Manual de Informatică pentru licenţă iunie și septembrie 2016

Specializarea Informatică

Tematica generală:

Partea 1. Algoritmică şi programare

1. Căutari (secvenţială şi binară), sortări (selecţie, bubblesort, quicksort). Metoda backtracking.
2. Concepte OOP în limbaje de programare (Python, C++, Java, C#): Clase şi obiecte. Membrii unei clase şi specificatorii de acces. Constructori şi destructori.
3. Relaţii între clase. Clase derivate şi relaţia de moştenire. Suprascrierea metodelor. Polimorfism. Legare dinamica. Clase abstracte şi interfeţe.
4. Diagrame de clase si de interacțiune între obiecte în UML: Pachete, clase şi interfețe. Relații între clase şi interfețe. Obiecte. Mesaje
5. Liste. Dicţionare. Specificarea operaţiilor caracteristice (fără implementări)
6. Identificarea structurilor şi tipurilor de date potrivite pentru rezolvarea problemelor (doar dintre cele de la punctul 5.). Folosirea unor biblioteci existente pentru aceste structuri (Python, Java, C++, C#).

Partea 2. Baze de date

1. Baze de date relaţionale. Primele trei forme normale ale unei relaţii.
2. Interogarea bazelor de date cu operatori din algebra relaţională.
3. Interogarea bazelor de date relaţionale cu SQL (Select).

Partea 3. Sisteme de operare

1. Structura sistemelor de fișiere Unix.
2. Procese Unix: creare, funcţiile fork, exec, exit, wait; comunicare prin pipe şi FIFO.
3. Programare shell Unix și comenzi Unix de bază: cat, cp, cut, echo, expr, file, find, grep, less, ls, mkdir, mv, ps, pwd, read, rm, sort, test, wc, who.

Cuprins

1. Algoritmică şi programare

1.1. Căutări şi sortări

1.1.1. Căutări

Datele se află în memoria internă, într-un şir de articole. Vom căuta un articol după un câmp al acestuia pe care îl vom considera cheie de căutare. În urma procesului de căutare va rezulta poziţia elementului căutat (dacă acesta există).

Notând cu k_1, k_2, ..., k_n cheile corespunzătoare articolelor şi cu a cheia pe care o căutăm, problema revine la a găsi (dacă există) poziţia p cu proprietatea $a = k_p$.

De obicei articolele sunt păstrate în ordinea crescătoare a cheilor, deci vom presupune că

$$k_1 < k_2 < .... < k_n .$$

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci şi să găsim în caz contrar locul în care ar trebui inserat un nou articol având cheia specificată, astfel încât să se păstreze ordinea existentă.

Deci *problema căutării* are următoarea specificare:

Date $a,n,(k_i, i=1,n);$

Precondiţia: $n \in N, n \geq 1$ *şi* $k_1 < k_2 < .... < k_n$;

Rezultate $p;$

Postcondiţia: $(p=1$ *şi* $a \leq k_1)$ *sau* $(p=n+1$ *şi* $a > k_n)$ *sau* $(1<p\leq n)$ *şi* $(k_{p-1} < a \leq k_p)$.

1.1.1.1. Căutare secvenţială

O primă metodă este căutarea **secvenţială**, în care sunt examinate succesiv toate cheile. Sunt deosebite trei cazuri: $a \leq k_1$, $a > k_n$, respectiv $k_1 < a \leq k_n$, căutarea având loc în al treilea caz.

```
Subalgoritmul CautSecv(a, n, K, p) este:          {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută p astfel ca: (p=1 şi a ≤ k1) sau}
                                                  { (p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp).
  Fie p := 0;                                                    {Cazul "încă negasit"}
  Dacă a ≤ k1 atunci p := 1 altfel
     Dacă a > kn atunci p := n + 1 altfel
          Pentru i := 2; n execută
               Dacă (p = 0) şi (a ≤ ki) atunci p := i sfdacă
          sfpentru
```

```
      sfdacă
   sfdacă
sf-CautSecv
```

Se observă că prin această metodă se vor executa în cel mai nefavorabil caz *n*-1 comparări, întrucât contorul *i* va lua toate valorile de la 2 la *n*. Cele *n* chei împart axa reală în *n*+1 intervale. Tot atâtea comparări se vor efectua în *n*-1 din cele n+1 intervale în care se poate afla cheia căutată, deci complexitatea medie are acelaşi ordin de mărime ca şi complexitatea în cel mai rău caz.

Evident că în multe situaţii acest algoritm face calcule inutile. Atunci când a fost deja găsită cheia dorită este inutil a parcurge ciclul pentru celelalte valori ale lui *i*. Cu alte cuvinte este posibil să înlocuim ciclul **PENTRU** cu un ciclu **CÂTTIMP**. Ajungem la un al doilea algoritm, dat în continuare.

```
Subalgoritmul CautSucc(a, n, K, p) este:          {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută p astfel ca: p=1 şi a ≤ k1) sau }
                                                  {(p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp).
   Fie p:=1;
   Dacă a>k1 atunci
      Câttimp p≤n şi a>kp execută p:=p+1 sfcât
   sfdacă
sf-CautSucc
```

În cel mai rău caz şi acest algoritm face acelaşi număr de operaţii ca şi subalgoritmul *Cautsecv*. În medie numărul operaţiilor este jumătate din numărul mediu de operaţii efecuat de subalgoritmul *Cautsecv* deci complexitatea este aceeaşi.

1.1.1.2. Căutare binară

O altă metodă, numită **căutare binară**, care este mult mai eficientă, utilizează tehnica "divide et impera" privitor la date. Se determină în ce relaţie se află cheia articolului aflat în mijlocul colecţiei cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecţiei. În acest mod, prin înjumătăţiri succesive se micşorează volumul colecţiei rămase pentru căutare. Căutarea binară se poate realiza practic prin apelul funcţiei `BinarySearch(a, n, K, 1, n)`, descrisă mai jos, folosită în subalgoritmul dat în continuare.

```
Subalgoritmul CautBin(a, n, K, p) este:           {n∈N, n≥1 şi k1 < k2 < .... < kn}
                                                  {Se caută  p astfel ca: (p=1 şi a ≤ k1) sau}
                                                  {(p=n+1 şi a>kn) sau (1<p≤n) şi (kp-1 < a ≤ kp)}
 Dacă a ≤ k1 atunci p := 1 altfel
     Dacă a > kn atunci p := n+1 altfel
         P := BinarySearch(a, n, K, 1, n)
     sfdacă
 sfdacă
sf-CautBin
```

```
Funcţia BinarySearch(a, n, K, St, Dr) este:
  Dacă St ≥ Dr - 1
     atunci BinarySearch := Dr
     altfel m := (St+Dr) Div 2;
       Dacă a ≤ k_m
          atunci BinarySearch := BinarySearch(a, n, K, St, m)
          altfel BinarySearch := BinarySearch(a, n, K, m, Dr)
       sfdacă
  sfdacă
sf-BinarySearch
```

În funcţia `BinarySearch` descrisă mai sus, variabilele `St` şi `Dr` reprezintă capetele intervalului de căutare, iar `m` reprezintă mijlocul acestui interval. Prin această metodă, într-o colecţie având `n` elemente, rezultatul căutării se poate furniza după cel mult $log_2 n$ comparări. Deci complexitatea în cel mai rău caz este direct proporţională cu $log_2 n$. Fără a insista asupra demonstraţiei, menţionăm că ordinul de mărime al complexităţii medii este acelaşi.

Se observă că funcţia `BinarySearch` se apelează recursiv. Se poate înlătura uşor recursivitatea, aşa cum se poate vedea în următoarea funcţie:

```
Funcţia BinSeaNerec(a, n, K, St, Dr) este:
 Câttimp Dr – St > 1 execută
         m := (St+Dr) Div 2;
         Dacă a ≤ k_m atunci Dr := m altfel St := m sfdacă
 sfcât
 BinSeaNerec := Dr
sf-BinSeaNerec
```

1.1.2. Sortări

Prin sortare internă vom înţelege o rearanjare a unei colecţii aflate în memoria internă astfel încât cheile articolelor să fie ordonate crescător (eventual descrescător).

Din punct de vedere al complexităţii algoritmilor problema revine la ordonarea cheilor. Deci specificarea problemei de **sortare internă** este următoarea:

Date n,K; *{K=(k_1,k_2,...,k_n)}*

Precondiţia: $k_i \in R$, i=1,n

Rezultate K';

Postcondiţia: K' este o permutare a lui K, dar ordonată crescător.

Deci $k_1 \le k_2 \le ... \le k_n$.

1.1.2.1. Sortare prin selecţie

O primă tehnică numită "*Selecţie*" se bazează pe următoarea idee: se determină poziţia elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecţia rămasă, până când mai rămâne doar elementul maxim.

```
Subalgoritmul Selectie(n, K) este:                    {Se face o permutare a celor}
                                                      {n componente ale vectorului K astfel}
                                                      {ca k1 ≤ k2 ≤ .... ≤ kn }
 Pentru i := 1; n-1 execută
   Fie ind := i;
   Pentru j := i + 1; n execută
      Dacă kj < kind atunci ind := j sfdacă
   sfpentru
   Dacă i < ind atunci t := ki; ki := kind; kind := t sfdacă
 sfpentru
sf-Selectie
```

Se observă că numărul de comparări este:

$$(n-1)+(n-2)+...+2+1=n(n-1)/2$$

indiferent de natura datelor. Deci complexitatea medie, dar şi în cel mai rău caz este $\boldsymbol{O(n^2)}$.

1.1.2.2. Bubble sort

Metoda "*BubbleSort*", compară două câte două elemente consecutive iar în cazul în care acestea nu se află în relaţia dorită, ele vor fi interschimbate. Procesul de comparare se va încheia în momentul în care toate perechile de elemente consecutive sunt în relaţia de ordine dorită.

```
Subalgoritmul BubbleSort(n, K) este:
  Repetă
    Fie kod := 0;                                  {Ipoteza "este ordine"}
    Pentru i := 2; n execută
      Dacă ki-1 > ki atunci
        t := ki-1;
        ki-1 := ki;
        ki := t;
        kod := 1                                        {N-a fost ordine!}
      sfdacă
    sfpentru
  pânăcând kod = 0 sfrep                                        {Ordonare}
sf-BubbleSort
```

Acest algoritm execută în cel mai nefavorabil caz (n-1)+(n-2)+ ... +2+1 = n(n-1)/2 comparări, deci complexitatea lui este $\boldsymbol{O(n^2)}$.

O variantă optimizată a algoritmului "*BubbleSort*" este :

```
Subalgoritmul BubbleSort(n, K) este:
  Fie  s := 0
  Repetă
    Fie kod := 0;                                        {Ipoteza "este ordine"}
    Pentru i := 2; n-s execută
      Dacă k_{i-1} > k_i atunci
        t := k_{i-1};
        k_{i-1} := k_i;
        k_i := t;
        kod := 1                                              {N-a fost ordine!}
      sfdacă
    sfpentru
    s := s + 1
  pânăcând kod = 0 sfrep                                                {Ordonare}
sf-BubbleSort
```

1.1.2.3. Quicksort

O metodă mai performantă de ordonare, care va fi prezentată în continuare, se numeşte "*QuickSort*" şi se bazează pe tehnica "divide et impera" după cum se poate observa în continuare. Metoda este prezentată sub forma unei proceduri care realizează ordonarea unui subşir precizat prin limita inferioară şi limita superioară a indicilor acestuia. Apelul procedurii pentru ordonarea întregului şir este : `QuickSort(n, K, 1, n)`, unde `n` reprezintă numărul de articole ale colecţiei date. Deci

```
Subalgoritmul SortareRapidă(n, K) este:
  Cheamă QuickSort(n, K, 1, n)
sf-SortareRapidă
```

Procedura `QuickSort(n, K, St, Dr)` va realiza ordonarea subşirului k_{St}, k_{St+1}, ..., k_{Dr}. Acest subşir va fi rearanjat astfel încât k_{St} să ocupe poziţia lui finală (când şirul este ordonat). Dacă `i` este această poziţie, şirul va fi rearanjat astfel încât următoarea condiţie să fie îndeplinită:

$$k_j \le k_i \le k_l \text{ , pentru } st \le j < i < l \le dr \qquad (*)$$

Odată realizat acest lucru, în continuare va trebui doar să ordonăm subşirul k_{St}, k_{St+1}, ... , k_{i-1} prin apelul recursiv al procedurii `QuickSort(n, K, St, i-1)` şi apoi subşirul k_{i+1}, ..., k_{Dr} prin apelul `QuickSort(n, K, i+1, Dr)`. Desigur ordonarea acestor două subşiruri (prin apelul recursiv al procedurii) mai este necesară doar dacă acestea conţin cel puţin două elemente.

Procedura `QuickSort` este prezentată în continuare :

```
Subalgoritmul QuickSort (n, K, St, Dr) este:
   Fie i := St; j := Dr; a := k_i;
   Repetă
     Câttimp k_j ≥ a şi (i < j) execută j := j - 1 sfcât
     k_i := k_j;
     Câttimp k_i ≤ a şi (i < j) execută i := i + 1 sfcât
     k_j := k_i ;
  pânăcând i = j sfrep
  Fie k_i := a;
  Dacă St < i-1 atunci Cheamă QuickSort(n, K, St, i - 1) sfdacă
  Dacă i+1 < Dr atunci Cheamă QuickSort(n, K, i + 1, Dr) sfdacă
sf-QuickSort
```

Complexitatea algoritmului prezentat este $O(n^2)$ în cel mai nefavorabil caz, dar complexitatea medie este de ordinul $\boldsymbol{O(n\log_2 n)}$.

1.1.3. Metoda backtracking

Metoda backtracking (căutare cu revenire) este aplicabilă in general unor probleme ce au mai multe soluţii.

Vom considera întâi un exemplu, după care vom indica câţiva algoritmi generali pentru această metodă.

Problema 1. (Generarea permutărilor) Fie *n* un număr natural. Determinaţi permutările numerelor 1, 2, ..., *n*.

