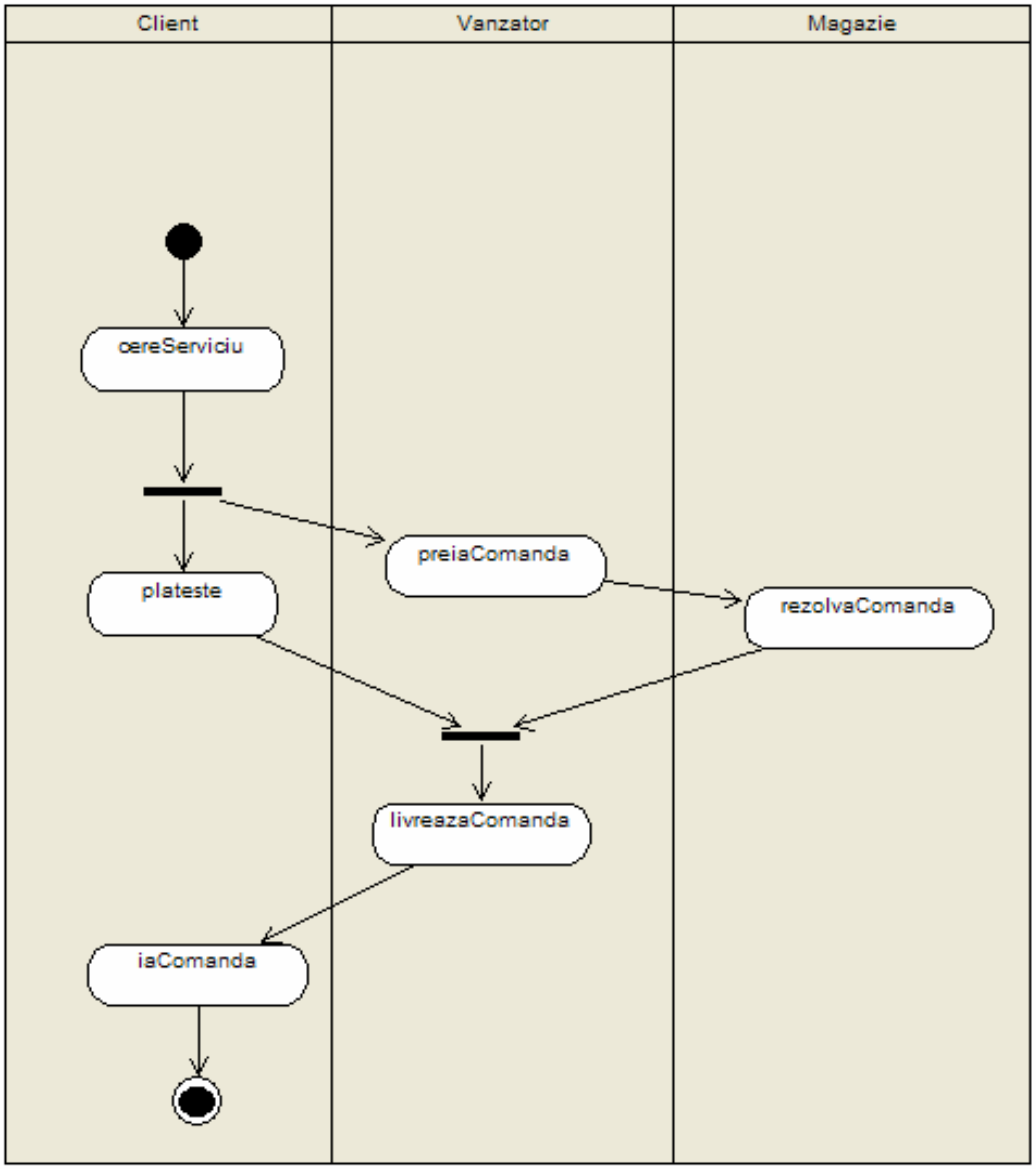
Există posibilitatea ca mai multe activități să se execute în paralel. Pentru sincronizarea acestora se folosesc aşa numitele bare de sincronizare. Acestea pot fi de două feluri:

- ***fork*** - poate avea o tranziție de intrare şi două sau mai multe tranziții de ieşire, fiecare tranziție de ieşire prezentând un flux de control independent. Activitățile de sub *fork* sunt concurente.

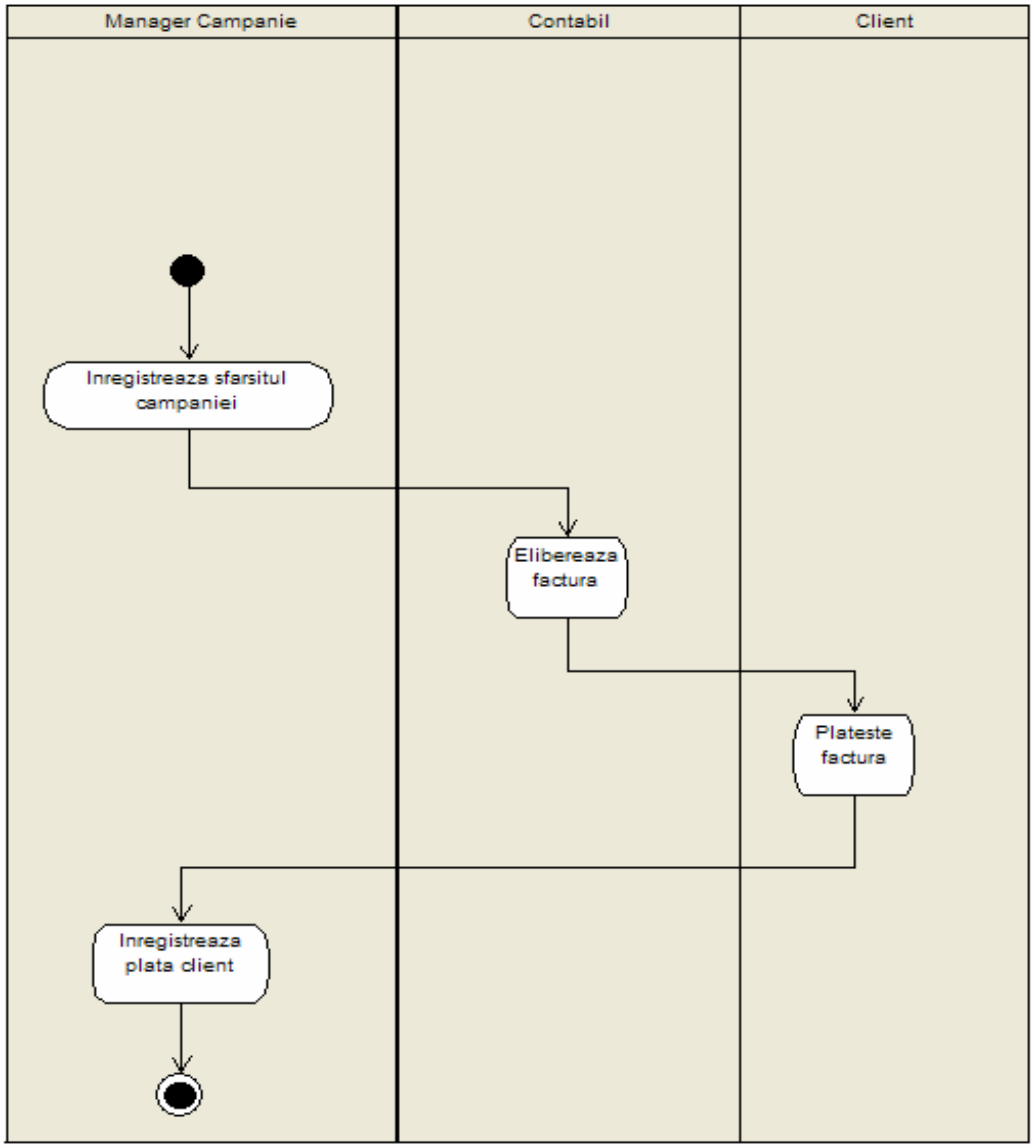
- ***join*** - reprezintă sincronizarea a două sau mai multor fluxuri de control. La *join* fiecare flux de control aşteaptă până când toate celelalte fluxuri de intrare ajung în acel punct. Poate avea două sau mai multe tranziţii de intrare şi o singură tranziţie de ieşire.

Figura 4.7. Notaţia grafică pentru barele de sincronizare

Conceptul de „swimlanes” modelează (arată) activităţile care au loc în interiorul unui sistem. Diagrama se împarte în coloane care se intitulează semnificativ pentru entitatea pe care o modelează (vezi figura 4.8a,b).



a)



b)

Figura 4.8. Diagrame de activități cu „swimlanes”

Obiecte

Acțiunile sunt realizate de către obiecte sau operează asupra unor obiecte. Un obiect poate interveni într-o diagramă de activități în două moduri:

- o operație a unui obiect poate fi folosită drept nume al unei activități (figura 4.9);
- un obiect poate fi privit ca intrare sau ieşire a unei activități (figura 4.10).

Obiectele pot fi conectate de acțiuni prin linii punctate cu o săgeată la unul din capete (orientarea săgeții indică tipul parametrului - intrare sau ieşire)

Un obiect poate apărea de mai multe ori în cadrul aceleiaşi diagrame de activități.

Fiecare apariție indică un alt punct (stare) în viața obiectului. Pentru a distinge aparițiile numele stării obiectului poate fi adăugat la sfârşitul numelui obiectului .

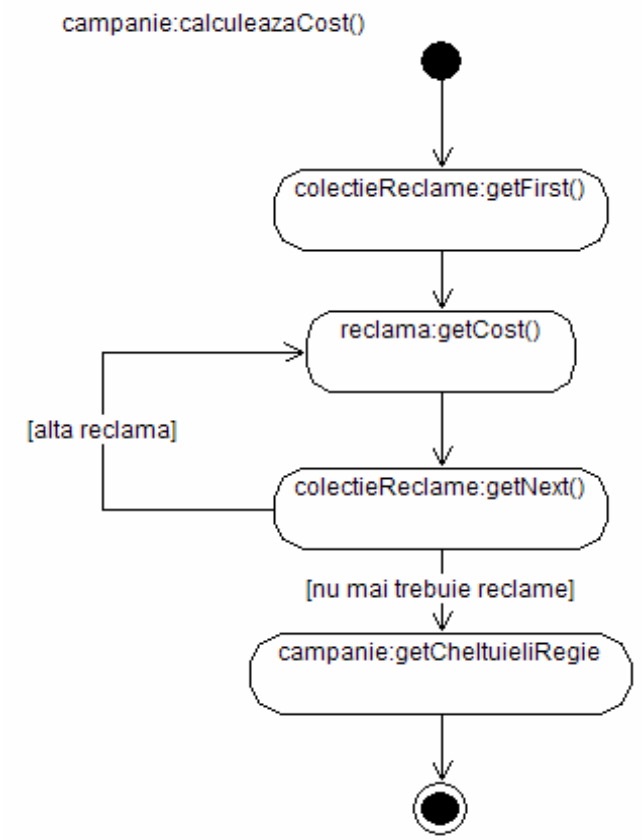


Figura 4.9. Diagrama de activități cu operația obiectului ca activitate

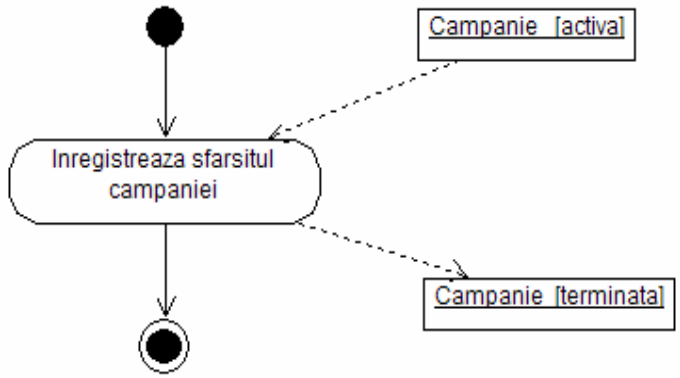


Figura 4.10. Diagrama de activități cu fluxuri de obiecte

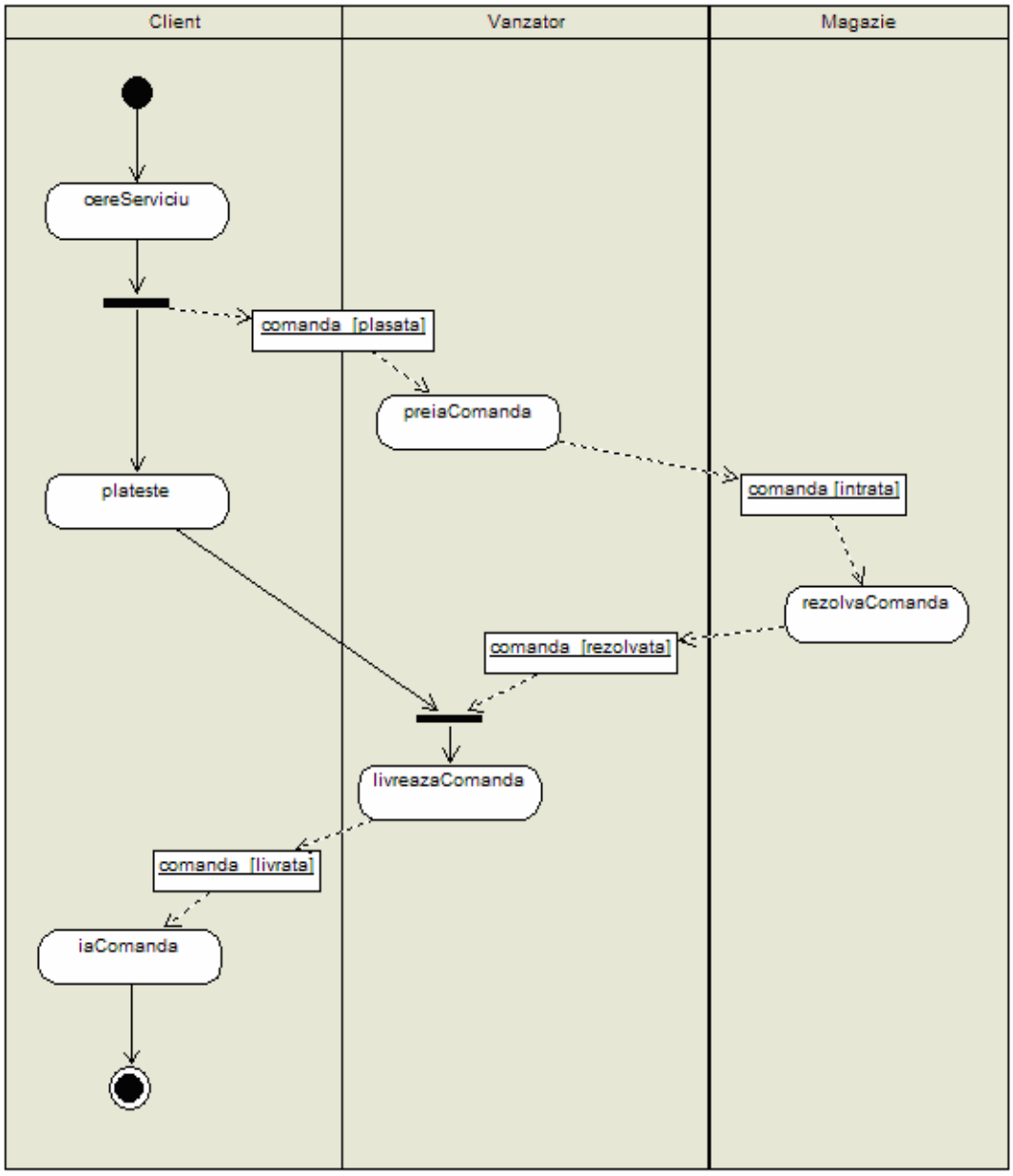


Figura 4.11. Diagramă de activități cu obiecte şi „swimlanes”

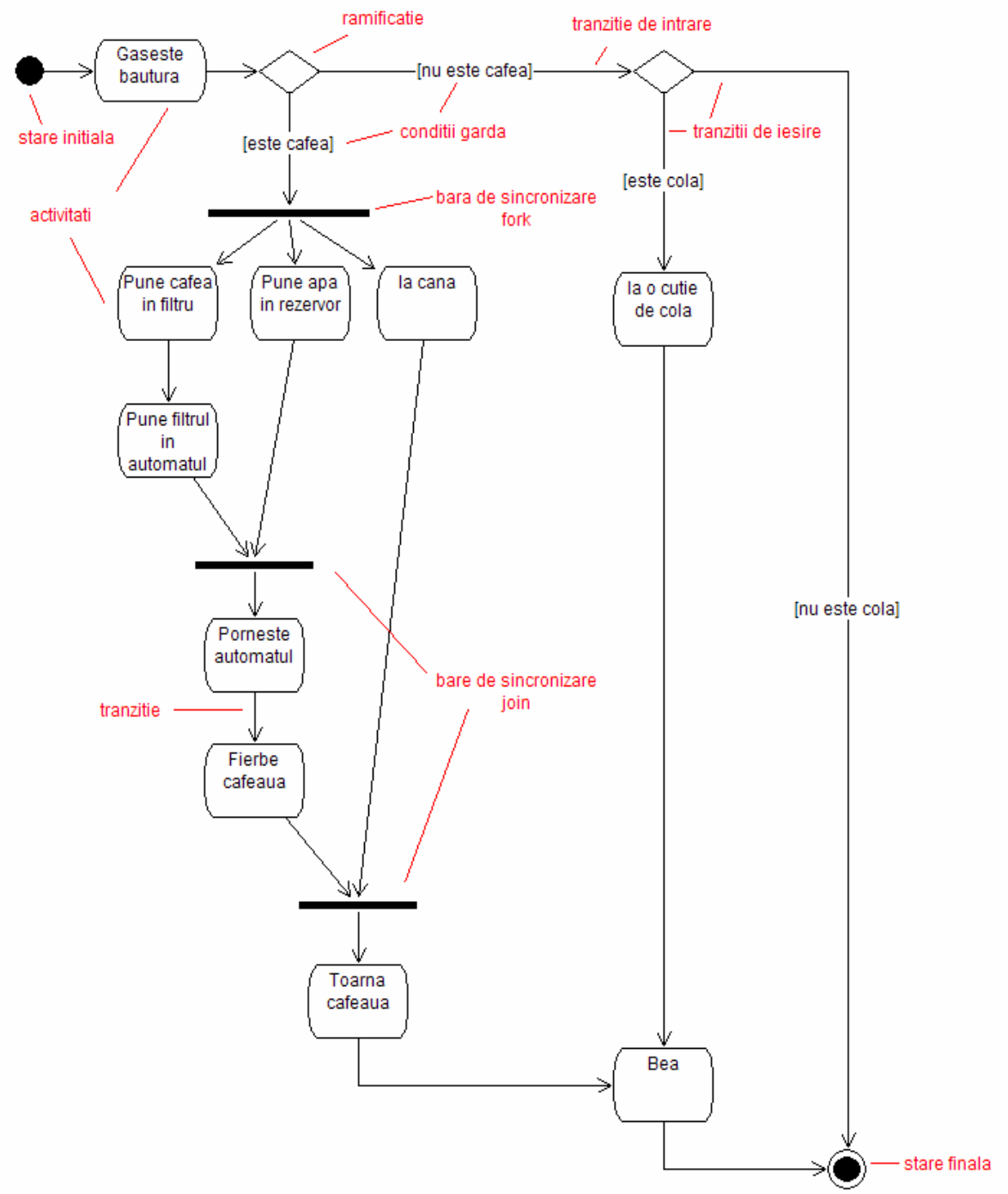


Figura 4.12. Exemplu de diagramă de activități pentru un automat de cafea

5. Diagrame de Interacțiuni

Diagramele de interacțiuni sunt folosite pentru a modela comportamentul unei mulțimi de obiecte dintr-un anumit *context* care interacționează în vederea îndeplinirii unui anumit *scop*.

Scopul specifică modul în care se realizează o operație sau un caz de utilizare.