O soluţie pentru generarea permutărilor, în cazul particular *n = 3*, ar putea fi:

```
Subalgoritmul Permutări1 este:
     Pentru i1 := 1; 3 execută
          Pentru i2 := 1; 3 execută
               Pentru i3 := 1; 3 execută
                    Fie posibil := (i1, i2, i3)
                    Dacă componentele vectorului posibil sunt distincte
                         atunci
                              Tipărește posibil
                    sfdacă
               sfpentru
          sfpentru
     sfpentru
sf-Permutări1
```

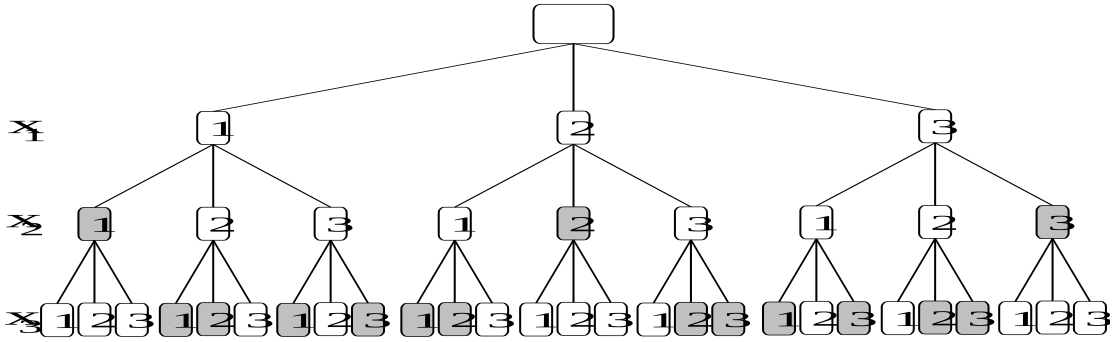


Figura 1.1 Reprezentare grafică a produsului cartezian $\{1, 2, 3\}^3$

Observaţii privind subalgoritmul *Permutări1*:

- Pentru *n* oarecare nu putem descrie un algoritm care să conţină *n* cicluri în textul sursă.
- Numărul total de vectori verificaţi este 3^3, iar în general n^n. Vectorii *posibil* verificaţi sunt reprezentaţi grafic în Figura 1.1 - fiecare vector este un drum de la rădăcină (de sus) spre frunze (baza arborelui).
- Algoritmul *atribuie valori tuturor componentelor vectorului x, apoi verifică dacă vectorul este o permutare*.

O îmbunătăţire a acestor algoritmi ar consta în a verifica anumite condiţii din problemă în timp ce se construiesc vectorii, evitând completarea inutilă a unor componente.

De exemplu, dacă prima componentă a vectorului construit (*posibil*) este 1, atunci este inutil să atribuim celei de a doua componente valoarea 1, iar componentei a treia oricare din valorile 1, 2 sau 3. Dacă *n* este mare se evită completarea multor vectori ce au prefixul (1, ...). În acest sens, (1, 3, ...) este un *vector promiţător* (pentru a fi o permutare), în schimb vectorul (1, 1, ...) nu este. Vectorul (1, 3, ...) satisface anumite *condiţii de continuare* (pentru a ajunge la soluţie) - are componente distincte. Nodurile haşurate din Figura 1.1 constituie valori care nu conduc la o soluţie.

Vom descrie un algoritm general pentru metoda Bactracking după care vom particulariza acest algoritm pentru problemele enunţate la începutul secţiunii. Pentru început vom face câteva observaţii şi notaţii privind metoda Backtracking aplicată unei probleme în care soluţiile se reprezintă pe vectori, nu neapărat de lungime fixă:

- spaţiul de căutare a soluţiilor (spaţiul soluţiilor posibile): $S = S_1 \times S_2 \times ... \times S_n$;
- *posibil* este vectorul pe care se reprezintă soluţiile;
- *posibil*[1..*k*] $\in S_1 \times S_2 \times ... \times S_k$ este un vector care poate conduce sau nu la o soluţie; *k* reprezintă indice pentru vectorul *posibil*, respectiv nivel în arborele care redă grafic procesul de căutare (Figura 1.2).
- *posibil*[1..*k*] este promiţător dacă satisface condiţii care pot conduce la o soluţie;
- *soluţie*(*n, k, posibil*) funcţie care verifică dacă vectorul (promiţător) *posibil*[1..*k*] este soluţie a problemei.

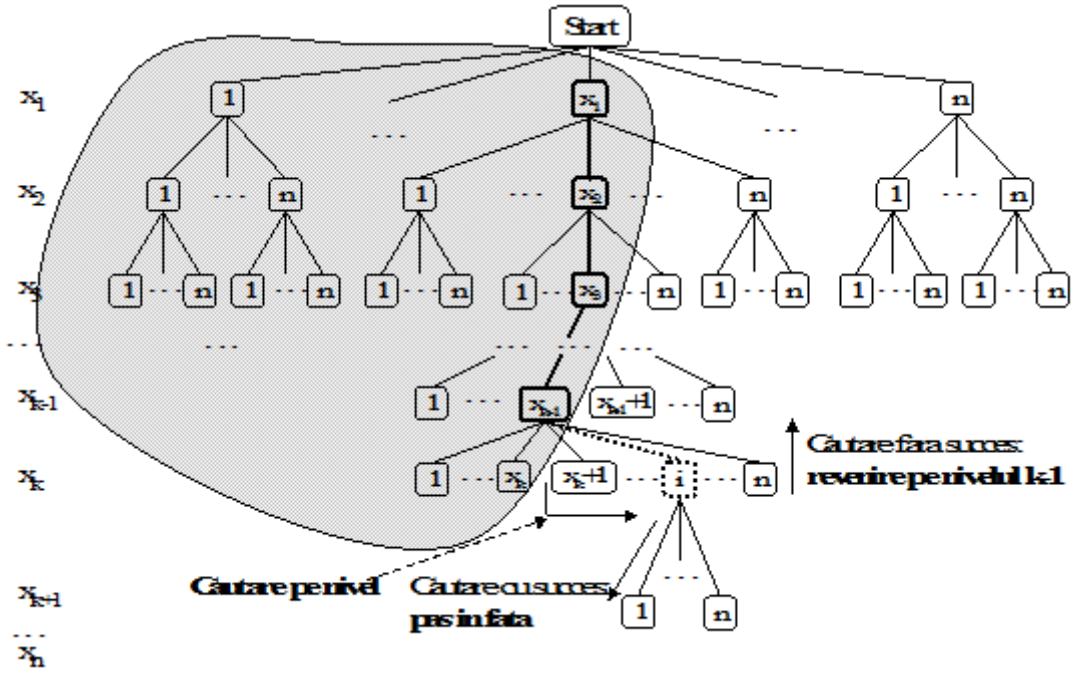


Figura 1.2. Spaţiul soluţiilor posibile pentru generarea permutărilor

Procesul de căutare poate fi urmărit în algoritmul care urmează:

```
Algoritmul Backtracking  este:                          {varianta nefinisată}
Fie k := 1
@Iniţializează căutarea pe nivelul k (= 1)
Câttimp k > 0 execută                          {posibil[1..k-1] este promiţător}
     @Caută (secvenţial)  pe nivelul  k o valoare v, pentru a completa în
          continuare vectorul posibil[1..k-1] astfel încât  posibil[1..k]  să
          fie promiţător
     Dacă căutarea este cu succes
          atunci Fie posibil[k] := v        {posibil[1..k] este promiţător}
              Dacă soluţie(n, k, posibil)
                atunci                  {o soluţie! (rămânem pe nivelul k)}
                    Tipareşte posibil[1..k]
                altfel                          {e doar un vector promiţător}
                    @Initializeaza cautarea pe nivelul k+1
                    Fie k := k + 1              {pas în faţă (pe nivelul k+1)}
              sfdacă
          altfel                  {pas în spate (revenire pe nivelul k-1)}
          k := k - 1
     sfdacă
sfcât
sf-Backtracking
```

Pentru a finisa acest algoritm trebuie să precizăm elementele nestandard prezente. Astfel, avem nevoie de funcţia booleană

```
condiţii-continuare(k, posibil, v)
```

funcţie care verifică dacă vectorul promiţător *posibil*[1..*k*-1] completat cu valoarea *v* conduce la un vector promiţător.

Apoi, pentru a iniţializa căutarea la nivelul *j* avem nevoie de a alege un element fictiv din mulţimea S_j, activitate realizată de funcţia

```
init(j)
```

care returnează acest element fictiv, care are rolul de a indica faptul că din mulţimea S încă nu s-a ales nici un element, deci după el urmează primul element propriu din această mulţime. Pentru a căuta o valoare pe nivelul j, în ipoteza că valoarea curentă nu e bună, avem nevoie de funcţia booleană

```
următor(j, v, nou)
```

care este *True* dacă poate alege o valoare din S_j care urmează după valoarea *v*, valoare notată prin *nou* şi *False* în cazul în care nu mai există alte valori în S_j, deci nu mai poate fi făcută alegerea. Cu aceste notaţii algoritmul devine:

```
Algoritmul Backtracking  este:                          {versiune finală}
Fie k := 1;
posibil[1] := init(1);
Câttimp k > 0 execută                          {posibil[1..k-1] este promiţător}
     Fie Găsit := false; v := posibil[k];
     Câttimp Următor(k, v,urm) şi not Găsit execută
          Fie v := urm;
          Dacă condiţii-continuare(k, posibil, v) atunci
               Găsit := true
          sfdacă
     sfcât
     Dacă Găsit
          atunci Fie posibil[k] := v;       {posibil[1..k] este promiţător}
              Dacă soluţie(n, k, posibil)
                atunci                 {o soluţie! (rămânem pe nivelul k)}
                    Tipareşte posibil[1..k]
                altfel                          {e doar un vector promiţător}
                    Fie k := k + 1;             {pas în faţă (pe nivelul k+1)}
                    posibil[k] := init(k)
              sfdacă
          altfel                  {pas în spate (revenire pe nivelul k-1)}
          k := k - 1;
     sfdacă
sfcât
sf-Backtracking
```

Procesul de căutare a unei valori pe nivelul *k* şi funcţiile *condiţii-continuare* şi *soluţie* sunt dependente de problema care se rezolvă. De exemplu, pentru generarea permutărilor funcţiile menţionate sunt:

```
Funcţia init(k) este:
   Init := 0
sf-init;

Funcţia Următor(k, v, urm) este:
 Dacă  v < n
   atunci Următor := True; urm := v + 1
   altfel Următor := False
 sfdacă
sf-urmator

Funcţia conditii-continuare(k, posibil, v) este:
    Kod := True;   i := 1;
    Câttimp  kod şi (i < k)  execută
        Dacă posibil[i] = v atunci kod := False sfdacă
        i := i + 1;
```

```
          sfcât
          conditii-continuare:=kod
      sf-conditii

      Funcţia soluţii(n, k, posibil) este:
         Soluţii := (k = n)
      sf-solutii
```

În încheiere, menţionăm că explorarea backtracking poate fi descrisă de asemenea recursiv. Dăm în acest scop următoru subalgoritm:

```
Subalgoritmul Backtracking(k, posibil)  este:

  {posibil[1..k] este promiţător}
  Dacă soluţie(n, k, posibil)  atunci
            {o soluţie! terminare apel recursiv, astfel}
            Tipareste posibil[1..k]
           {rămânem pe acelaşi nivel}
  altfel
     Pentru fiecare v valoare posibilă pentru posibil[k+1] execută
           Dacă condiţii-continuare(k + 1, posibil, v) atunci
                  posibil[k + 1] := v
                 Backtracking(k + 1, posibil)
                {pas in faţă}
            sfdacă
       sfpentru
  sfdacă
  {terminare apel Backtracking(k, posibil)}
sf-Backtracking {deci, pas în spate (revenire)}
```

cu apelul iniţial **Cheamă** `Backtracking(0, posibil)`.

1.2. Concepte OOP în limbaje de programare

1.2.1. Noţiunea de clasă

1.2.1.1. Realizarea protecţiei datelor prin metoda programării modulare

Dezvoltarea programelor prin programare procedurală înseamnă folosirea unor funcţii şi proceduri pentru scrierea programelor. În limbajul C lor le corespund funcţiile care returnează o valoare sau nu. Însă în cazul aplicaţiilor mai mari ar fi de dorit să putem realiza şi o protecţie corespunzătoare a datelor. Acest lucru ar însemna că numai o parte a funcţiilor să aibă acces la datele problemei, acelea care se referă la datele respective. Programarea modulară oferă o posibilitate de realizare a protecţiei datelor prin folosirea clasei de memorie static. Dacă într-un fişier se declară o dată aparţinând clasei de memorie statică în afara funcţiilor, atunci ea poate fi folosită începând cu locul declarării până la sfârşitul modulului respectiv, dar nu şi în afara lui.

Să considerăm următorul exemplu simplu referitor la prelucrarea vectorilor de numere întregi. Să se scrie un modul referitor la prelucrarea unui vector cu elemente întregi, cu funcţii corespunzătoare pentru iniţializarea vectorului, eliberarea zonei de memorie ocupate şi ridicarea la pătrat, respectiv afişarea elementelor vectorului. O posibilitate de implementare a modulului este prezentată în fişierul **vector1.cpp**:

```
#include <iostream>

using namespace std;

static int* e;              //elementele vectorului
static int d;               //dimensiunea vectorului

void init(int* e1, int d1)  //initializare
{
  d = d1;
  e = new int[d];
  for(int i = 0; i < d; i++)
    e[i] = e1[i];
}

void distr()                //eliberarea zonei de memorie ocupata
{
  delete [] e;
}

void lapatrat()             //ridicare la patrat
{
  for(int i = 0; i < d; i++)
    e[i] *= e[i];
}

void afiseaza()             //afisare
{
  for(int i = 0; i < d; i++)
    cout << e[i] << ' ';
  cout << endl;
}
```

Modulul se compilează separat obţinând un program obiect. Un exemplu de program principal este prezentat în fişierul **vector2.cpp**:

```
extern void init( int*, int);  //extern poate fi omis
extern void distr();
extern void lapatrat();
extern void afiseaza();
//extern int* e;
int main() {
  int x[5] = {1, 2, 3, 4, 5};
  init(x, 5);
  lapatrat();
  afiseaza();
  distr();
  int y[] = {1, 2, 3, 4, 5, 6};
  init(y, 6);
  //e[1]=10;                   eroare, datele sunt protejate
  lapatrat();
  afiseaza();
  distr();
```

```
  return 0;
}
```

Observăm că deşi în programul principal se lucrează cu doi vectori nu putem să-i folosim împreună, deci de exemplu modulul **vector1.cpp** nu poate fi extins astfel încât să realizeze şi adunarea a doi vectori. În vederea înlăturării acestui neajuns s-au introdus tipurile abstracte de date.