Contextul unei interacțiuni (unde pot găsi interacțiuni) poate fi:

- *sistem* / *un subsistem* (uzual) - mulțimea obiectelor din sistem care colaborează între ele;
- *operație* - interacțiuni între parametri, variabile locale şi globale;
- *clasă* - interacțiuni între atributele unei clase (cum colaborează ele), interacțiuni cu obiecte globale, sau cu parametrii unei operații.

Obiectele care participă la o interacțiune pot fi lucruri concrete sau prototipuri. De obicei, într-o colaborare obiectele reprezintă prototipuri ce joacă diferite roluri, şi nu obiecte specifice din lumea reală.

Între obiectele care participă la o colaborare se pot stabili legături.

O ***legătură*** (link) reprezintă o conexiune semantică între obiecte. Ea este o instanță a unei asocieri şi poate avea toate atributele specifice asocierii (nume, roluri, navigare, agregare), dar nu şi multiplicitate.

Obiectele care interacționează comunică între ele, comunicarea făcându-se prin schimb de mesaje.

Un ***mesaj*** specifică o comunicare între obiecte. El poartă o informație şi este urmat de o activitate. Primirea unei instanțe a unui mesaj poate fi considerată o instanță a unui eveniment.

Unui mesaj îi este asociată o *acțiune* care poate avea ca efect schimbarea stării actuale a obiectului.

Forma generală a unui mesaj este

```
[cond_gardă] acțiune (lista_parametrilor)
```

unde

condiție_gardă – condiție booleană care se evaluează la fiecare apariție a mesajului specificat; acțiunea se execută doar când rezultatul evaluării este `true;`

Tipuri de acțiuni existente în UML:

- ***call***: invocă o operație a unui obiect. Un obiect îşi poate trimite lui însuşi un mesaj (invocare locală a unei operații). Este cel mai comun tip de mesaj. Operația apelată trebuie să fie definită de obiectul apelat şi vizibilă apelantului.
- ***return***: returnează o valoare apelantului.
- ***send***: trimite un semnal unui obiect.
- ***create***: creează un obiect.
- ***destroy***: distruge un obiect. Un obiect se poate autodistruge.

O diagramă de interacțiuni constă dintr-o mulțime de obiecte şi relațiile dintre ele (inclusiv mesajele care se transmit). Există două tipuri de diagrame de interacțiuni:

- *diagrama de secvență;*
- *diagrama de colaborare*.

Cele două diagrame specifică aceeaşi informaţie, însă pun accentul pe aspecte diferite.

5.1 Diagrama de secvenţă

Diagrama de secvenţă pune accentul pe aspectul temporal (ordonarea în timp a mesajelor).

Notaţia grafică este un ***tabel*** (figurile 5.1, 5.2) care are pe axa X *obiecte*, iar pe axa Y *mesaje* ordonate crescător în timp.

Axa Y arată pentru fiecare obiect:

- linia vieţii - linie punctată verticală;
- perioada în care obiectul preia controlul execuţiei - reprezentată printr-un dreptunghi subţire pe linia vieţii; în această perioadă obiectul efectuează o acţiune, direct sau prin intermediul procedurilor subordonate.

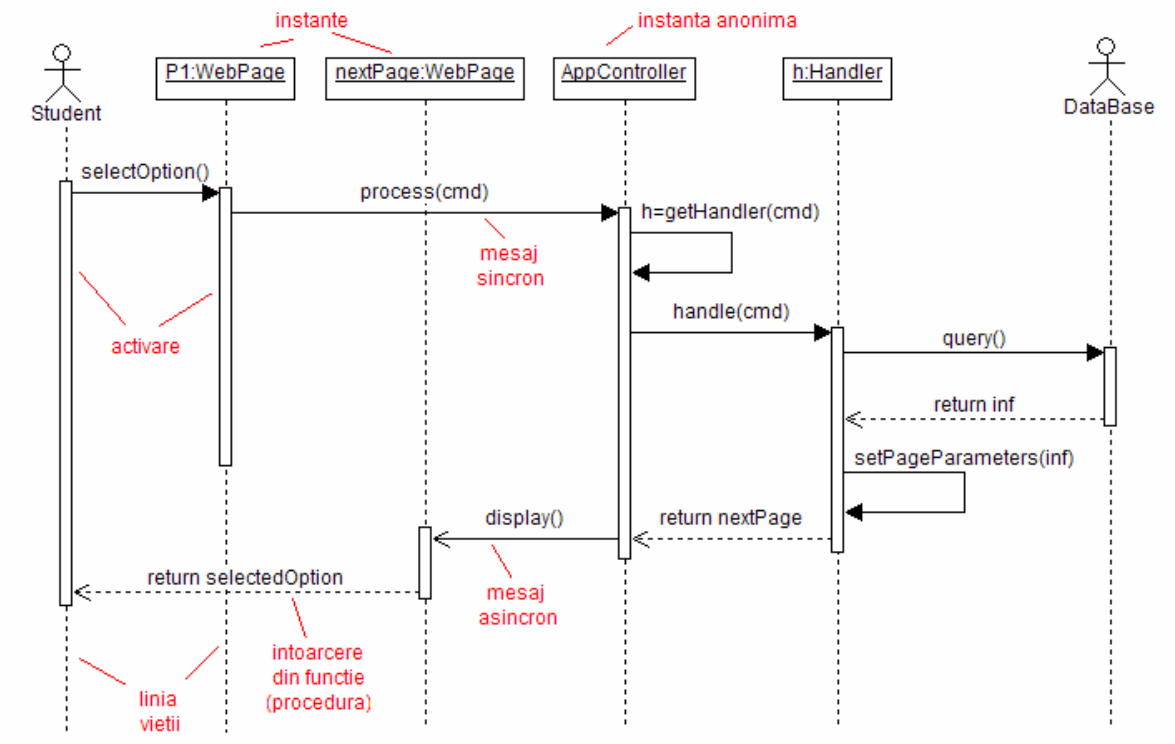


Figura 5.1. Exemplu de diagramă de secvenţă

Notaţia grafică pentru mesaje

A ——▶ B **Comunicare sincronă.** Controlul execuţiei trece de la A la B şi revine la A după ce B îşi termină execuţia. *Exemplu*: apel de funcţie.

A ——> B **Comunicare asincronă**. A trimite un semnal după care îşi continuă execuţia mai departe. *Exemplu*: aruncarea unei excepţii.

<---------- **Întoarcere dintr-o funcţie** (**procedură**). În cazul în care este omisă se consideră implicit că revenirea se face la sfârşitul activării.

În figura 5.2 este prezentat un exemplu de diagramă de secvenţă. Mesajul `extrageNume()` este primul mesaj recepţionat de `Client` şi corespunde cererii `Managerului de Campanie` de a furniza numele clientului selectat. Obiectul Client

recepționează apoi mesajul `listeazăcampanii()` şi începe a doua perioadă de activare. Obiectul `Client` trimite apoi mesajul `extrageDetaliiCampanie()` fiecărui obiect `Campanie` pe rând pentru a construi o listă a campaniilor. Această operațiune repetată se numeşte ***iterație*** şi se marchează prin caracterul „*” înaintea numelui mesajului. Condiția de continuare sau de încetare poate fi precizată în interiorul numelui mesajului. Condiția de continuare poate fi scrisă sub forma:

```
[pentru toți clienții campaniilor] *extrageDetaliiCampanie()
```

`Manager Campanie` trimite apoi un mesaj unui obiect particular `Campanie` pentru a obține o listă a reclamelor. Obiectul `Campanie` deleagă responsabilitatea pentru a extrage titlul reclamei fiecărui obiect `Reclamă` deşi obiectul `Campanie` păstrează responsabilitatea pentru lista reclamelor (fapt indicat de păstrarea activării şi după ce este trimis mesajul).

Când o reclamă este adăugată la campanie este creat un obiect `Reclama`. Această creare este indicată de mesajul `Reclama()` (care invocă un constructor).

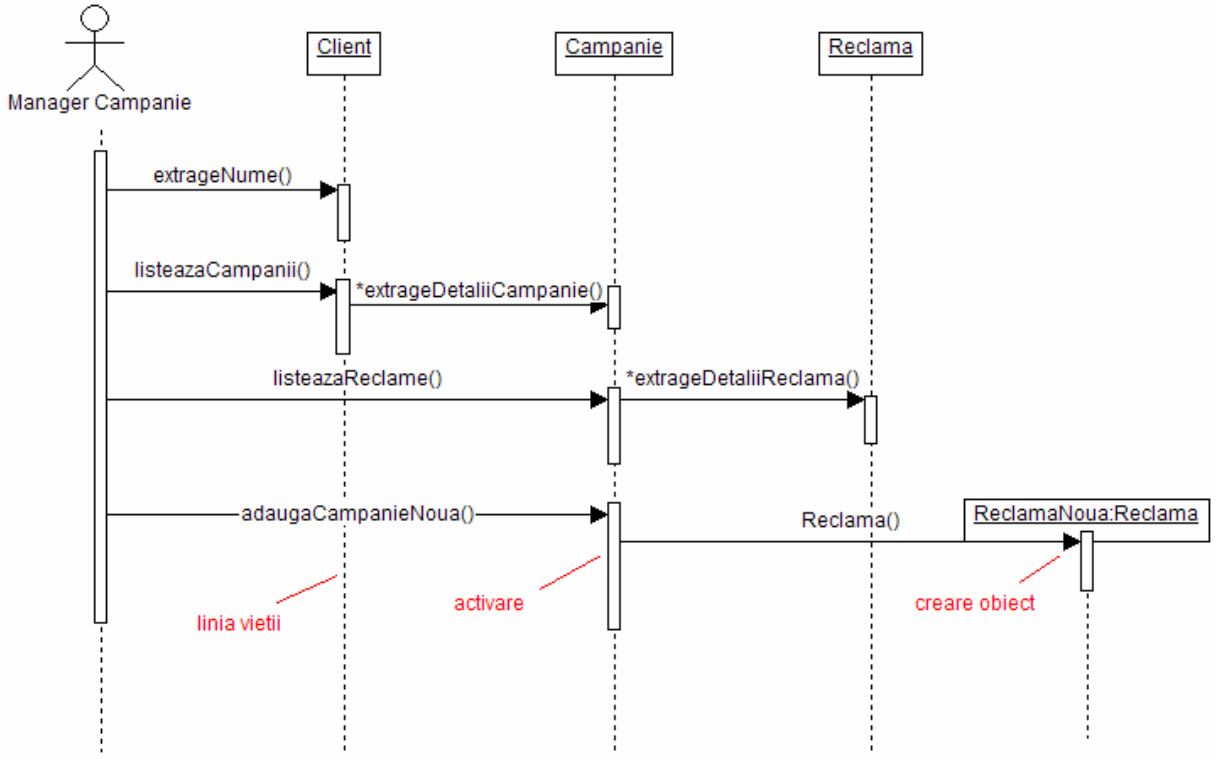


Figura 5.2. Exemplu de diagramă de secvență care
modelează adăugarea unei noi reclame unei campanii

Obiectele pot fi create sau distruse la diferite momente ale interacțiunii. Distrugerea unui obiect este marcată printr-un “X” pe linia vietii. Un obiect poate fi distrus când primeşte un mesaj (ca în figura 5.3) sau se poate distruge singur la sfârşitul unei activări.

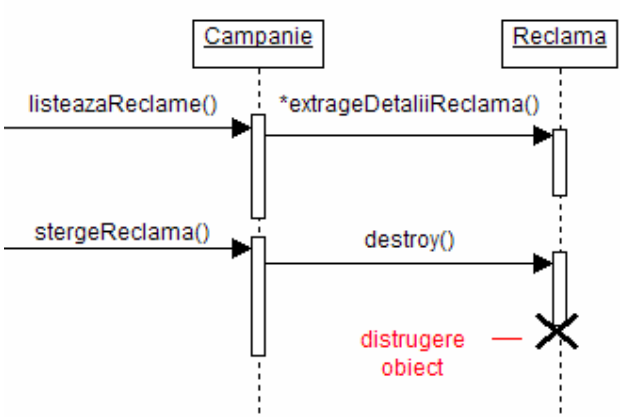


Figura 5.3. Distrugerea unui obiect

Un obiect îşi poate trimite un mesaj lui însuşi. Acest mesaj este cunoscut sub numele de mesaj reflexiv şi este reprezentat de o săgeată care pleacă şi se termină pe o activare a aceluiaşi obiect. Diagram de secvență din figura 5.4 include mesajul reflexiv `calculeazăCheltuieliregie`() trimis de obiectul `Campanie` lui însuşi.

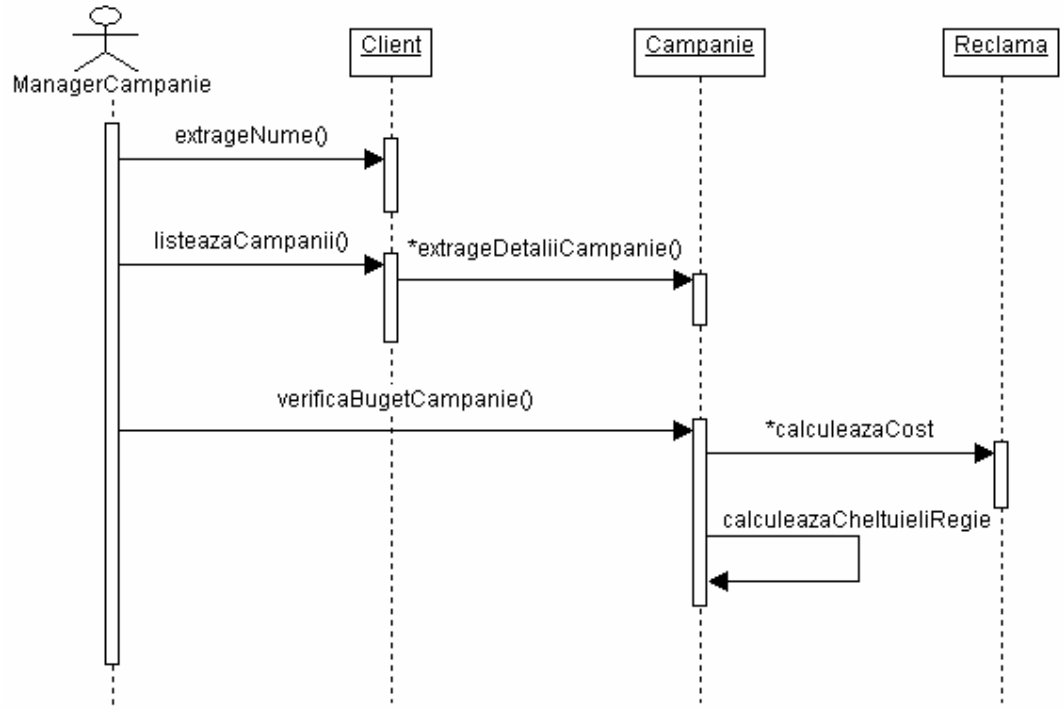


Figura 5.4. Diagramă de secvență care modelează `Verifică bugetul campaniei`

După cum am mai precizat, revenirea controlului la obiectul care a trimis un mesaj se poate marca explicit în diagrama de secvență printr-o săgeată trasată cu linie întreruptă (figura 5.5). Valoarea de revenire de obicei nu se prezintă într-o diagramă de secvență.

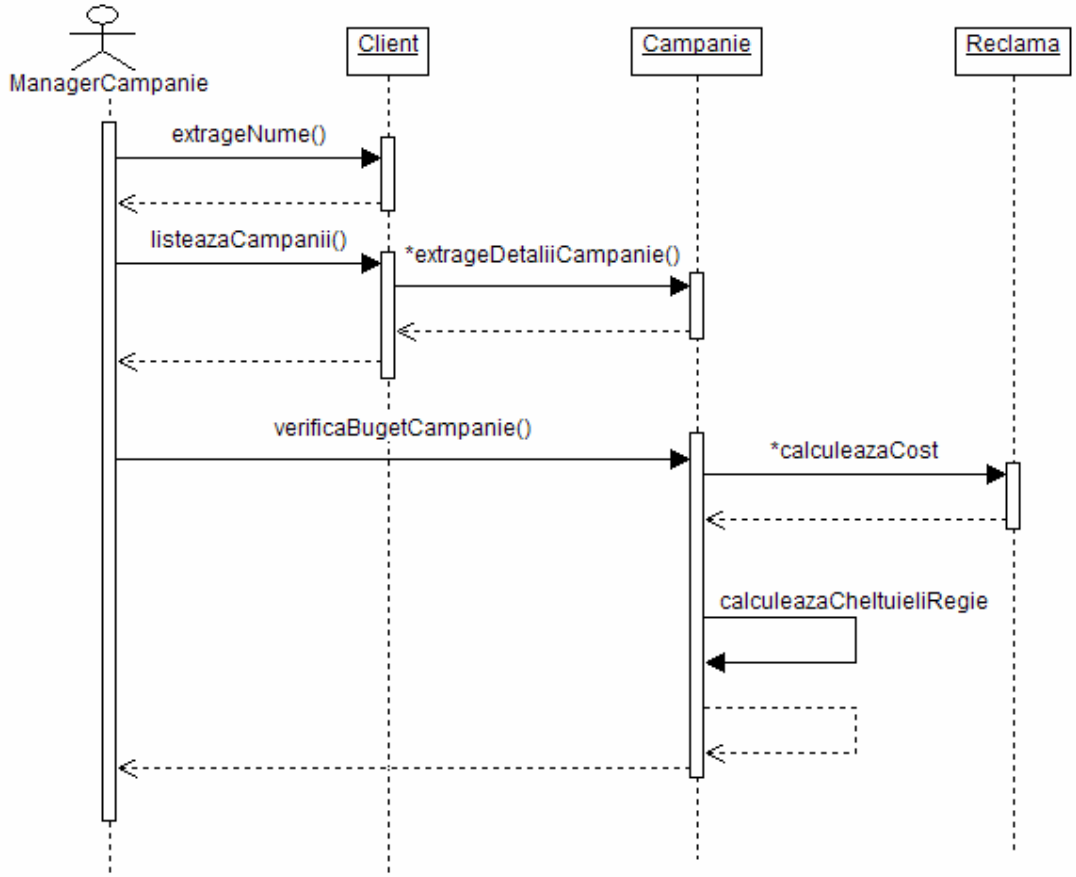


Figura 5.5. Diagramă de secvență care modelează `Verifică bugetul campaniei` cu marcarea explicită a revenirilor

Dacă *mesajele sincrone* (care invocă o operație) determină suspendarea execuției obiectului sursă până când destinatarul își termină execuția, *mesajele asincrone* nu necesită suspendarea execuției obiectului ce trimite mesajul. Mesajele asincrone sunt des folosite în aplicațiile de timp real în care operațiile diferitelor obiecte se execută în paralel, fie din motive de eficiență, fie deoarece sistemul simulează procese concurente. Este necesar ca o operație invocată să notifice obiectul care a invocat-o în momentul când își termină execuția. Această notificare se face printr-un mesaj explicit (numit *callback*).

Constrângeri de timp

O diagramă de secvență poate fi etichetată cu constrângeri de timp în moduri diferite. În figura 6 se asociază expresiile de timp cu numele mesajului astfel încât constrângerile de timp pot fi specificate pentru execuția unei operații sau transmisia unui mesaj. Spre exemplu funcția `a.sendTime` furnizează timpul la care mesajul `a` este trimis şi `d.receiveTime` furnizează timpul la care o instanță a clasei A primeşte mesajul `d`. Există construcții care pot fi utilizate pentru a marca un interval de timp – în figura 5.6 este marcat intervalul de timp scurs între recepția mesajului `b` şi trimiterea mesajului `c`. Constrângerile de timp sunt utilizate frecvent în modelarea sistemelor de timp real. Pentru alte tipuri de sisteme constrângerile de timp nu sunt semnificative.