1.2.1.2. Tipuri abstracte de date

Tipurile abstracte de date realizează o legătură mai strânsă între datele problemei şi operaţiile (funcţiile) care se referă la aceste date. Declararea unui tip abstract de date este asemănătoare cu declararea unei structuri, care în afară de date mai cuprinde şi declararea sau definirea funcţiilor referitoare la acestea.

De exemplu în cazul vectorilor cu elemente numere întregi putem declara tipul abstract:

```
struct vect {
  int* e;
  int d;
  void init(int* e1, int d1);
  void distr() { delete [] e; }
  void lapatrat();
  void afiseaza();
};
```

Funcţiile declarate sau definite în interiorul structurii vor fi numite *funcţii membru* iar datele *date membru*. Dacă o funcţie membru este definită în interiorul structurii (ca şi funcţia *distr* din exemplul de mai sus), atunci ea se consideră funcţie *inline*. Dacă o funcţie membru se defineşte în afara structurii, atunci numele funcţiei se va înlocui cu numele tipului abstract urmat de operatorul de rezoluţie (::) şi numele funcţiei membru. Astfel funcţiile *init*, *lapatrat* şi *afiseaza* vor fi definite în modul următor:

```
void vect::init(int *e1, int d1)
{
  d = d1;
  e = new int[d];
  for(int i = 0; i < d; i++)
    e[i] = e1[i];
}

void vect::lapatrat()
{
  for(int i = 0; i < d; i++)
    e[i] *= e[i];
}
void vect::afiseaza()
{
  for(int i = 0; i < d; i++)
    cout << e[i] << ' ';
  cout << endl;
}
```

Deşi prin metoda de mai sus s-a realizat o legătură între datele problemei şi funcţiile referitoare la aceste date, ele nu sunt protejate, deci pot fi accesate de orice funcţie utilizator, nu numai de funcţiile membru. Acest neajuns se poate înlătura cu ajutorul claselor.

1.2.1.3. Declararea claselor

Un tip abstract de date clasă se declară ca şi o structură, dar cuvântul cheie struct se înlocuieşte cu *class*. Ca şi în cazul structurilor referirea la tipul de dată clasă se face cu numele după cuvântul cheie class (numele clasei). Protecţia datelor se realizează cu modificatorii de protecţie: *private*, *protected* şi *public*. După modificatorul de protecţie se pune caracterul ':'. Modificatorul *private* şi *protected* reprezintă date protejate, iar *public* date neprotejate. Domeniul de valabilitate a modificatorilor de protecţie este până la următorul modificator din interiorul clasei, modificatorul implicit fiind *private*. Menţionăm că şi în cazul structurilor putem să folosim modificatori de protecţie, dar în acest caz modificatorul implicit este *public*.

De exemplu clasa vector se poate declara în modul următor:

```
class vector {
  int* e;  //elementele vectorului
  int d;   //dimensiunea vectorului
public:
  vector(int* e1, int d1);
  ~vector() { delete [] e; }
  void lapatrat();
  void afiseaza();
};
```

Se observă că datele membru *e* şi *d* au fost declarate ca date de tip *private* (protejate), iar funcţiile membru au fost declarate publice (neprotejate). Bineînţeles, o parte din datele membru pot fi declarate publice, şi unele funcţii membru pot fi declarate protejate, dacă natura problemei cere acest lucru. În general, datele membru protejate pot fi accesate numai de funcţiile membru ale clasei respective şi eventual de alte funcţii numite *funcţii prietene (*sau funcţii *friend).*

O altă observaţie importantă referitoare la exemplul de mai sus este că iniţializarea datelor membru şi eliberarea zonei de memorie ocupată s-a făcut prin funcţii membru specifice.

Datele declarate cu ajutorul tipului de dată clasă se numesc *obiectele* clasei, sau simplu *obiecte*. Ele se declară în mod obişnuit în forma:

```
nume_clasă listă_de_obiecte;
```

De exemplu, un obiect de tip vector se declară în modul următor:

```
vector v;
```

Iniţializarea obiectelor se face cu o funcţie membru specifică numită *constructor*. În cazul distrugerii unui obiect se apelează automat o altă funcţie membru specifică numită *destructor*. În cazul exemplului de mai sus

```
vector(int* e1, int d1);
```

este un constructor, iar

```
~vector() { delete [] e; }
```

este un destructor.

Tipurile abstracte de date de tip *struct* pot fi şi ele considerate clase cu toate elementele neprotejate. Constructorul de mai sus este declarat în interiorul clasei, dar nu este definit, iar destructorul este definit în interiorul clasei. Rezultă că destructorul este o funcţie inline. Definirea funcţiilor membru care sunt declarate, dar nu sunt definite în interiorul clasei se face ca şi în cazul tipurilor abstracte de date de tip struct, folosind operatorul de rezoluţie.

1.2.1.4. Membrii unei clase. Pointerul this

Referirea la datele respectiv funcţiile membru ale claselor se face cu ajutorul operatorilor punct (.) sau săgeată (->) ca şi în cazul referirii la elementele unei structuri. De exemplu, dacă se declară:

```
vector v;
vector* p;
```

atunci afişarea vectorului v respectiv a vectorului referit de pointerul p se face prin:

```
v.afiseaza();
p->afiseaza();
```

În interiorul funcţiilor membru însă referirea la datele respectiv funcţiile membru ale clasei se face simplu prin numele acestora fără a fi nevoie de operatorul punct (.) sau săgeată (->). De fapt compilatorul generează automat un pointer special, pointerul *this*, la fiecare apel de funcţie membru, şi foloseşte acest pointer pentru identificarea datelor şi funcţiilor membru.

Pointerul *this* va fi declarat automat ca pointer către obiectul curent. În cazul exemplului de mai sus pointerul *this* este adresa vectorului *v* respectiv adresa referită de pointerul *p*.

Dacă în interiorul corpului funcţiei membru *afiseaza* se utilizează de exemplu data membru *d*, atunci ea este interpretată de către compilator ca şi this->d.

Pointerul *this* poate fi folosit şi în mod explicit de către programator, dacă natura problemei necesită acest lucru.

1.2.1.5. Constructorul

Iniţializarea obiectelor se face cu o funcţie membru specifică numită constructor. Numele constructorului trebuie să coincidă cu numele clasei. O clasă poate să aibă mai mulţi constructori. În acest caz aceste funcţii membru au numele comun, ceea ce se poate face datorită posibilităţii de supraîncărcare a funcţiilor. Bineînţeles, în acest caz numărul şi/sau

tipul parametrilor formali trebuie să fie diferit, altfel compilatorul nu poate să aleagă constructorul corespunzător.

Constructorul nu returnează o valoare. În acest caz nu este permis nici folosirea cuvântului cheie *void*.

Prezentăm în continuare un exemplu de tip clasa cu mai mulţi constructori, având ca date membru numele şi prenumele unei persoane, şi cu o funcţie membru pentru afişarea numelui complet.

Fişierul **`persoana.h`**:

```
class persoana {
  char* nume;
  char* prenume;
public:
  persoana();                       //constructor implicit
  persoana(char* n, char* p);       //constructor
  persoana(const persoana& p1);     //constructor de copiere
  ~persoana();                      //destructor
  void afiseaza();
};
```

Fişierul **`persoana.cpp`**:

```
#include <iostream>
#include <cstring>
#include "persoana.h"

using namespace std;

persoana::persoana()
{
  nume = new char[1];
  *nume = 0;
  prenume = new char[1];
  *prenume = 0;
  cout << "Apelarea constructorului implicit." << endl;
}

persoana::persoana(char* n, char* p)
{
  nume = new char[strlen(n)+1];
  prenume = new char[strlen(p)+1];
  strcpy(nume, n);
  strcpy(prenume, p);
  cout << "Apelare constructor (nume, prenume).\n";
}

persoana::persoana(const persoana& p1)
{
  nume = new char[strlen(p1.nume)+1];
  strcpy(nume, p1.nume);
  prenume = new char[strlen(p1.prenume)+1];
  strcpy(prenume, p1.prenume);
  cout << "Apelarea constructorului de copiere." << endl;
}
```

```
persoana::~persoana()
{
  delete[] nume;
  delete[] prenume;
}

void persoana::afiseaza()
{
  cout << prenume << ' ' << nume << endl;
}
```

Fişierul **`persoanaTest.cpp`**:

```
#include "persoana.h"

int main() {
  persoana A;             //se apeleaza constructorul implicit
  A.afiseaza();
  persoana B("Stroustrup", "Bjarne");
  B.afiseaza();
  persoana *C = new persoana("Kernighan","Brian");
  C->afiseaza();
  delete C;
  persoana D(B);          //echivalent cu persoana D = B;
                          //se apeleaza constructorul de copire
  D.afiseaza();
  return 0;
}
```

Observăm prezenţa a doi constructori specifici: *constructorul implicit* şi *constructorul de copiere*. Dacă o clasă are constructor fără parametri atunci el se va numi *constructor implicit. Constructorul de copiere* se foloseşte la iniţializarea obiectelor folosind un obiect de acelaşi tip (în exemplul de mai sus o persoană cu numele şi prenumele identic). Constructorul de copiere se declară în general în forma:

```
nume_clasă(const nume_clasă& obiect);
```

Cuvântul cheie *const* exprimă faptul că argumentul constructorului de copiere nu se modifică.

O clasă poate să conţină ca date membru obiecte ale unei alte clase. Declarând clasa sub forma:

```
class nume_clasa {
  nume_clasa_1 ob_1;
  nume_clasa_2 ob_2;
  ...
  nume_clasa_n ob_n;
  ...
};
```

antetul constructorului clasei *nume_clasa* va fi de forma:

```
nume_clasa(lista_de_argumente):
      ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)
```

unde `lista_de_argumente` respectiv `l_arg_i` reprezintă lista parametrilor formali ai constructorului clasei `nume_clasa` respectiv ai obiectului `ob_i`.

Din lista `ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)` pot să lipsească obiectele care nu au constructori definiţi de programator, sau obiectul care se iniţializează cu un constructor implicit, sau cu toţi parametrii impliciţi.

Dacă clasa conţine date membru de tip obiect atunci se vor apela mai întâi constructorii datelor membru, iar după aceea corpul de instrucţiuni al constructorului clasei respective.

Fişierul **`pereche.cpp`**:

```
#include <iostream>
#include "persoana.h"

using namespace std;

class pereche {
  persoana sot;
  persoana sotie;
public:
  pereche()               //definitia constructorului implicit
  {                       //se vor apela constructorii impliciti
  }                       //pentru obiectele sot si sotie
  pereche(persoana& sotul, persoana& sotia);
  pereche(char* nume_sot, char* prenume_sot,
    char* nume_sotie, char* prenume_sotie):
    sot(nume_sot, prenume_sot),
    sotie(nume_sotie, prenume_sotie)
  {
  }
  void afiseaza();
};

inline pereche::pereche(persoana& sotul, persoana& sotia):
  sot(sotul), sotie(sotia)
{
}

void pereche::afiseaza()
{
  cout << "Sot: ";
  sot.afiseaza();
  cout << "Sotie: ";
  sotie.afiseaza();
}

int main() {
  persoana A("Pop", "Ion");
  persoana B("Popa", "Ioana");
  pereche AB(A, B);
  AB.afiseaza();
  pereche CD("C","C","D","D");
  CD.afiseaza();
  pereche EF;
```

```
  EF.afiseaza();
  return 0;
}
```

Observăm că în cazul celui de al doilea constructor, parametrii formali *sot* şi *sotie* au fost declaraţi ca şi referinţe la tipul *persoana*. Dacă ar fi fost declaraţi ca parametri formali de tip *persoana*, atunci în cazul declaraţiei:

```
pereche AB(A, B);
```

constructorul de copiere s-ar fi apelat de patru ori. În astfel de situaţii se creează mai întâi obiecte temporale folosind constructorul de copiere (două apeluri în cazul de faţă), după care se execută constructorii datelor membru de tip obiect (încă două apeluri).

1.2.1.6. Destructorul

Destructorul este funcţia membru care se apelează în cazul distrugerii obiectului. Destructorul obiectelor globale se apelează automat la sfârşitul funcţiei *main* ca parte a funcţiei *exit*. Deci, nu este indicată folosirea funcţiei *exit* într-un destructor, pentru că acest lucru duce la un ciclu infinit. Destructorul obiectelor locale se execută automat la terminarea blocului în care s-au definit. În cazul obiectelor alocate dinamic, de obicei destructorul se apelează indirect prin operatorul *delete* (obiectul trebuie să fi fost creat cu operatorul *new*). Există şi un mod explicit de apelare a destructorului, în acest caz numele destructorului trebuie precedat de numele clasei şi operatorul de rezoluţie.

Numele destructorului începe cu caracterul ~ după care urmează numele clasei. Ca şi în cazul constructorului, destructorul nu returnează o valoare şi nu este permisă nici folosirea cuvântului cheie void. Apelarea destructorului în diferite situaţii este ilustrată de următorul exemplu. Fişierul **`destruct.cpp`**:

```
#include <iostream>
#include <cstring>

using namespace std;

class scrie {  //scrie pe stdout ce face.
  char* nume;
public:
  scrie(char* n);
  ~scrie();
};

scrie::scrie(char* n)
{
  nume = new char[strlen(n)+1];
  strcpy(nume, n);
  cout << "Am creat obiectul: " << nume << '\n';
}

scrie::~scrie()
{
  cout << "Am distrus obiectul: " << nume << '\n';
  delete nume;
```

```
}

void functie()
{
  cout << "Apelare functie" << '\n';
  scrie local("Local");
}

scrie global("Global");

int main() {
  scrie* dinamic = new scrie("Dinamic");
  functie();
  cout << "Se continua programul principal" << '\n';
  delete dinamic;
  return 0;
}
```

1.3. Relaţii între clase

1.3.1. Bazele teoretice

Prin folosirea tipurilor abstracte de date, se creează un tot unitar pentru gestionarea datelor şi a operaţiilor referitoare la aceste date. Cu ajutorul tipului abstract clasă se realizează şi protecţia datelor, deci în general elementele protejate nu pot fi accesate decât de funcţiile membru ale clasei respective. Această proprietate a obiectelor se numeşte *încapsulare (encapsulation).*

În viaţa de zi cu zi însă ne întâlnim nu numai cu obiecte separate, dar şi cu diferite legături între aceste obiecte, respectiv între clasele din care obiectele fac parte. Astfel se formează o ierarhie de clase. Rezultă a doua proprietate a obiectelor: *moştenirea (inheritance)*. Acest lucru înseamnă că se moştenesc toate datele şi funcţiile membru ale clasei de bază de către clasa derivată, dar se pot adăuga elemente noi (date membru şi funcţii membru) în clasa derivată. În cazul în care o clasă derivată are mai multe clase de bază se vorbeşte despre *moştenire multiplă*.

O altă proprietate importantă a obiectelor care aparţin clasei derivate este că funcţiile membru moştenite pot fi supraîncărcate. Acest lucru înseamnă că o operaţie referitoare la obiectele care aparţin ierarhiei are un singur identificator, dar funcţiile care descriu această operaţie pot fi diferite. Deci, numele funcţiei şi lista parametrilor formali este aceeaşi în clasa de bază şi în clasa derivată, dar descrierea funcţiilor diferă între ele. Astfel, în clasa derivată funcţiile membru pot fi specifice clasei respective, deşi operaţia se identifică prin acelaşi nume. Această proprietate se numeşte *polimorfism*.

1.3.2. Declararea claselor derivate

O clasă derivată se declară în felul următor:

```
class nume_clasă_derivată : lista_claselor_de_bază {
  //date membru noi şi funcţii membru noi
};
```

unde `lista_claselor_de_bază` este de forma:

`elem_1, elem_2, ..., elem_n`

şi `elem_i` pentru orice $1 \leq i \leq n$ poate fi

`public clasă_de_bază_i`

sau

`protected clasă_de_bază_i`

sau

`private clasă_de_bază_i`

Cuvintele cheie *public*, *protected* şi *private* se numesc şi de această dată *modificatori de protecţie*. Ei pot să lipsească, în acest caz modificatorul implicit fiind *private*. Accesul la elementele din clasa derivată este prezentat în tabelul 1.

Accesul la elementele din clasa de bază	Modificatorii de protecţie referitoare la clasa de bază	Accesul la elementele din clasa derivată
public	public	public
protected	public	protected
private	public	inaccesibil
public	protected	protected
protected	protected	protected
private	protected	inaccesibil
public	private	private
protected	private	private
private	private	inaccesibil

Tabelul 1: accesul la elementele din clasa derivată

Observăm că elementele de tip *private* ale clasei de bază sunt inaccesibile în clasa derivată. Elementele de tip *protected* şi *public* devin de tip *protected*, respectiv *private* dacă modificatorul de protecţie referitor la clasa de bază este *protected* respectiv *private*, şi rămân neschimbate dacă modificatorul de protecţie referitor la clasa de bază este *public*. Din acest motiv în general datele membru se declară de tip *protected* şi modificatorul de protecţie referitor la clasa de bază este *public*. Astfel datele membru pot fi accesate, dar rămân protejate şi în clasa derivată.

1.3.3. Funcţii virtuale

Noţiunea de polimorfism ne conduce în mod firesc la problematica determinării funcţiei membru care se va apela în cazul unui obiect concret. Să considerăm următorul exemplu. Declarăm clasa de bază *baza*, şi o clasă derivată din acestă clasă de bază, clasa *derivata*. Clasa de bază are două funcţii membru: *functia_1* şi *functia_2*. În interiorul funcţiei membru *functia_2* se apelează *functia_1*. În clasa derivată se supraîncarcă funcţia membru *functia_1*, dar funcţia membru *functia_2* nu se supraîncarcă. În programul principal se declară un obiect al clasei derivate şi se apelează funcţia membru *functia_2* moştenită de la clasa de bază. În limbajul C++ acest exemplu se scrie în următoarea formă.

Fişierul `virtual1.cpp`:

```
#include <iostream>

using namespace std;

class baza {
  public:
    void functia_1();
    void functia_2();
};

class derivata : public baza {
  public:
    void functia_1();
};

void baza::functia_1()
{
  cout << "S-a apelat functia membru functia_1"
       << " a clasei de baza" << endl;
}

void baza::functia_2()
{
  cout << "S-a apelat functia membru functia_2"
       << " a clasei de baza" << endl;
  functia_1();
}

void derivata::functia_1()
{
  cout << "S-a apelat functia membru functia_1"
       << " a clasei derivate" << endl;
}

int main() {
  derivata D;
  D.functia_2();
}
```

Prin execuţie se obţine următorul rezultat:

```
S-a apelat functia membru functia_2 a clasei de baza
S-a apelat functia membru functia_1 a clasei de baza
```

Însă acest lucru nu este rezultatul dorit, deoarece în cadrul funcţiei *main* s-a apelat funcţia membru *functia_2* moştenită de la clasa de bază, dar funcţia membru *functia_1* apelată de *functia_2* s-a determinat încă în faza de compilare. În consecinţă, deşi funcţia membru *functia_1* s-a supraîncărcat în clasa derivată nu s-a apelat funcţia supraîncărcată ci funcţia membru a clasei de bază.

Acest neajuns se poate înlătura cu ajutorul introducerii noţiunii de funcţie membru *virtuală*. Dacă funcţia membru este virtuală, atunci la orice apelare a ei, determinarea funcţiei membru corespunzătoare a ierarhiei de clase nu se va face la compilare ci la execuţie, în funcţie de natura obiectului pentru care s-a făcut apelarea. Această proprietate se numeşte *legare dinamică*, iar dacă determinarea funcţiei membru se face la compilare, atunci se vorbeşte de *legare statică*.

Am văzut că dacă se execută programul `virtual1.cpp` se apelează funcţiile membru *functia_1* şi *functia_2* ale clasei de bază. Însă funcţia membru *functia_1* fiind supraîncărcată în clasa derivată, ar fi de dorit ca funcţia supraîncărcată să fie apelată în loc de cea a clasei de bază.

Acest lucru se poate realiza declarând *functia_1* ca funcţie membru virtuală. Astfel, pentru orice apelare a funcţiei membru *functia_1*, determinarea acelui exemplar al funcţiei membru din ierarhia de clase care se va executa, se va face la execuţie şi nu la compilare. Ca urmare, funcţia membru *functia_1* se determină prin *legare dinamică*.

În limbajul C++ o funcţie membru se declară virtuală în cadrul declarării clasei respective în modul următor: antetul funcţiei membru se va începe cu cuvântul cheie *virtual*.

Dacă o funcţie membru se declară virtuală în clasa de bază, atunci supraîncărcările ei se vor considera virtuale în toate clasele derivate ale ierarhiei.

În cazul exemplului de mai sus declararea clasei de bază se modifică în felul următor.

```
class baza {
  public:
    virtual void functia_1();
    void functia_2();
};
```

Rezultatul obţinut prin execuţie se modifică astfel:

```
S-a apelat functia membru functia_2 a clasei de baza
S-a apelat functia membru functia_1 a clasei derivate
```

Deci, într-adevăr se apelează funcţia membru *functia_1* a clasei derivate.

Prezentăm în continuare un alt exemplu în care apare necesitatea introducerii funcţiilor membru virtuale. Să se definească clasa *fractie* referitoare la numerele raţionale, având ca date membru numărătorul şi numitorul fracţiei. Clasa trebuie să aibă un constructor, valoarea implicită pentru numărător fiind zero iar pentru numitor unu, precum şi două funcţii membru: *produs* pentru a calcula produsul a două fracţii şi *inmulteste* pentru înmulţirea obiectului curent cu fracţia dată ca şi parametru. De asemenea, clasa *fractie* trebuie să aibă şi o funcţie membru pentru afişarea unui număr raţional. Folosind clasa *fractie* ca şi clasă de bază se va defini clasa derivată *fractie_scrie*, pentru care se va supraîncărca funcţia *produs*, astfel încât concomitent cu efectuarea înmulţirii să se afişeze pe *stdout* operaţia respectivă. Funcţia *inmulteste* nu se va supraîncărca, dar operaţia efectuată trebuie să se afişeze pe dispozitivul standard de ieşire şi în acest caz. Fişierul **`fvirt1.cpp`**:

```
#include <iostream>

using namespace std;

class fractie {
protected:
     int numarator;
     int numitor;
public:
     fractie(int numarator1 = 0, int numitor1 = 1);
     fractie produs(fractie& r);    //calculeaza produsul a doua
                                    //fractii, dar nu simplifica
     fractie& inmulteste(fractie& r);
     void afiseaza();
};

fractie::fractie(int numarator1, int numitor1)
{
     numarator = numarator1;
     numitor   = numitor1;
}

fractie fractie::produs(fractie& r)
{
     return fractie(numarator * r.numarator, numitor * r.numitor);
}

fractie& fractie::inmulteste(fractie& q)
{
     *this = this->produs(q);
     return *this;
}

void fractie::afiseaza()
{
     if ( numitor )
          cout << numarator << " / " << numitor;
     else
          cerr << "Fractie incorecta";
}

class fractie_scrie: public fractie{
public:
     fractie_scrie( int numarator1 = 0, int numitor1 = 1 );
     fractie produs( fractie& r);
};

inline fractie_scrie::fractie_scrie(int numarator1, int numitor1) :
fractie(numarator1, numitor1)
{
}

fractie fractie_scrie::produs(fractie& q)
{
     fractie r = fractie(*this).produs(q);
     cout << "(";
     this->afiseaza();
     cout << ") * (";
     q.afiseaza();
     cout << ") = ";
```

```
     r.afiseaza();
     cout << endl;
     return r;
}

int main()
{
     fractie p(3,4), q(5,2), r;
     r = p.inmulteste(q);
     p.afiseaza();
     cout << endl;
     r.afiseaza();
     cout << endl;
     fractie_scrie p1(3,4), q1(5,2);
     fractie r1, r2;
     r1 = p1.produs(q1);
     r2 = p1.inmulteste(q1);
     p1.afiseaza();
     cout << endl;
     r1.afiseaza();
     cout << endl;
     r2.afiseaza();
     cout << endl;
     return 0;
}
```

Prin execuţie se obţine:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

Observăm că rezultatul nu este cel dorit, deoarece operaţia de înmulţire s-a afişat numai o singură dată, şi anume pentru expresia `r1 = p1.produs(q1)`. În cazul expresiei `r2 = p1.inmulteste(q1)` însă nu s-a afişat operaţia de înmulţire. Acest lucru se datorează faptului că funcţia membru *inmulteste* nu s-a supraîncărcat în clasa derivată. Deci s-a apelat funcţia moştenită de la clasa *fractie*. În interiorul funcţiei *inmulteste* s-a apelat funcţia membru *produs*, dar deoarece această funcţie membru s-a determinat încă în faza de compilare, rezultă că s-a apelat funcţia referitoare la clasa *fractie* şi nu cea referitoare la clasa derivată *fractie_scrie*. Deci, afişarea operaţiei s-a efectuat numai o singură dată.

Soluţia este, ca şi în exemplul anterior, declararea unei funcţii membru virtuale, şi anume funcţia *produs* se va declara ca funcţie virtuală. Deci declararea clasei de bază se modifică în felul următor:

```
class fractie {
protected:
     int numarator;
     int numitor;
public:
     fractie(int numarator1 = 0, int numitor1 = 1);
     virtual fractie produs(fractie& r);  //calculeaza produsul a doua
                                          //fractii, dar nu simplifica
     fractie& inmulteste(fractie& r);
```

```
    void afiseaza();
};
```

După efectuarea acestei modificări prin executarea programului obţinem:

```
15 / 8
15 / 8
(3 / 4) * (5 / 2) = 15 / 8
(3 / 4) * (5 / 2) = 15 / 8
15 / 8
15 / 8
15 / 8
```

Deci, se observă că afişarea operaţiei s-a făcut de două ori, pentru ambele expresii. Funcţiile virtuale, ca şi alte funcţii membru de fapt, nu trebuie neapărat supraîncărcate în clasele derivate. Dacă nu sunt supraîncărcate atunci se moşteneşte funcţia membru de la un nivel superior.

Determinarea funcţiilor membru virtuale corespunzătoare se face pe baza unor tabele construite şi gestionate în mod automat. Obiectele claselor care au funcţii membru virtuale conţin şi un pointer către tabela construită. De aceea gestionarea funcţiilor membru virtuale necesită mai multă memorie şi un timp de execuţie mai îndelungat.

1.3.4. Clase abstracte

În cazul unei ierarhii de clase mai complicate, clasa de bază poate avea nişte proprietăţi generale despre care ştim, dar nu le putem defini numai în clasele derivate. De exemplu să considerăm ierarhia de clase din Figura 1.3.

Observăm că putem determina nişte proprietăţi referitoare la clasele derivate. De exemplu greutatea medie, durata medie de viaţă şi viteza medie de deplasare. Aceste proprietăţi se vor descrie cu ajutorul unor funcţii membru. În principiu şi pentru clasa *animal* există o greutate medie, durată medie de viaţă şi viteză medie de deplasare. Dar aceste proprietăţi ar fi mult mai greu de determinat şi ele nici nu sunt importante pentru noi într-o generalitate de acest fel. Totuşi pentru o tratare generală ar fi bine, dacă cele trei funcţii membru ar fi declarate în clasa de bază şi definite în clasele derivate. În acest scop s-a introdus noţiunea de *funcţie membru virtuală pură*.

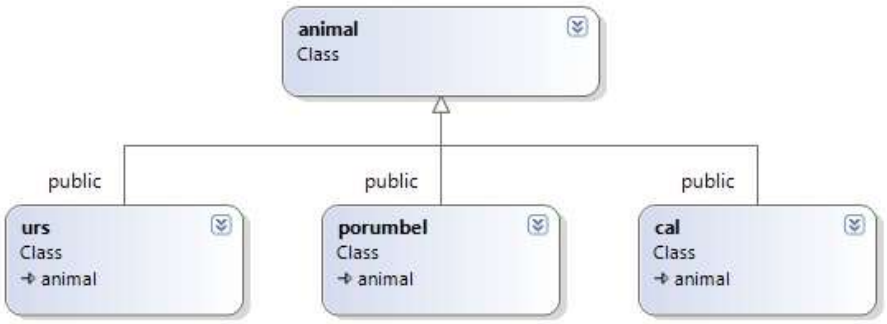


Figura 1.3. Ierarhie de clase referitoare la animale

Funcţia virtuală pură este o funcţie membru care este declarată, dar nu este definită în clasa respectivă. Ea *trebuie* definită într-o clasă derivată. Funcţia membru virtuală pură se declară în modul următor. Antetul obişnuit al funcţiei este precedat de cuvântul cheie *virtual*, şi antetul se termină cu *= 0*. După cum arată numele şi declaraţia ei, funcţia membru virtuală pură este o funcţie virtuală, deci selectarea exemplarului funcţiei din ierarhia de clase se va face în timpul execuţiei programului.