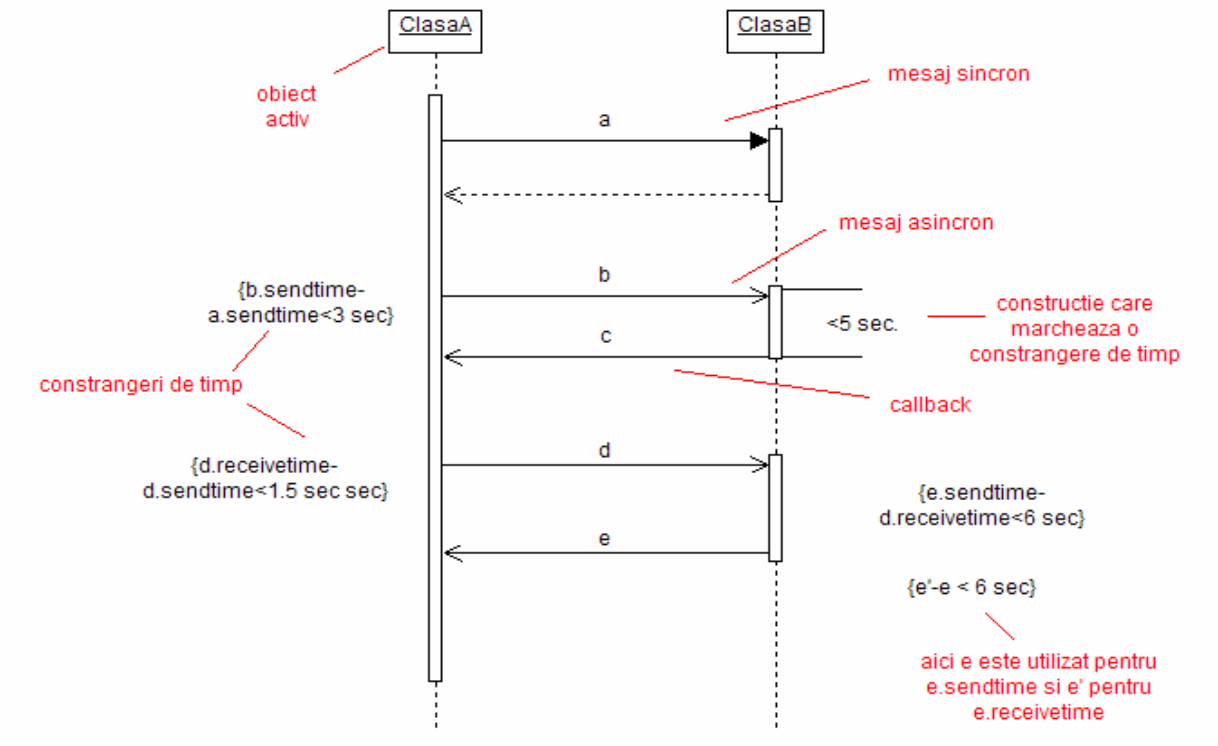


Figura 5.6. Diagramă de secvență cu tipuri diferite de mesaje şi constrângeri de timp

Ramificații

Diagramele de secvență permit reprezentările ramificațiilor prin mai multe săgeți care pleacă din acelaşi punct şi eventual sunt etichetate cu condiții.

În figura 5.7 este prezentat un exemplu de diagramă de secvență cu ramificații.

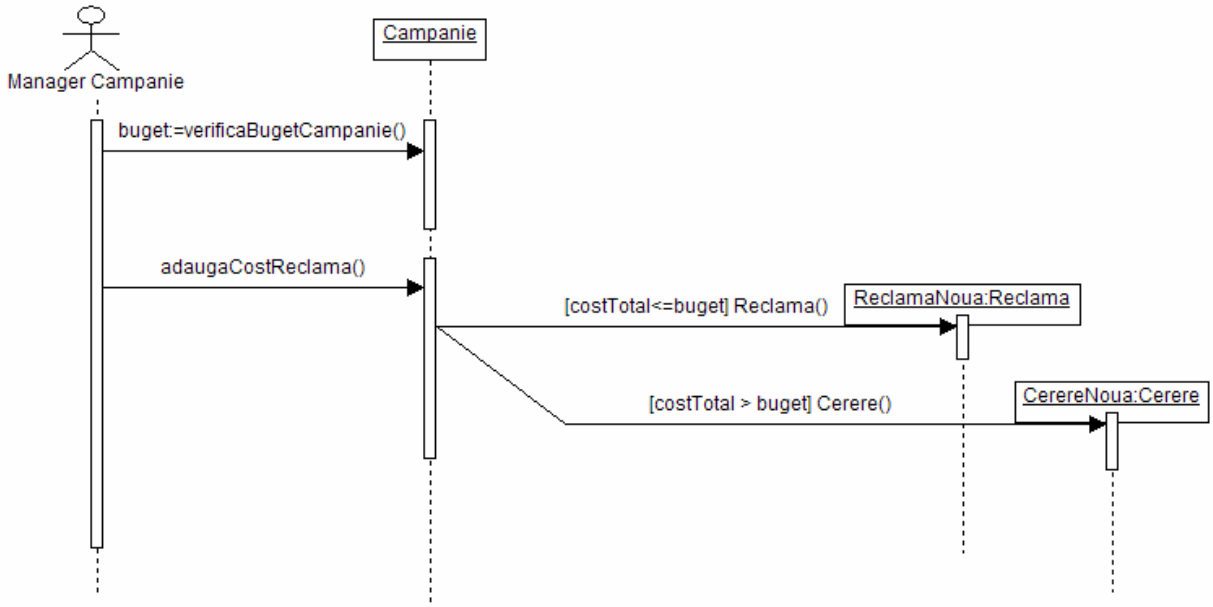


Figura 5.7. Exemplu de diagramă de secvențe cu ramificații.

5.2 Diagrama de colaborare

Diagrama de colaborare este o diagramă de interacţiuni care pune accentul pe organizarea structurală a obiectelor care participă la interacţiune.

Diagrama de colaborare poate conţine:

- obiecte;
- legături între obiecte;
- mesajele prin care obiectele comunică.

Diagramele de colaborare au multe asemănări cu diagramele de secvenţă, exprimând aceleaşi informaţii dar într-un alt format. Pot fi create la nivele diverse de detaliu şi în diferite stadii de dezvoltare a procesului software. Deoarece au un conţinut similar, pot fi folosite pentru generarea diagramelor de secvenţă şi viceversa.

Diferenţa semnificativă faţă de diagrama de secvenţă este aceea că diagrama de colaborare arată explicit legăturile dintre obiecte. De asemenea, la diagrama de colaborare timpul nu are o dimensiune explicită. Din acest motiv, ordinea în care sunt trimise mesajele este reprezentată prin numere de secvenţă.

Mesajele dintr-o diagramă de colaborare sunt reprezentate de un set de simboluri care sunt asemănătoare celor utilizate în diagrama de secvenţă, dar cu câteva elemente adiţionale pentru a marca secvenţierea şi recurenţa.

Notă. Termenii care vor apărea între paranteze pătrate sunt opţionali iar termenii care vor apărea între acolade pot să nu apară sau să apară de mai multe ori.

Sintaxa unui mesaj este următoarea:

```
    [predecesor] [condiţie_gardă] expresie_secvenţă
[valoare_întoarsă ':='] nume_mesaj '('[listă_argumente]')'
```

unde

- *predecesor* – listă a numerelor de secvenţă a mesajelor care trebuie trimise înainte de trimiterea mesajului curent (permite sincronizarea trimiterii mesajelor); permite specificarea detaliată a căilor ramificaţiilor.

  Sintaxa *predecesorului* este următoarea:

  ```
  număr_secvenţă { ',' număr_secvenţă } '/'
  ```

  '/' – marchează sfârşitul listei şi se include doar dacă este precizat explicit predecesorul.

- *condiţie_gardă* – expresie booleană care permite condiţionarea transmiterii mesajului (se scrie în OCL – Object Constraint Language) şi se poate utiliza pentru reprezentarea sincronizării diferitelor fluxuri de control.

- *expresie_secvenţă* – listă de întregi separaţi prin caracterul `'.'`, urmată opţional de un *nume* (o singură literă), un termen recurenţă şi terminată de caracterul `':'`.

  Sintaxa *expresiei_ secvenţă* este următoarea:

  ```
  întreg { '.' întreg } [nume] [recurenţă] ':'
  ```

  Se observă că numerotarea este asemănătoare celei utilizate la numerotarea capitolelor şi paragrafelor într-un document.

  întreg – precizează ordinea mesajului; poate fi folosit într-o construcţie de tip buclă sau ramificaţie.

Exemplu: mesajul 5.1.3 se transmite după 5.1.2 şi ambele se transmit după activarea mesajului 5.1.

nume – se foloseşte pentru a diferenția două mesaje concurente când acestea au acelaşi număr de secvență.

Exemplu: mesajele 3.2.1a şi 3.2.1b se transmit simultan în cadrul activării mesajului 3.2.

recurența – permite specificarea modului de transmitere a mesajelor:

- secvențial - `'*''['clauză_iterație']'`
- paralel - `'||''['clauză_iterație']'`
- ramificație - `'['clauză_condiție']'`

În tabelul următor se prezintă câteva exemple de tipuri de mesaje:

Tipuri de mesaje:	**Exemple**
mesaj simplu	`4: adaugă ReclamaNoua ()`
subapeluri cu valoarea întoarsă *Valoarea întoarsă este plasată în variabila nume*	`3.1.2: nume:= extrageNume ()`
mesaj condițional *mesajul este trimis doar dacă este adevărată condiția* `[balanță > 0]`	`[balanță > 0] 5: debit (sumă)`
sincronizare cu alte mesaje *mesajul* `4: playVideo` *este trimis doar după ce mesajele concurente* `3.1a` *şi* `3.1b` *sunt complete*	`3.1a, 3.1b / 4: playVideo()`
iterații	`[i = 1..n] update ()`

În figura 5.8 este prezentată diagrama de colaborare corespunzătoare diagramei de secvență din figura 5.1.