Clasele care conţin cel puţin o funcţie membru virtuală pură se vor numi *clase abstracte*.

Deoarece clasele abstracte conţin funcţii membru care nu sunt definite, nu se pot crea obiecte aparţinând claselor abstracte. Dacă funcţia virtuală pură nu s-a definit în clasa derivată atunci şi clasa derivată va fi clasă abstractă şi ca atare nu se pot defini obiecte aparţinând acelei clase.

Să considerăm exemplul de mai sus şi să scriem un program, care referitor la un *porumbel*, *urs* sau *cal* determină dacă el este gras sau slab, rapid sau încet, respectiv tânăr sau bătrân. Afişarea acestui rezultat se va face de către o funcţie membru a clasei *animal* care nu se supraîncarcă în clasele derivate. Fişierul **`abstract1.cpp`**:

```
#include <iostream>

using namespace std;

class animal {
protected:
  double greutate;  // kg
  double virsta;    // ani
  double viteza;    // km / h
public:
  animal( double g, double v1, double v2);
  virtual double greutate_medie() = 0;
  virtual double durata_de_viata_medie() = 0;
  virtual double viteza_medie() = 0;
  int gras() { return greutate > greutate_medie(); }
  int rapid() { return viteza > viteza_medie(); }
  int tanar()
    { return 2 * virsta < durata_de_viata_medie(); }
  void afiseaza();
};

animal::animal( double g, double v1, double v2)
{
  greutate = g;
  virsta = v1;
  viteza = v2;
}

void animal::afiseaza()
{
  cout << ( gras() ? "gras, " : "slab, " );
  cout << ( tanar() ? "tanar, " : "batran, " );
  cout << ( rapid() ? "rapid" : "incet" ) << endl;

}

class porumbel : public animal {
public:
  porumbel( double g, double v1, double v2):
    animal(g, v1, v2) {}
```

```
  double greutate_medie() { return 0.5; }
  double durata_de_viata_medie() { return 6; }
  double viteza_medie() { return 90; }
};


class urs: public animal {
public:
  urs( double g, double v1, double v2):
    animal(g, v1, v2) {}
  double greutate_medie() { return 450; }
  double durata_de_viata_medie() { return 43; }
  double viteza_medie() { return 40; }
};

class cal: public animal {
public:
  cal( double g, double v1, double v2):
    animal(g, v1, v2) {}
  double greutate_medie() { return 1000; }
  double durata_de_viata_medie() { return 36; }
  double viteza_medie() { return 60; }
};

int main() {
  porumbel p(0.6, 1, 80);
  urs u(500, 40, 46);
  cal c(900, 8, 70);
  p.afiseaza();
  u.afiseaza();
  c.afiseaza();
  return 0;
}
```

Observăm că deşi clasa *animal* este clasă abstractă, este utilă introducerea ei, pentru că multe funcţii membru pot fi definite în clasa de bază şi moştenite fără modificări în cele trei clase derivate.

1.3.5. Interfeţe

În limbajul C++ nu s-a definit noţiunea de interfaţă, care există în limbajele Java sau C#. Dar orice clasă abstractă, care conţine numai funcţii virtuale pure, se poate considera o interfaţă. Bineînţeles, în acest caz nu se vor declara nici date membru în interiorul clasei. Clasa abstractă *animal* conţine atât date membru, cât şi funcţii membru nevirtuale, deci ea nu se poate considera ca şi un exemplu de interfaţă.

În continuare introducem o clasă abstractă *Vehicul*, care nu conţine numai funcţii membru virtuale pure, şi două clase derivate din această clasă abstractă. Fişierul **`vehicul.cpp`**:

```
#include <iostream>
using namespace std;

class Vehicul
{
public:
     virtual void Porneste() = 0;
```

```
     virtual void Opreste() = 0;
     virtual void Merge(int km) = 0;
     virtual void Stationeaza(int min) = 0;
};

class Bicicleta : public Vehicul
{
public:
     void Porneste();
     void Opreste();
     void Merge(int km);
     void Stationeaza(int min);
};

void Bicicleta::Porneste() {
     cout << "Bicicleta porneste." << endl;
}
void Bicicleta::Opreste() {
     cout << "Bicicleta se opreste." << endl;
}
void Bicicleta::Merge(int km) {
     cout << "Bicicleta merge " << km <<
          " kilometri." << endl;
}
void Bicicleta::Stationeaza(int min) {
     cout << "Bicicleta stationeaza " << min <<
          " minute." << endl;
}

class Masina : public Vehicul
{
public:
     void Porneste();
     void Opreste();
     void Merge(int km);
     void Stationeaza(int min);
};

void Masina::Porneste() {
     cout << "Masina porneste." << endl;
}
void Masina::Opreste() {
     cout << "Masina se opreste." << endl;
}
void Masina::Merge(int km) {
     cout << "Masina merge " << km <<
          " kilometri." << endl;
}
void Masina::Stationeaza(int min) {
     cout << "Masina stationeaza " << min <<
          " minute."  << endl;
}

void Traseu(Vehicul *v)
{
     v->Porneste();
     v->Merge(3);
     v->Stationeaza(2);
     v->Merge(2);
     v->Opreste();
}
```

```
int main()
{
     Vehicul *b = new Bicicleta;
     Traseu(b);
     Vehicul *m = new Masina;
     Traseu(m);
     delete m;
     delete b;
}
```

În funcția *main* s-au declarat două obiecte dinamice de tip *Bicicleta,* respectiv *Masina,* și în acest fel, apelând funcția *Traseu* obținem rezultate diferite, deși această funcție are ca parametru formal numai un pointer către o clasă abstractă *Vehicul*.

1.4. Diagrame de clase şi interacţiuni între obiecte în UML: Pachete, clase şi interfețe. Relații între clase şi interfețe. Obiecte. Mesaje

Limbajul de modelare unificat UML (Unified Modelling Language) [29] defineşte un set de *elemente de modelare* şi *notaţii grafice* asociate acestora. Elementele de modelare pot fi folosite pentru descrierea oricăror sisteme software. În particular, UML conţine elemente ce pot fi folosite şi pentru cele orientate pe obiecte.

Această secţiune conţine câteva elemente de bază folosite pentru descrierea **structurii** şi **comportamentului** unui sistem software orientat pe obiecte - *diagrame de clase* şi de *interacţiuni între obiecte*. Aceste elemente corespund selecţiei facute în [30], capitolele 3 şi 4.

Înainte de a trece la prezentarea acestor elemente selectate, o să indicăm contextul în care acestea se folosesc pentru dezvoltarea unui sistem software. Întrebările principale la care trebuie să răspundem în acest sens sunt: (A) **ce tipuri de modele** construim, (B) **când sunt construite modelele** în diferite procese de dezvoltare şi (C) care este **legătura dintre modele şi codul scris**.

Pentru a răspunde pe scurt acestor întrebări, vom da exemple ce se referă la o aplicaţie folosită de un casier pentru înregistrarea vânzărilor la un punct de vânzare într-un magazin. Aplicaţia este numită **POS** (Point of Sale) şi presupune implementarea unui singur caz de utilizare, înregistrarea unei vânzări.

A. Tipuri de modele

Privind tipurile de modele pe care le construim, cel mai potrivit este să folosim termenii şi conceptele introduse de arhitecturile dirijate de modele, în principal **Model-Driven Architecture - MDA** [31]. Conform ghidului MDA, modele pe care le putem construi sunt prezentate în continuare.

CIM - Modele independente de calcule (Computational Independent Models). Aceste modele descriu **ce face** sistemul şi **nu cum furnizează acest comportament.** Ele se mai numesc şi **modele ale domeniului** (sau **modele de afaceri - business models**) şi descriu spaţiul problemei. Din perspectivă structurală, diagramele de clase sunt folosite pentru a defini **conceptele domeniului**. Diagramele de interacţiune sunt rar folosite în cazul acestor

modele. Pentru a exprima comportamentul dorit al sistemului, se folosesc alte elemente ca şi cazuri de utilizare, procese de afaceri (business processes), etc - dar acestea nu sunt discutate în această secţiune.

Primul model din **Figura 1.4** prezintă un extras din modelul conceptual pentru aplicaţia POS. Modelul este construit pentru a surprinde conceptele folosite de utilizatori. Aceste concepte sunt folosite pentru a exprima comportamentul dorit, folosind alte elemente de modelare.

PIM - Modele independente de platformă (Platform Independent Models). Aceste modele descriu **cum funcţionează sistemul,** într-o manieră independentă de posibilele platforme concrete în care va fi implementat. La acest nivel se introduc **elemente arhitecturale**, iar diagramele de clase şi de interacţiuni între obiecte constituie două instrumente de bază pentru descrierea detaliată a sistemului (**proiectare detaliată**). Desigur, se folosesc si alte elemente de modelare, structurale şi comportamentale, dar acestea nu sunt discutate aici, de exemplu - colaborări, maşini cu stări, activităţi, etc.

Al doilea model din **Figura 1.4** prezintă un extras din modelul PIM pentru *POS*. Atât modelele CIM cât şi cele PIM conţin doar construcţii UML (ex. tipuri de date definite în specificaţia UML) şi eventual extensii ale acestui limbaj (independente de platformă).

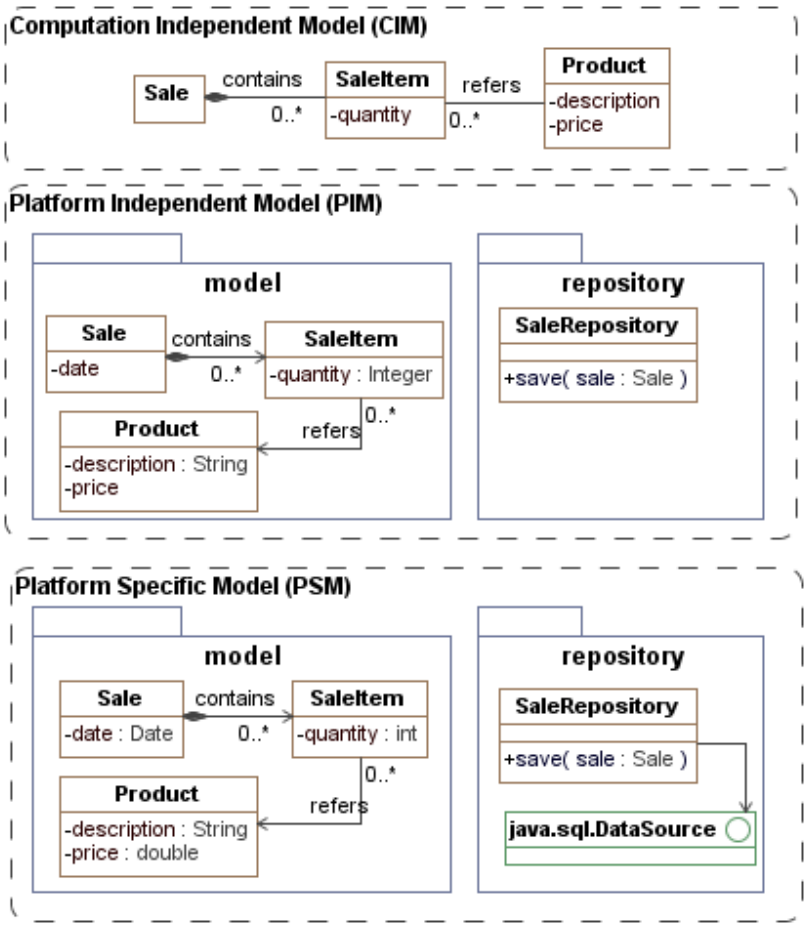


Figura 1.4 Tipuri de modele

PSM - Modele specifice unor platforme (Platform Specific Models). Acestea sunt o transformare a celor *PIM* către diferite platforme alese. Arhitectul poate decide construirea unui astfel de model pentru a exprima diferite elemente folosite ale platformelor alese, de

exemplu tipuri de date specifice. Diagramele de clase şi de interacţiune între obiecte sunt de asemenea folosite pentru aceste modele.

Ultimul model prezentat în **Figura 1**.4 reprezintă o transformare a modelului PIM în contextul implementării sistemului în Java. Tipurile de date din acest model sunt tipuri Java (ex. String, double), iar modelul include şi un tip de date din pachetul java.sql.

Conform ghidului MDA, definim modele astfel încât în final să **generăm cod către anumite platforme alese**. Codul poate fi generat pornind de la modele PIM sau PSM. În procesul de generare se folosesc **corespondenţe între elementele modelelor şi elementele platformei alese**.

B. Procese de dezvoltare şi instrumente CASE

Diferite procese de dezvoltare indică folosirea unor modele de diferite tipuri, la diferite momente pe parcursul dezvoltării.

Procesele de dezvoltare dirijate de modele subscriu în general ghidului [31] şi crează modele PIM, opţional derivate din modele CIM. Apoi, din modele PIM, generează cod, folosind în mod opţional un modele intermediare PSM. Aceste procese de dezvoltare presupun folosirea unor instrumente de proiectare (CASE - Computer Aided Software Engineering) care suportă această infrastructură de transformare a modelelor. Exemple în acest sens sunt toate procesele dirijate de modele pentru aplicaţii orientate pe servicii folosesc limbaje/extensii ale UML independente de platformă, de exemplu SoaML[1].

Există **procese dirijate de modele care nu folosesc modele PIM ci direct modele PSM**. Acestea se bazează pe specificaţii elaborate pentru anumite platforme, de exemplu arhitecturi bazate pe componente ce oferă servicii, SCA[2].

Procesele de dezvoltare mai elaborate de tipul RUP[3] (pentru sisteme mari), recomandă folosirea tuturor modelelor **CIM, PIM şi PSM**, în contextul folosirii instrumentelor CASE.

Procesele de dezvoltare de tip agil nu recomandă în general folosirea instrumentelor CASE, de exemplu dezvoltarea dirijată de teste[4] sau dezvoltarea agilă dirijată de modele[5], dar recomandă construirea unor modele înainte de a începe codificarea. **Modelele sunt de fapt schiţe** (scrise pe hârtie sau pe tablă) şi sunt folosite pentru a comunica idei despre proiectarea sistemului.

Indiferent de procesul de dezvoltare folosit, majoritatea instrumentelor de dezvoltare moderne permit **sincronizarea imediată între codul scris şi modelele asociate** codului. Această sincronizare este de fapt între cod şi modele PSM. De exemplu, un instrument CASE care sincronizează modelele cu codul scris în Java, o face între cod şi modele PSM conform platformei Java.

O ultimă şi recentă categorie de procese de dezvoltare care propune folosirea modelelor **PIM** şi **generarea directă şi completă a codului** este categoria proceselor ce se bazează pe

---

[1] OMG. *Service Oriented Architecture Modeling Language*, 2009. http://www.omg.org/spec/SoaML/
[2] Open SOA. *Service Component Architecture Specifications*, 2007. http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
[3] IBM. *IBM Rational Unified Process*, 2007. http://www-01.ibm.com/software/awdtools/rup/
[4] Beck, K. *Test-Driven Development by Example*, Addison Wesley, 2003.
[5] Ambler, S.W. *Agile Model Driven Development (AMDD): The Key to Scaling Agile Software Development*, 2008. http://www.agilemodeling.com/essays/amdd.htm

modele executabile. Astăzi, adoptarea standardului pentru modele executabile UML (fUML - Foundational UML)[6] este în curs de finalizare. Conform acestor procese, în viitorul apropiat ne aşteptăm la adoptarea unui **stil nou de dezvoltare** în care vom construi doar **modele** şi vom scrie **cod într-un limbaj textual**[7] **definit pe elementele din aceste modele**. Astfel, modelele PSM şi codul scris în limbaje ca şi Java, C++ sau C# vor fi lăsate în grija instrumentelor CASE care le vor genera automat.

C. Corespondenţa dintre modele şi cod

Corespondenţele dintre modele şi cod sunt importante după cum relevă punctele (A) şi (B). Dacă generăm cod din modele PIM, respectiv dacă folosim un instrument CASE care sincronizează modele PSM cu codul, e important sa ştim ce elemente se vor genera din modelele construite. Chiar dacă lucrăm agil şi folosim "schiţe" de modele (fără a folosi instrumente CASE), se pune aceeaşi problemă.

Ţinând cont şi de modelele executabile amintite anterior (care sunt la nivel PIM), în secţiunile care urmează vom discuta numai **corespondenţele dintre modele PIM** şi **limbajele C++**, **Java** şi **C#**. Modelele PSM conţin în plus faţă de cele PIM şi tipuri de date specifice anumitor limbaje, astfel corespondenţele dintre modele PSM şi cod sunt aceleaşi, doar că sunt prezente în modele şi extensii UML conform tipurilor specifice.

1.4.1. Diagrame de clase

Diagramele sunt reprezentări grafice (în general 2D) ale unor elemente dintr-un model. **Diagramele de clase** reprezintă tipurile de obiecte folosite în sistem şi relaţiile dintre acestea. Elementele structurale selectate în această secţiune sunt (a) **tipuri de obiecte**: clase, interfeţe, enumerări; (b) **gruparea elementelor** folosind pachete şi (c) **relaţii între aceste elemente**: asocieri, generalizări, realizări şi dependenţe.

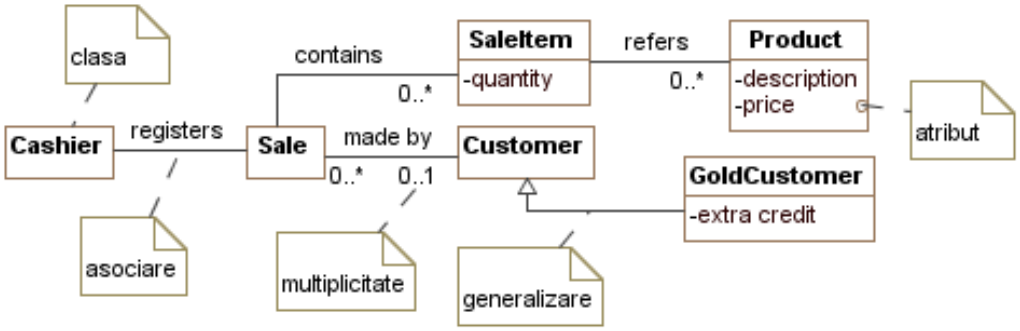


Figura 1.5 Model conceptual

Figura 1.5 prezintă un model conceptual iniţial pentru *POS*. Clasele sunt folosite pentru a identifica conceptele acestui domeniu. Acolo unde nu e relevant, compartimentul cu atributele claselor este ascuns. Proprietăţile claselor sunt definite prin atribute şi asocieri, iar tipurile de date pentru atribute nu sunt precizate.

[6] OMG. Semantics Of A Foundational Subset For Executable UML Models (FUML), 2010. http://www.omg.org/spec/FUML/

[7] OMG. Concrete Syntax For UML Action Language (Action Language For Foundational UML - ALF), 2010. http://www.omg.org/spec/ALF/

Modelele conceptuale sunt de tip CIM şi sunt folosite pentru a genera modele PIM. Ca şi modele CIM, ele pot să nu conţină detalii privind reprezentarea atributelor. Dacă procesul de dezvoltare folosit nu presupune folosirea unui model CIM (ci PIM sau PSM), atunci modelul din **Figura 1.5** este un model PIM sau PSM incomplet.

În context PIM, arhitectura sistemului din perspectivă structurală este descrisă folosind pachete (organizate ierarhic) şi dependenţe între acestea - a se vedea **Figura 1.6** pentru *POS*. Pachetele sunt definite cu responsabilităţi coezive, de exemplu interacţiunea cu utilizatorul (*ui*), faţadă peste domeniu (*service*), entităţi (*model*) şi depozite de obiecte sau obiecte de acces la date (*repository, inmemory repository*).

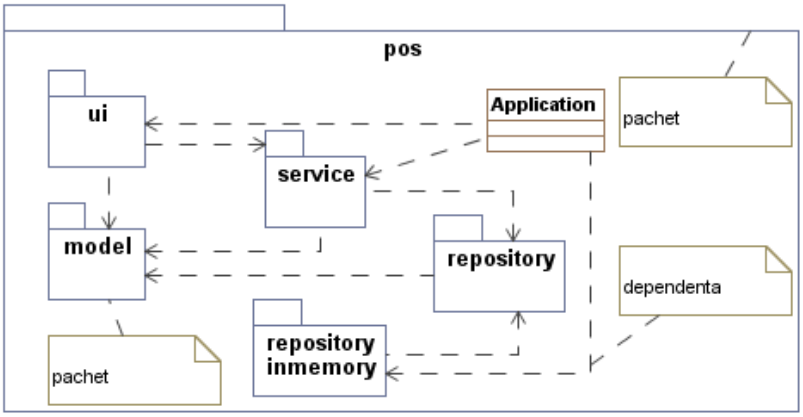


Figura 1.6 Arhitectură stratificată

Grija principală pe care o avem atunci când stabilim arhitectura sistemului este să respectăm principiile orientate pe obiecte SOLID[8,9]. Pachetele din **Figura 1.6** sunt proiectate conform principiului responsabilităţii unice (Single Responsibility), conform căruia obiectele ar trebui să aibă o singură responsabilitate, iar obiectele cu responsabilităţi înrudite ar trebui grupate logic.

Dependenţa dintre două elemente software (*A* depinde de *B*) indică faptul că atunci când un element se va modifica (*B*), este posibil ca elementul dependent (*A*) să trebuiască de asemenea modificat. Dependenţele dintre pachetele din **Figura 1.6** respectă recomandările arhitecturilor stratificate, adică elementele de pe straturile superioare sunt dependente de cele de pe straturile inferioare, de exemplu *ui* depinde de *service* şi *model*, *service* depinde de *model* şi *repository*, însă service nu depinde de implementarea concretă pentru repository, anume *repository inmemory*. Inversarea acestei ultime dependenţe urmează un alt principiu SOLID, anume inversarea dependenţelor (Dependency Inversion). **Figura 1.7** prezintă detalii privind această inversare a dependenţelor dintre *service* şi *repository inmemory*. În loc ca *StoreService* să fie dependent de implementarea concretă *InmemorySaleRepository*, a fost introdusă interfaţa *SaleRepository* pentru a decupla aceste două elemente. De fapt *SaleRepository* abstractizează accesul la obiectele de tip *Sale*, făcând posibilă astfel înlocuirea pachetului *repository inmemory* din sistem cu o altă implementare, fără a afecta celelalte pachete din sistem.

[8] Robert C. Martin. *Design Principles and Design Patterns*, 2004. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[9] SOLID Design Principles: *Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion*, http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

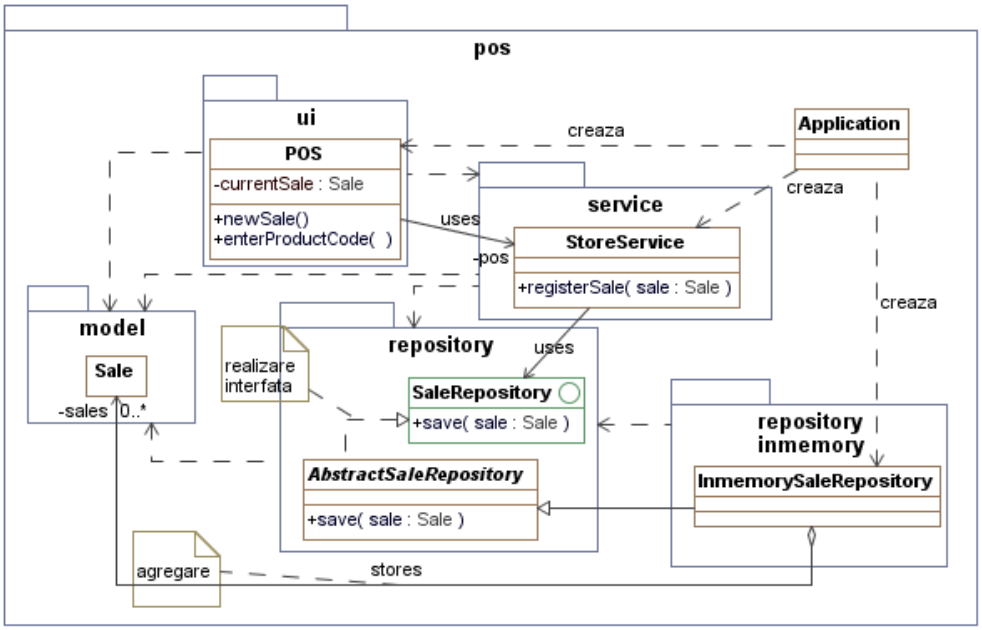


Figura 1.7 Arhitectură stratificată - justificare dependenţe

La nivel PIM sau PSM, diagramele de clase sunt folosite pentru a rafina entităţile şi relaţiile dintre acestea - a se vedea **Figura 1.8**. Aceste elemente vor fi discutate în subsecţiunile care urmează.

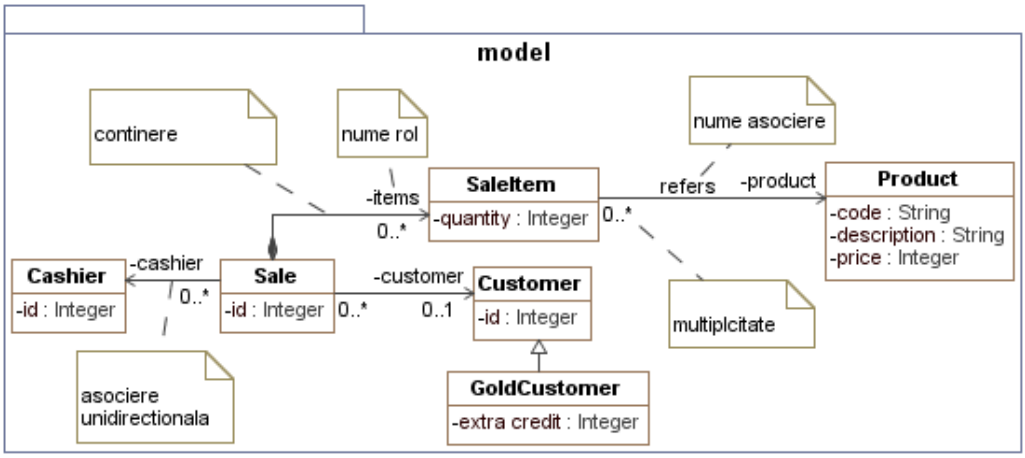


Figura 1.8 Entităţile POS

A. Pachete

Pachetele UML [29, 30] grupează elemente din model şi oferă un spaţiu de nume pentru elementele grupate. Din perspectiva elementelor discutate în acest document, pachetele pot conţine tipuri de date şi alte pachete. Un tip de date sau un pachet poate fi parte a unui singur pachet.

În termeni ai limbajelor de programare, pachetele UML corespund pachetelor Java şi spaţiilor de nume C++ şi C#. Pachetele UML sunt referite folosind operatorul de rezoluţie ::,

la fel ca şi in C++ şi C#. De exemplu, numele complet al pachetului *ui* din **Figura 1.6** sau **Figura 1.7** este *pos*::*ui*.

Diagramele de clase ce indică cu preponderenţă pachetele unui sistem sunt folosite pentru a descrie arhitectura acestuia - a se vedea **Figura 1.6** pentru sistemul *POS*. Dependenţele între pachete indică un sumar al dependenţelor dintre elementele conţinute şi elementele din alte pachete. Din perspectivă arhitecturală, buna gestionare a acestor dependenţe este crucială în procesul de construire şi întreţinere a sistemului.

B. Clase

O clasă UML [29, 30] reprezintă o mulţime de obiecte cu aceleaşi elemente structurale (proprietăţi) şi comportamentale (operaţii). Clasele UML sunt tipuri de date şi corespund claselor din limbajele Java, C++ şi C#. O clasă poate fi declarată **abstractă** şi în acest caz nu poate fi instanţiată la fel ca şi în Java, C++ şi C#.

O clasă UML poate fi **derivată** din mai multe clase, la fel ca şi în C++. Folosirea moştenirii multiple în model nu duce la o corespondenţă directă între model şi cod în cazul limbajelor Java sau C#.