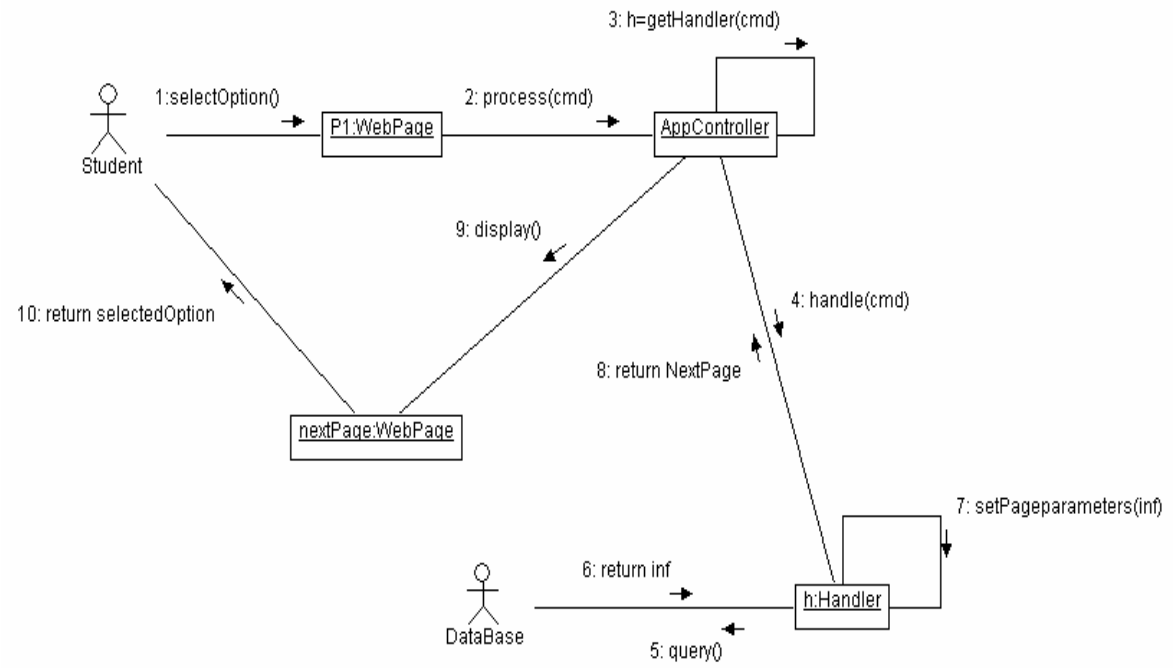


Figura 5.8. Exemplu de diagramă de colaborare

Există mai multe posibilități de interacțiune pentru un use case particular. Acestea se datorează alocărilor posibile diferite ale responsabilităților. Spre exemplu, interacțiunile din figura 5.9 pot avea trăsături nedorite. Mesajul `extrageDetaliiCampanie` trimis de `Client` obiectului `Campanie` necesită ca obiectul `Client` să returneze aceste detalii obiectului `AdaugaReclama`. Dacă detaliile despre campanie includ doar numele campaniei atunci un volum relativ mic de date este pasat de la `Campanie` la `Client` şi apoi la `AdaugaReclama`. Acest fapt poate fi acceptabil.

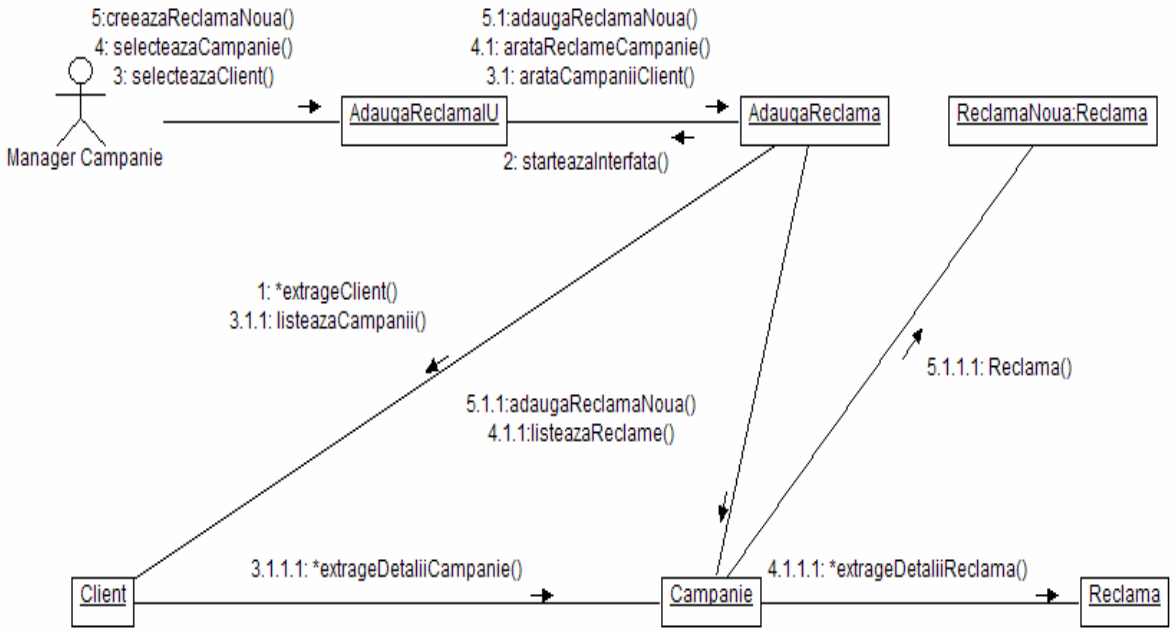


Figura 5.9. Diagrama de colaborare pentru use case-ul
`Adaugă o reclamă nouă unei campanii`

Pe de altă parte, dacă detaliile despre campanie includ data de start, de terminare şi bugetul campaniei, atunci prin `Client` se pasează mai mult de o dată. În acest caz obiectul `Client` este acum responsabil pentru furnizarea unor date semnificative pentru campanii în loc de obiectul `Campanie`. S-ar putea deci transfera date direct de la `Campanie` la `AdaugaReclama`. Această posibilitate este prezentată în figura 5.10, în care `AdaugaReclama` preia responsabilitatea de a extrage detalii despre campanie direct de la obiectul `Campanie`. În această interacțiune, obiectul `Client` este responsabil doar de furnizarea unei liste de campanii obiectului `AdaugăReclamă`.

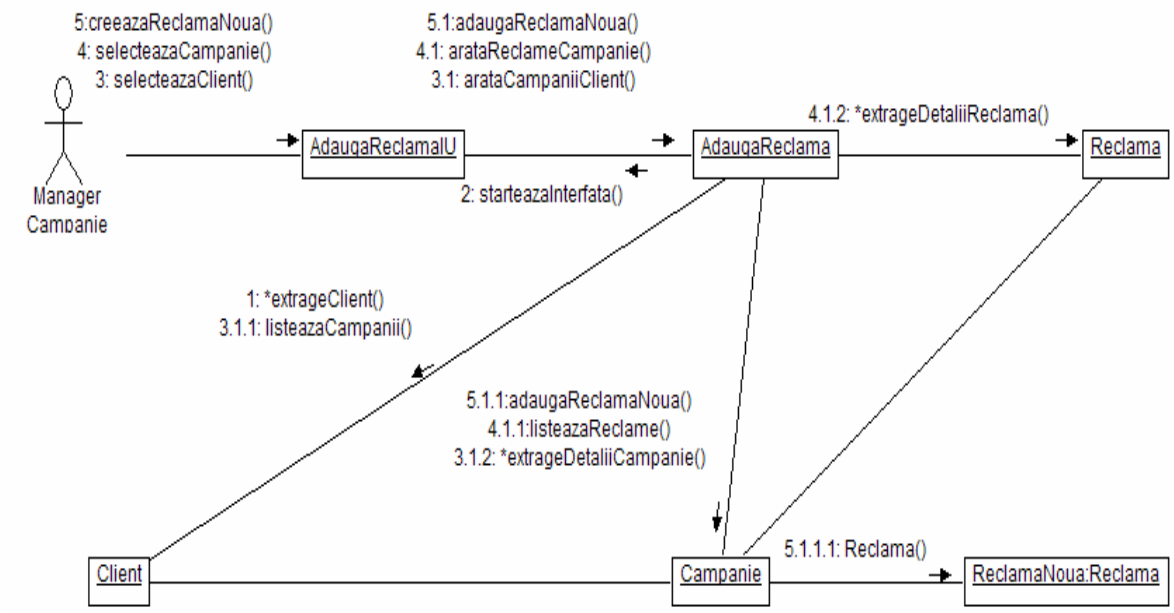


Figura 5.10. Diagramă de colaborare alternativă pentru use case-ul `Adaugă o reclamă nouă unei campanii`

În figura 5.11 este prezentată o diagramă de colaborare care prezintă interacțiunile pentru o singură operație – `verificăBugetCampanie()` – care este una din operațiile din diagramele din figurile 5.9 şi 5.10.

Diagramele de colaborare sunt preferate de unii dezvoltatori diagramelor de secvență deoarece interacțiunile între obiecte pot fi translate uşor în diagramele de clase datorită vizibilității legăturilor între obiecte.

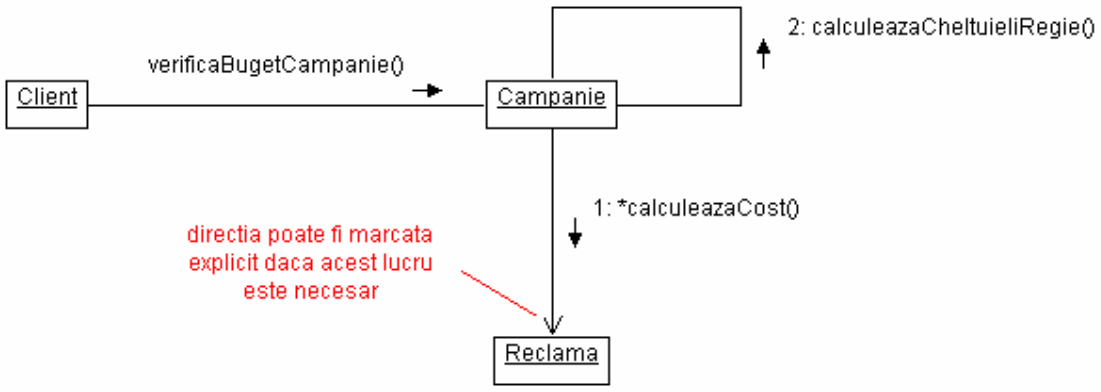


Figura 5.11. Diagrama de colaborare pentru operația `verificaBugetCampanie()`

6. Organizarea modelelor în pachete

Pachetul (package) este o grupare de elemente ale unui model (use case-uri, clase etc.) şi reprezintă baza necesară controlului configuraţiei, depozitării şi accesului. Este un container logic pentru elemente între care se stabilesc legături.

Pachetul defineşte un spaţiu de nume.

Toate elementele UML pot fi grupate în pachete (cel mai des pachetele sunt folosite pentru a grupa clase). Un element poate fi conţinut într-un singur pachet.

Un pachet poate conţine subpachete, deci se creează o structură arborescentă (similară cu organizarea fişierelor/directoarelor).

Notaţia grafică pentru pachet este prezentată în figura 6.1.

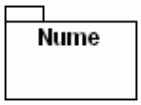


Figura 6.1. Notaţia grafică pentru pachet

Pachetele pot face referire la alte pachete, iar modelarea se face folosind unul din stereotipurile <<import>> (import public) şi <<acces>> (import privat) asociate unei relaţii de dependenţă (vezi figura 6.2).

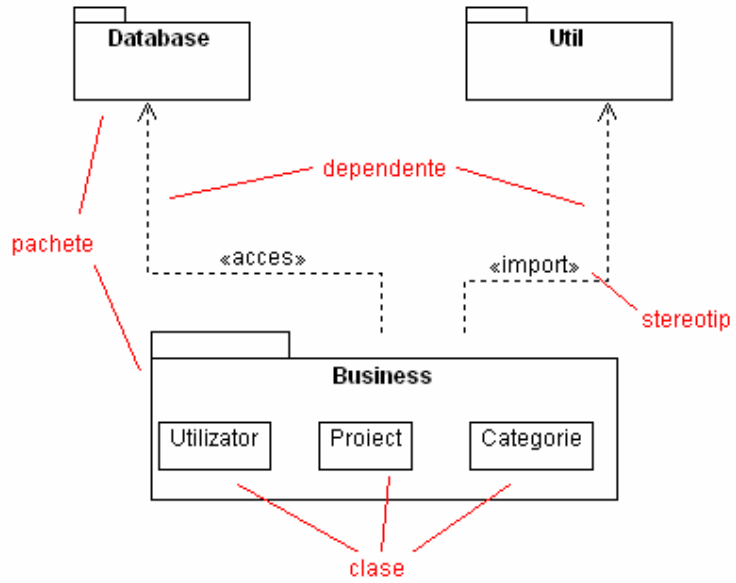


Figura 6.2. Exemple de relaţii care se pot stabili între pachete

Ambele tipuri de relaţii permit folosirea elementelor aflate în pachetul destinaţie de către elementele aflate în pachetul sursă fără a fi necesară calificarea numelor elementelor din pachetul destinaţie.

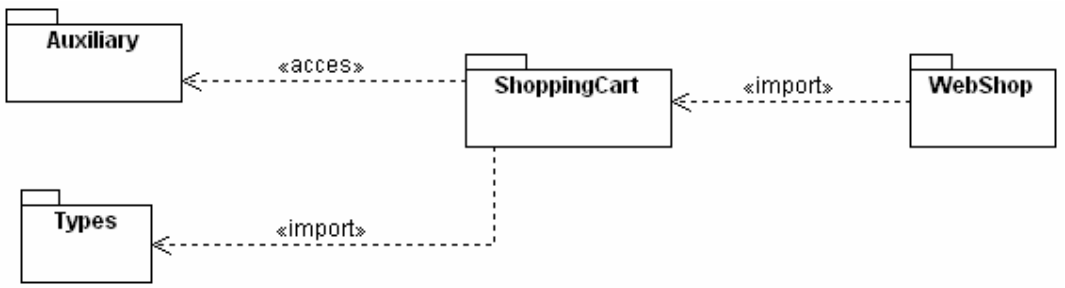


Figura 6. 3. Exemplu de diagramă de pachete

În figura 6.3 elementele din `Types` sunt importate în `ShoppingCart` şi apoi sunt importate mai departe de către `WebShop`. Elementele din `Auxiliary` pot fi accesate însă doar din `ShoppingCart` şi nu pot fi referite folosind nume necalificate din `WebShop`.

Utilitatea pachetelor. Pachetele împart sistemele mari în subsisteme mai mici şi mai uşor de gestionat. De asemenea permit dezvoltare paralelă iterativă şi definirea unor interfețe clare între pachete promovează refolosirea codului (ex. pachet care oferă funcții grafice).

7. Diagrame de implementare

Diagrama de componente

Diagrama de componente este o diagramă de implementare care modelează dependențele dintre componentele software ale sistemului şi entitățile care le implementează (fişiere cod sursă, cod binar, executabile, scripturi etc.).

Într-un proiect de dimensiune mare, vor exista multe fişiere care realizează sistemul. Aceste fişiere depind unele de altele. Natura acestor dependențe e dată de limbajul (limbajelor) folosite pentru dezvoltarea proiectului. Dependențele pot exista în momentul compilării, link-editării sau rulării. Există de asemenea dependențe între fişiere sursă şi cele executabile sau obiect, rezultate din primele prin compilare.

În figura 7.1 este prezentată o diagramă de componente care descrie dependențele dintre o sursă scrisă în C++ şi fişierul header asociat, dependența fişierului obiect de cele două fişiere anterioare şi dependența fişierului executabil de fişierul obiect. Se pot adăuga şi stereotipuri care se folosesc pentru a arăta tipurile diferitelor componente.

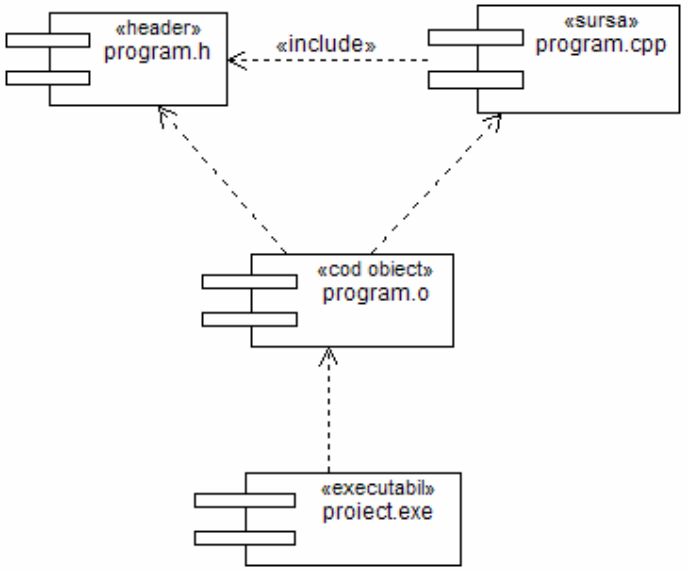


Figura 7.1. Diagramă de componente care arată dependențele în C++

O alternativă de reprezentare a unei părți a diagramei de componente este de a utiliza notația pentru interfață în UML pentru a arăta specificația unei clase (fişierul header în C++) ca o interfață şi corpul ca o componentă (vezi figura 7.2).

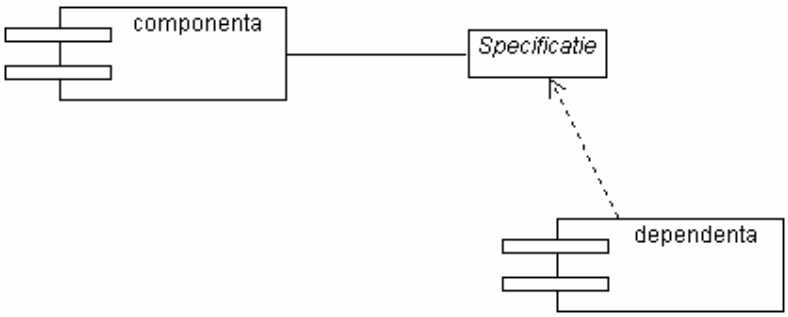


Figura 7.2. Dependența unei componente printr-o interfață cu o altă componentă

Observații.

- Componentele unei diagrame de componente pot fi componente fizice ale sistemului.
- Diagramele de componente sunt utilizate în general pentru a marca dependențele la scară mare între componentele unui sistem (vezi figura 7.3).

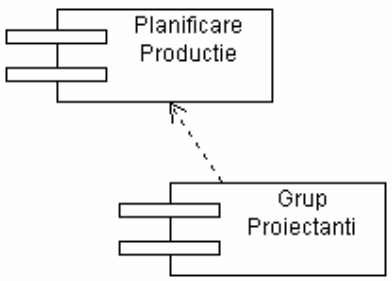


Figura 7.3. Exemplu de dependență la scară mare între componente unui sistem

- Obiectele active care rulează pe fire de execuție separate pot fi prezentate în diagrama de componente (vezi figura 7.4)

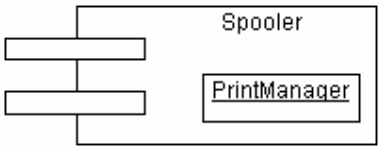


Figura 7.4. Exemplu de obiect activ în interiorul unei componente

- În timpul analizei şi la începutul proiectării, se folosesc diagramele de pachete pentru a arăta gruparea logică a diagramelor de clase (sau a modelelor care utilizează alte tipuri de diagrame) în pachete referitoare la subsisteme.
- În timpul implementării, diagramele de pachete pot fi folosite a arăta gruparea componentelor fizice în subsisteme.
- Diagrama de componente poate fi combinată cu diagrama de plasare pentru a arăta localizarea fizică a componentelor sistemului. Clasele dintr-un pachet logic pot fi distribuite peste locațiile fizice dintr-un sistem fizic, iar diagramele de componente şi plasare arată tocmai acest lucru.