O clasă UML poate **realiza/implementa** mai multe interfeţe la fel ca şi în Java sau C#. Corespondenţa între modelele ce conţin clase ce implementează mai multe interfeţe şi C++ este realizată via clase C++ pur abstracte şi moştenire multiplă.

Toate clasele din **Figura 1.8**sunt concrete, iar *AbstractSaleRepository* din **Figura 1.7** este clasă abstractă (numele scris italic).

Principiul substituţiei este aplicabil pentru instanţele de tipul unor clase şi interfeţe, la fel ca şi în Java, C++ şi C#. Adică, instanţele din program pot fi înlocuite cu instanţe ale tipurilor derivate fără să alterăm semantic programul.

C. Interfeţe

O interfaţă UML [29, 30] este un tip de date ce declară un set de operaţii, adică un contract pe care clasele pot să-l realizeze. Acest concept corespunde aceluiaşi concept din Java/C# şi claselor pur abstracte din C++.

SaleRepository din Figura **1.7** este o interfaţă. Atunci când evidenţierea metodelor interfeţei nu este relevantă, notaţia grafică pentru interfeţe este cea din Figura **1.9**.

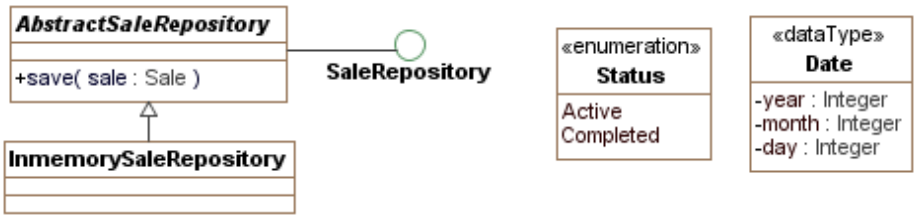


Figura 1.9 Interfaţă, enumerare şi tipuri structurate

D. Enumerări şi obiecte valorice

Enumerările UML [29, 30] descriu un set de simboluri care nu au asociate valori aşa cum aceleaşi concepte se regăsesc în C++, Java şi C#.

Tipurile structurate [29, 30] se modelează folosind stereotipul *datatype* şi corespund structurilor din C++/C# şi tipurilor primitive din Java - a se vedea **Error! Reference source not found.**. Instanţele acestor tipuri sunt identificate doar prin valoarea lor. Ele sunt folosite pentru a descrie proprietăţile claselor şi corespund obiectelor valorice (şablonul value object[10]), cu deosebirea că nu pot avea identitate.

E. Generalizări şi realizări de interfeţe

Generalizarea [29, 30] este o relaţie între un tip de date mai general (de bază) şi unul mai specializat (derivat). Această relaţie poate fi aplicată între două clase sau două interfeţe, corespunzând relaţiilor de moştenire din Java şi C++/C# dintre clase, respectiv interfeţe (clase pur abstracte în cazul C++).

Realizarea unei interfeţe în UML [29, 30] reprezintă o relaţie între o clasă şi o interfaţă prin care se indică faptul că clasa este conformă contractului specificat de interfaţă. Aceste realizări corespund implementărilor interfeţelor din Java şi C#, respectiv moştenirii în C++. A se vedea notaţiile grafice dintre *AbstractSaleRepository* şi *SaleRepository* în **Figura 1.7** şi **Figura** 1.9.

F. Proprietaţi

Proprietăţile [29, 30] reprezintă aspecte structurale ale unui tip de date. Proprietăţile unei clase sunt introduse prin atribute şi asocieri. Un **atribut** descrie o proprietate a clasei în al doilea compartiment al ei, sub forma:

vizibilitate nume: *tip multiplicitate = valoare* {*proprietati*}

Numele este obligatoriu, la fel ca şi **vizibilitatea** care poate fi publica (+), privată (-), protetejată (#) sau la nivel de pachet (fără specificator). Vizibilitatea UML corespunde specificatorilor de acces cu acelaşi nume din Java, având aceeaşi semantică. Vizibilitatea la nivel de pachet nu se regaseşte în C++, iar în C# are o corespondenţă prin specificatorul internal din C# dar care are şi conotaţii de distribuire a elementelor software (elementele declarate internal în C# fiind accesibile doar în distribuţia binară dll sau exe din care acestea fac parte).

Celelalte elemente folosite la declararea unei proprietăţi sunt opţionale. **Tipul** proprietăţii poate fi oricare: clasă, interfaţă, enumerare, tip structurat sau tip primitiv. **Tipurile primitive în UML** sunt tipuri valorice [29]. UML defineşte următoarele tipuri primitive: String, Integer, Boolean şi UnlimitedNatural. Primele trei tipuri primitive sunt în corespondenţă cu tipurile cu acelaşi nume din limbajele Java, C++ şi C#, dar cu observaţiile:

- Tipul String este clasă în Java şi C#, instanţele de tip String fiind nemodificabile, spre deosebire de C++ unde şirurile de caractere sunt modificabile. Codificarea caracterelor nu este precizată în UML, în timp ce în Java şi C# ea este Unicode, iar în C++ ASCII.
- Tipul Integer în UML este în precizie nelimitată, în timp ce în cele 3 limbaje plaja de valori este limitată.

Multiplicitatea poate fi 1 (valoare implicită, atunci când multiplicitatea nu e precizată), 0..1 (optional), 0..* (zero sau mai multe valori), 1..* (unu sau mai multe valori), m..n (între *m* şi *n* valori, unde *m* şi *n* sunt constante, *n* putând fi *). Pentru o proprietate cu multiplicitate ***m..**** putem preciza în plus dacă:

[10] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

- Valorile se pot repeta sau nu - **implicit valorile sunt unice** (adică mulţime), în caz contrar precizăm explicit prin **nonunique** (adică container cu valori posibil duplicate).
- Valorile pot fi referite prin indici sau nu - **implicit nu** (deci colecţie), în caz contrar precizăm explicit **ordered** (deci listă).

Exemple de proprietăţi:
multime : Integer[0..]* - mulţime de valori întregi (unice)
lista : Integer[0..] {ordered}* - listă cu valori întregi şi distincte (unice)
lista : Integer[0..] {ordered, nonunique}* - listă de întregi
colectie : Integer[0..] {nonunique}* - colecţie de întregi

Proprietăţilor din UML le corespund câmpuri sau variabile de tip obiect în Java şi C++, respectiv proprietăţi în C#. Dificultăţi de interpretare se ridică în ceea ce priveşte proprietăţile cu multiplicitate *m..**. Pentru exemplele de mai sus putem considera următoarele corespondenţe cu Java (in mod similar şi cu C++/C#):

- mulţimi de întregi:
  - *int[] multime* sau *Integer[] multime*, urmând să asigurăm prin operaţii că *multime* va conţine valori distincte, sau cel mai potrivit
  - *java.util.Set multime*
- liste cu valori întregi şi distincte:
  - *int[] lista, Integer[] lista* sau *java.util.List lista,* urmând să asigurăm prin operaţii că *lista* va conţine valori distincte
- liste de întregi:
  - *int[] lista, Integer[] lista,* sau *java.util.List lista*
- colecţii de întregi:
  - *int[] colectie, Integer[] colectie,* sau *java.util.Collection colectie*

Asocierile UML [29, 30] reprezintă un set de tuple, fiecare tuplu făcând legătura între două instanţe ale unor tipuri de date. În acest sens, o asociere este un tip de date care leagă proprietăţi ale altor tipuri de date.

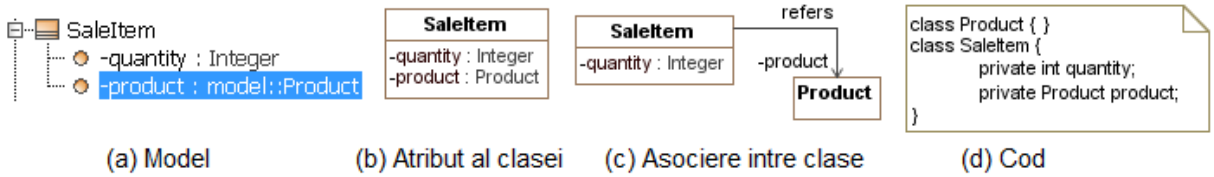


Figura 1.10 Asocieri unidirecţionale

Figura 1.10 (a) prezintă modelul rezultat după adăugarea atributelor *quantity* şi *product* în clasa *SaleItem* reprezentată grafic în diagrama (b). Codul (d) scris în Java/C# corespunde acestei situaţii. Dacă considerăm că e mai potrivită o reprezentare grafică pentru relaţia dintre clasele *SaleItem* şi *Product*, atunci în loc să adăugăm *product* ca şi atribut, folosim o asociere unidirecţională de la *SaleItem* spre *Product*. Atunci când se adaugă asocierea unidirecţională, în model se creează o asociere şi o proprietate în clasa *SaleItem*, având numele rolului, adică *product*. Astfel, codul (d) corespunde reprezentării grafice (c) a modelului (a) care mai conţine o asociere nearătată în figură. **Asocierile unidirecţionale** introduc proprietăţi în clasa sursă, de tipul clasei destinaţie. Numele proprietăţii coincide cu numele rolului asocierii, iar forma generală de definire a proprietăţilor (prezentată la începutul acestei subsecţiuni) se aplică şi în acest caz.

Decizia folosirii asocierilor în locul atributelor este luată în funcţie de context. De exemplu, atunci când modelăm entităţile unui aplicaţii folosim asocieri pentru a indica relaţiile dintre entităţi şi folosim atribute atunci când descriem entităţile folosind obiecte valorice/descriptive. În general, folosim asocieri când dorim să evidenţiem importanţa tipurilor şi a legăturilor dintre ele.

Asocierile bidirecţionale leagă două proprietăţi din două clase diferite sau din aceeaşi clasă. **Figura 1.11** prezintă o asociere bidirecţională între *SaleItem* şi *Product*, precum şi codul Java/C# corespunzător acestei situaţii.

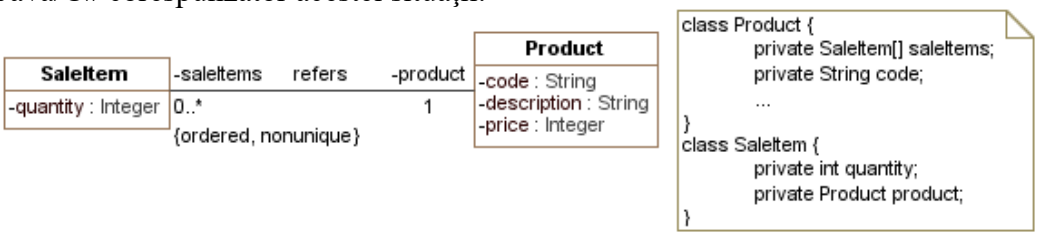


Figura 1.11 Asocieri bidirecţionale

Modelele conceptuale conţin asocieri bidirecţionale. Păstrarea asocierilor bidirecţionale în modelele PIM/PSM poate conduce la o execuţie ineficientă datorată reprezentării obiectelor. Un pas obligatoriu ce trebuie făcut în cadrul proiectării detaliate este **rafinarea asocierilor**, în primul rând prin **transformarea celor bidirecţionale în unidirecţionale**. **Figura 1.8** prezintă rezultatul rafinării asocierilor din **Figura 1.5**.

Relaţiile întreg-parte sunt modelate în UML folosind agregări şi conţineri. O **agregare** este o asociere prin care indicăm că un obiect este parte a unui alt obiect. De exemplu, **Figura 1.7** indică prin agregarea dintre *InmemorySaleRepository* şi *Sale* că primul obiect memorează toate obiectele de tip *Sale* (vânzările sunt parte a acestui depozit). O **conţinere** este o agregare prin care se indică în plus că obiectele conţinute pot fi părţi a unui singur întreg, de exemplu, un element al vânzării (*SaleItem*) în **Figura 1.8** poate fi parte doar dintr-o singură vânzare (*Sale*). **Rafinarea asocierilor** include şi stabilirea relaţiilor de **agregare** şi **conţinere**.

Ca şi în Java, C++ şi C#, putem defini **proprietăţi statice sau de tip clasă**, în diagrame acestea fiind reprezentate prin subliniere.

G. Dependenţe

Între două elemente software, *client* şi *furnizor*, există o **dependenţă** [29, 30] dacă schimbarea definiţiei *furnizorului* poate duce la schimbarea *clientului*. De exemplu dacă o clasă *C* trimite un *mesaj* altei clase *F,* atunci *C* este dependentă de *F* deoarece schimbarea definiţiei mesajului în *F* va implica schimbări în *C* privind modul de transmitere. Ca regulă generală, ar trebui să minimizăm dependenţele în model, în timp ce păstrăm coezive aceste elemente.

În UML se pot indica explicit dependenţele între orice elemente din model. Prezentarea lor explicită însă poate face modelul greu de citit. Din acest motiv, dependenţele se prezintă explicit în mod selectiv, evidenţiind elementele principale şi arhitecturale - a se vedea **Figura 1.6** şi **Figura 1.7**.

H. Operaţii

Operaţiile în UML [29, 30] definesc comportamentul obiectelor şi corespund metodelor din limbajele de programare orientate obiect. De fapt, operaţiile specifică comportamentul (reprezintă antetul), iar corpul/implementarea este definită de elemente comportamentale ca şi interacţiuni, maşini cu stări şi activităţi - implementările sunt numite **metode** în UML. Sintaxa specificării operaţiilor este:

vizibilitate nume (*lista-parametri*) : *tip-returnat* {*proprietăţi*}

unde *vizibilitatea*, *tipul-returnat* şi *proprietăţile* sunt definite ca şi în cazul proprietăţilor claselor. În lista *proprietăţilor* operaţiei se poate preciza dacă este doar o operaţie de interogare *{query}*, adică o operaţie ce nu modifică starea obiectului apelant - implicit, operaţiile sunt considerate comenzi, adică modifică starea obiectelor. Parametrii în *lista-parametrilor* sunt separaţi prin virgulă, un parametru fiind de forma:

direcţie nume: *tip* = *valoare-implicită*,

direcţia putând fi: *in*, *out* şi *in-out*, implicit fiind *in*.

Ca şi în Java, C++ şi C#, putem defini **operaţii statice sau de tip clasă**, în diagrame acestea fiind reprezentate prin subliniere.