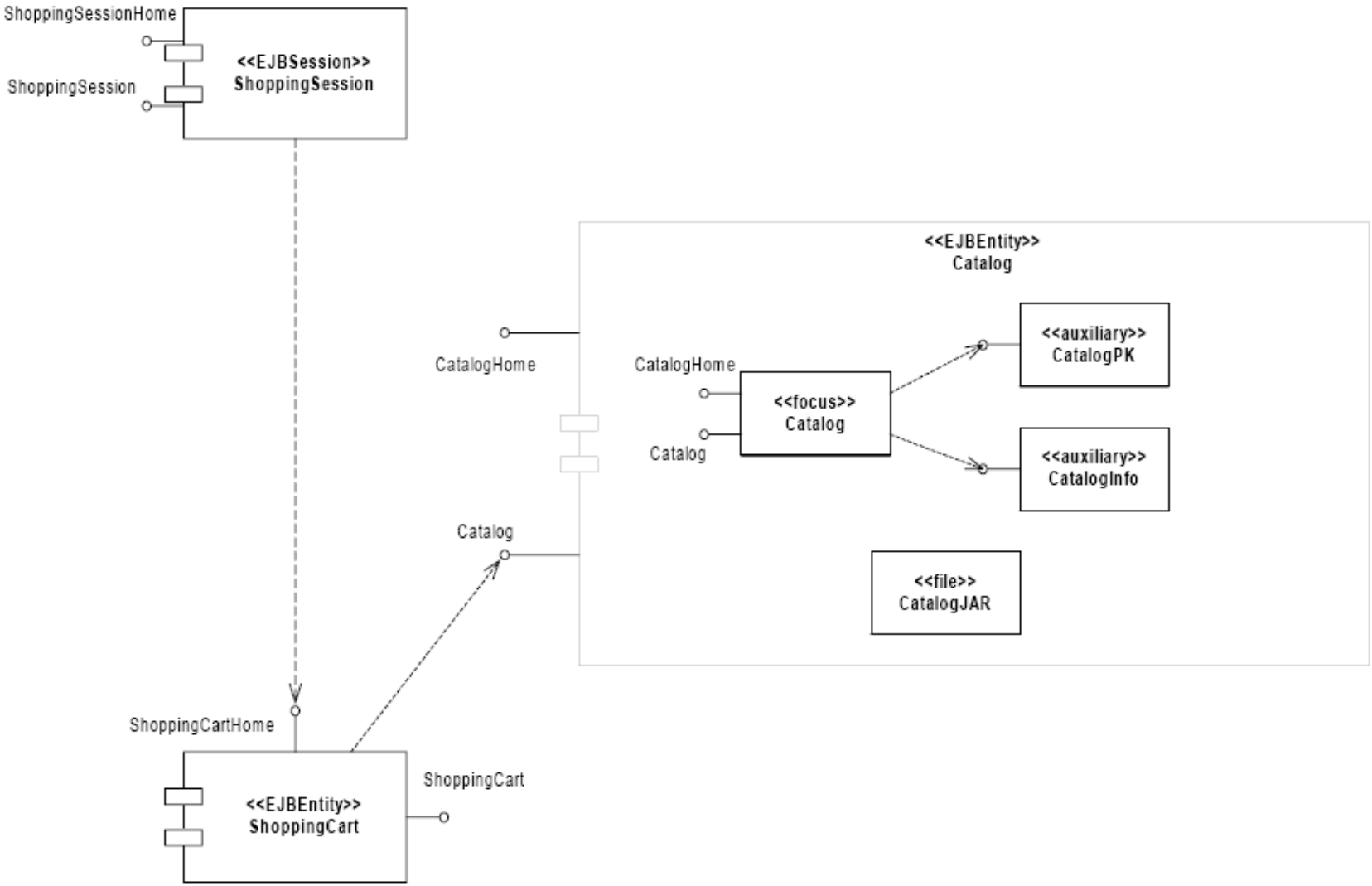


Figura 7.5. Exemplu de diagramă de componente

Diagrama de plasare

Diagrama de plasare arată configurația procesării elementelor care execută direct activități specifice şi componentele de program, procesele, obiectele care determină funcționarea acestor componente.

Componentele care nu au rol direct în execuție nu sunt arătate în această diagramă.

Diagrama de plasare este alcătuită din:

- noduri;
- Asocieri.

Nodurile arată computerele iar asocierile marchează rețeaua şi protocoalele care sunt folosite pentru a comunica între noduri (se modelează sistemele client / server din punct de vedere al topologiei).

Nodurile mai pot fi utilizate pentru a modela şi alte resurse cum ar fi personalul uman şi resursele mecanice.

Diagramele de plasare modelează arhitectura fizică a sistemului.

Notația grafică pentru noduri este prezentată în figura 7.6.

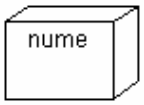


Figura 7.6. Notația grafică pentru nodurile unei diagrame de plasare.

Diagramele de plasare pot arăta fie tipuri de maşini fie instanțe particulare ca în figura 7.7.

În figura 7.8 se prezintă locul `bazei de date Vanzari` (pe server) şi câteva componente ale calculatoarelor clienților.

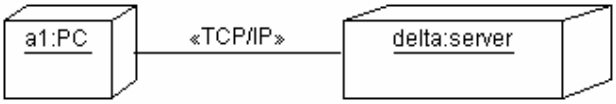


Figura 7.7. Exemplu de diagramă de plasare

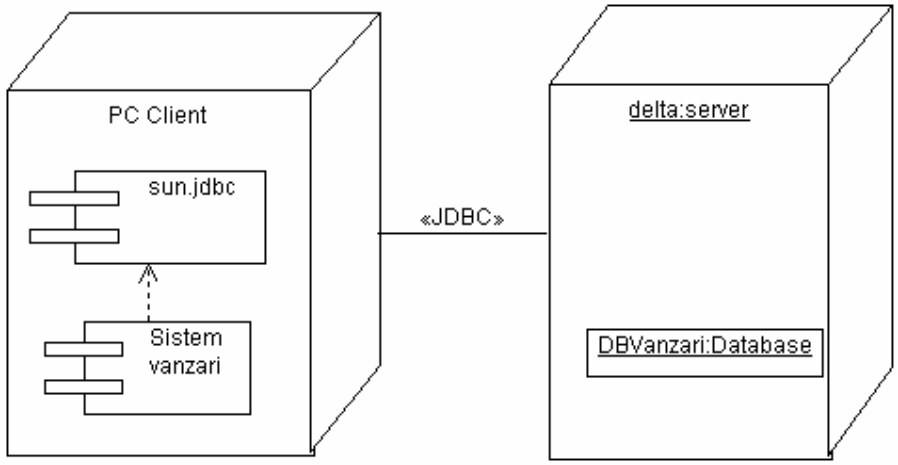


Figura 7.8. Diagramă de plasare cu componente ale `PC Client`
şi cu un obiect activ pe server

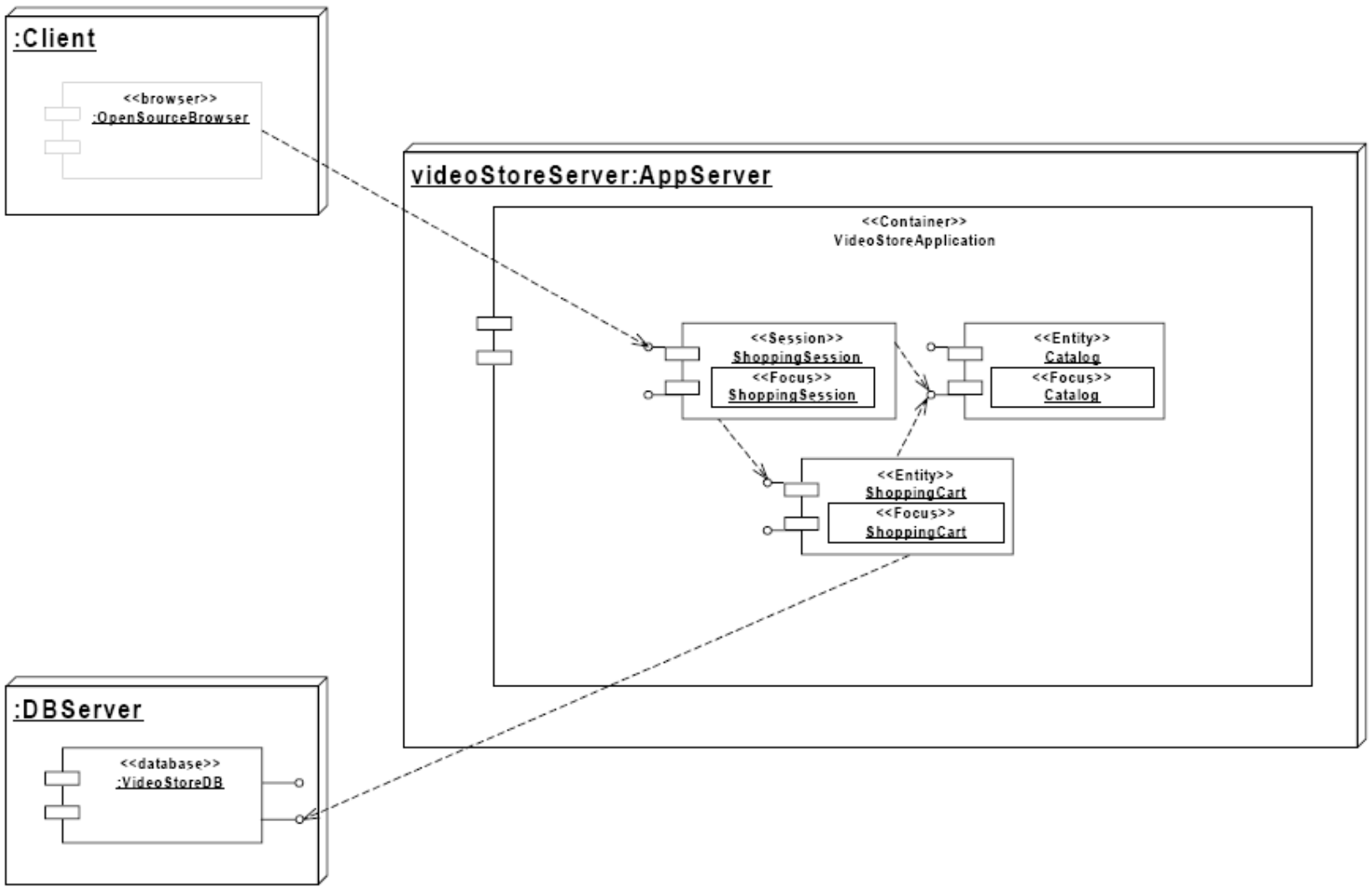


Figura 7.9. Exemplu de diagramă de plasare