1.4.2. Diagrame de interacţiune

Interacţiunile UML [29, 30] descriu comportamentul sistemului, indicând cum colaborează mai mulţi participanţi într-un anume scenariu. Există mai multe tipuri de interacţiuni, dar în această secţiune vom discuta numai despre **diagrame de secvenţe**, un tip de interacţiuni care descriu mesajele transmise între participanţi.

În general diagramele de secvenţă descriu **un singur scenariu**. De exemplu, diagrama din **Figura** pune faţă în faţă casierul şi sistemul *POS*, descriind fluxul normal de desfăşurare pentru singurul caz de utilizare discutat aici, înregistrarea unei vânzări. O astfel de diagramă ajută la identificarea **interfeţei publice a sistemului**. Pornind de la descrierea cazurilor de utilizare, acţiunile utilizatorilor sunt modelate ca şi mesaje la care sistemul trebuie să răspundă. Mesajele 1, 2, 4 şi 6 din **Figura** 1.12 indică faptul că sistemul va trebui să facă anumite calcule şi să răspundă utilizatorului.

Bara verticală asociată unui participant reprezintă o **axa temporală**. Pe această axă se plasează **bare de activare** pentru a indica când anume acel participant este implicat în interacţiune. **Mesajele** au nume, sunt în general numerotate şi pot indica un **răspuns** returnat. Toate mesajele din figurile acestei secţiuni sunt sincrone (cele asincrone nu intră în scopul acestui document). Interacţiunile pot conţine **fragmente**: **cicluri** şi **alternative**.

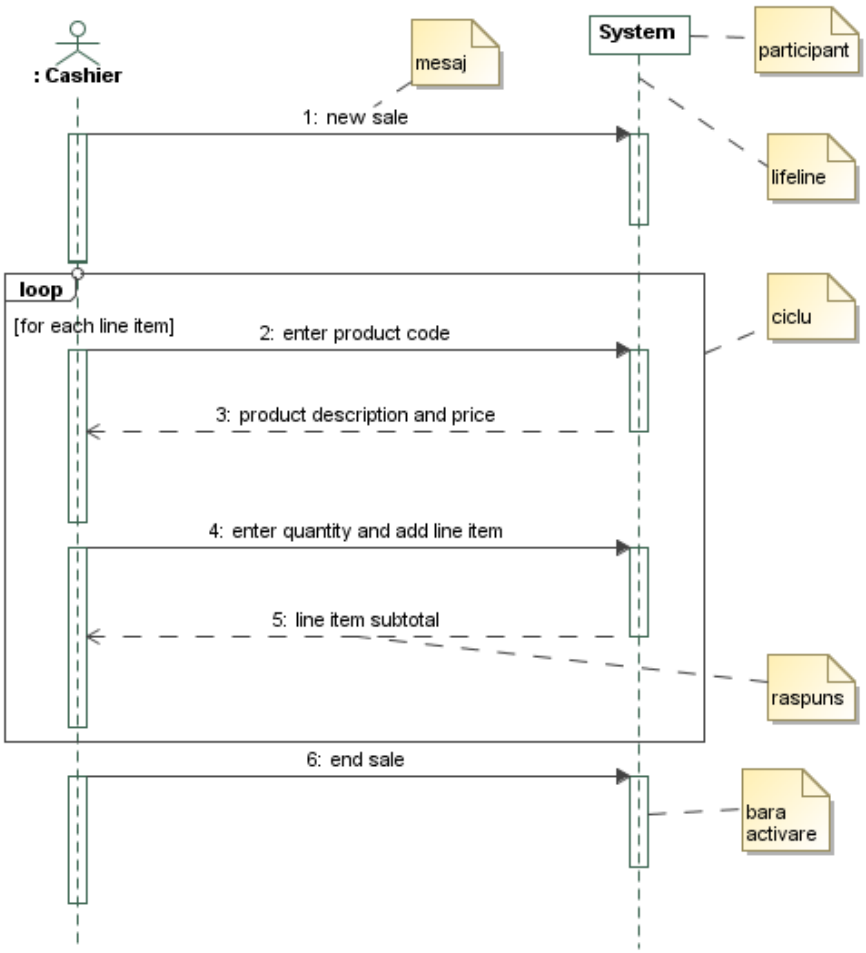


Figura 1.12 Interfaţa sistemului

Diagrame de tipul celei prezentate în **Figura** 1.12 pot fi definite în context CIM, înainte de stabilirea unei arhitecturi. La nivel PIM, odată identificat ceea **ce trebuie să facă sistemul**, se folosesc diagrame de secvenţă pentru a detalia **cum vor colabora obiectele din sistem**, conform responsabilităţilor precizate prin arhitectura stabilită. **Figura 1.13** prezintă detaliat colaborarea în cazul sistemului *POS*.

Participanţii din **Figura** 1.12 nu indică instanţe ale unor tipuri din model. De astă dată, participanţii principali din **Figura 1.13** sunt obiecte *controller*, *service* şi *repository*, conform arhitecturii *POS*. Diagrama prezintă mesaje adresate către controller (1, 3, 6 şi 9) deoarece în diagramă nu e prezent şi elementul de interfaţă utilizator.

Participanţii fiind obiecte, mesajele transmise vor fi apeluri de metode ale obiectelor spre care sunt trimise mesajele. Astfel, diagrama ne conduce la identificarea metodelor obiectelor. **Figura** 1.14 prezintă metodele identificate pe baza interacţiunilor din **Figura 1.13**.

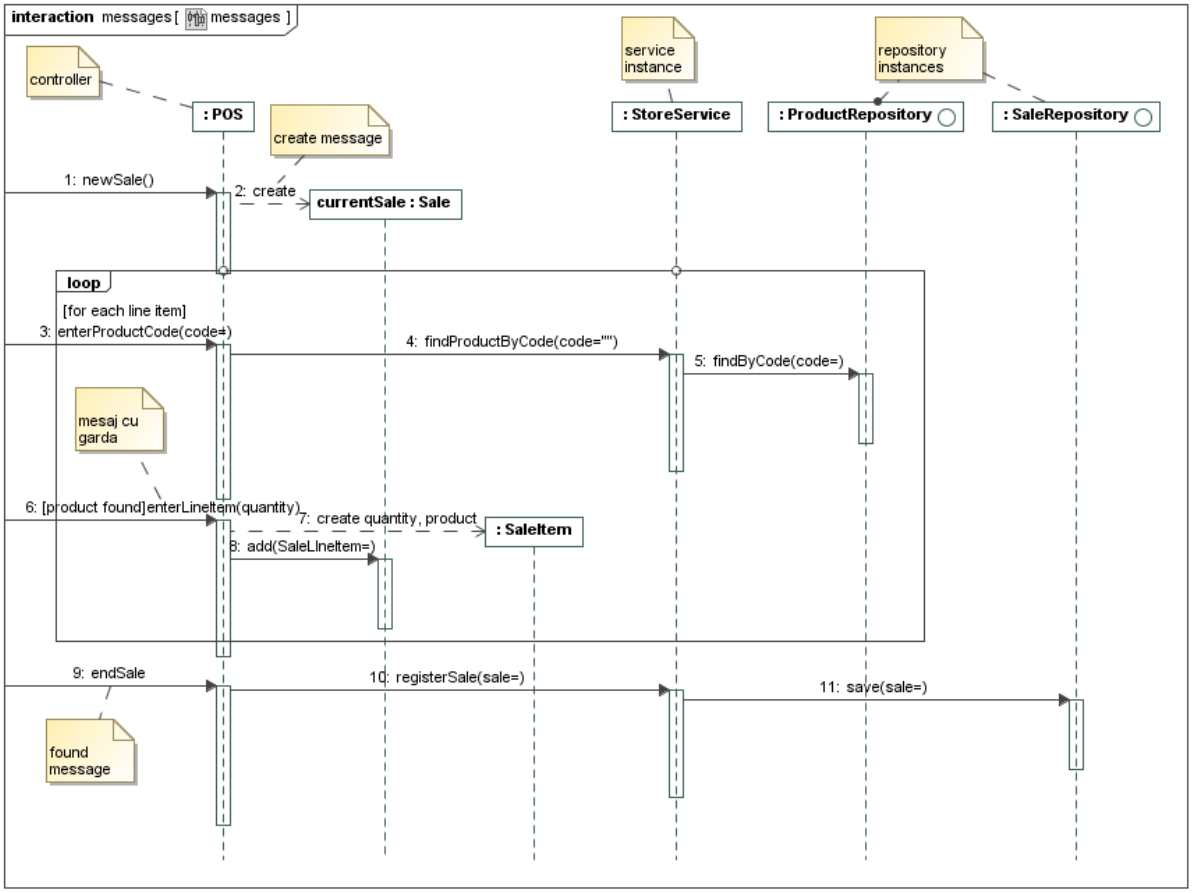


Figura 1.13 Proiectare detaliată - interacţiuni între obiecte

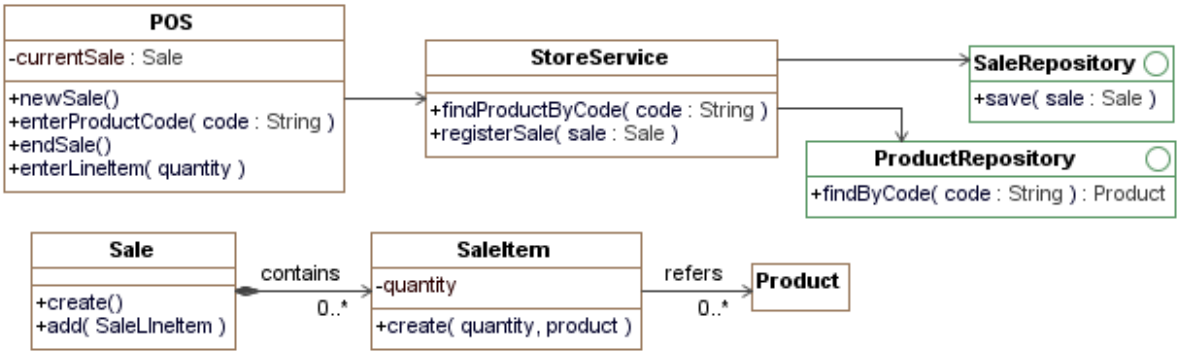


Figura 1.14 Proiectare detaliată - diagramă de clase

1.5. Liste şi dicţionare

In cele ce urmează vom prezenta două dintre containerele des folosite in programare şi anume ***listele*** şi ***dicţionarele***. Vom specifica tipurile abstracte de date corespunzătoare, indicând şi specificând operaţiile caracteristice. Pentru fiecare operaţie din interfaţa unui tip de date, vom da specificarea operaţiei în limbaj natural, indicând datele şi precondiţiile operaţiei (**pre**), precum şi rezultatele şi postcondiţiile operaţiei (**post**).

1.5.1. Liste

In limbajul uzual cuvântul "listă" referă o "înşirare, într-o anumită ordine, a unor nume de persoane sau de obiecte, a unor date etc." Exemple de liste sunt multiple: listă de cumpărături, listă de preţuri, listă de studenţi, etc. Ordinea în listă poate fi interpretată ca un fel de „legătură" între elementele listei (după prima cumpărătură urmează a doua cumpărătură, după a doua cumpărătură urmează a treia cumpărătură, etc) sau poate fi văzută ca fiind dată de numărul de ordine al elementului în listă (1-a cumpărătură, a 2-a cumpărătură, etc). Tipul de date `Listă` care va fi definit în continuare permite implementarea în aplicaţii a acestor situaţii din lumea reală.

Ca urmare, o ***listă*** o putem vedea ca pe o secvenţă de elemente $< l_1, l_2, .., l_n >$ de un acelaşi tip (`TElement`) aflate într-o anumită ordine, fiecare element având o *poziţie* bine determinată în cadrul listei. Ca urmare, poziţia elementelor în cadrul listei este esenţială, astfel accesul, ştergerea şi adăugarea se pot face pe orice poziţie în listă. Lista poate fi văzută ca o colecţie dinamică de elemente în care este esenţială ordinea elementelor. Numărul *n* de elemente din listă se numeşte *lungimea* listei. O listă de lungime 0 se va numi lista *vidă*. Caracterul de dinamicitate al listei este dat de faptul că lista îşi poate modifica în timp lungimea prin adăugări şi ştergeri de elemente în/din listă.

In cele ce urmează, ne vom referi la listele liniare. O listă liniară, este o structură care fie este vidă (nu are nici un element), fie

- are un unic prim element;
- are un unic ultim element;
- fiecare element din listă (cu excepţia ultimului element) are un singur succesor;
- fiecare element din listă (cu excepţia primului element) are un singur predecesor.

Ca urmare, într-o listă liniară se pot insera elemente, şterge elemente, se poate determina succesorul (predecesorul) unui element, se poate accesa un element pe baza ***poziţiei*** sale în listă.

O listă liniară se numeşte ***circulară*** dacă se consideră predecesorul primului nod a fi ultimul nod, iar succesorul ultimului nod a fi primul nod.

Conform definiţiei anterioare, fiecare element al unei listei liniare are o poziţie bine determinată în listă. De asemenea, este importantă prima poziţie în cadrul listei, iar dacă se cunoaşte poziţia unui element din listă atunci pe baza aceastei poziţii se poate identifica elementul din listă, poziţia elementului predecesor şi poziţia elementului succesor în listă (dacă acestea există). Ca urmare, într-o listă se poate stabili o ordine între poziţiile elementelor în cadrul listei.

Poziţia unui element în cadrul listei poate fi văzută în diferite moduri:

1. ca fiind dată de ***rangul*** (numărul de ordine al) elementului în cadrul listei. În acest caz este o similitudine cu tablourile, poziţia unui element în listă fiind indexul acestuia în cadrul listei. Într-o astfel de abordare, lista este văzută ca un tablou dinamic în care se pot accesa/adăuga/şterge elemente pe orice poziţie în listă.
2. ca fiind dată de o ***referinţă*** la locaţia unde se stochează elementul listei (ex: pointer spre locaţia unde se memorează elementul).

Pentru a asigura generalitatea, vom abstractiza noţiunea de ***poziţie*** a unui element în listă şi vom presupune că elementele listei sunt accesate prin intermediul unei ***poziţii*** generice.

Vom spune că o poziţie *p* într-o listă este ***validă*** dacă este poziţia unui element al listei. Spre exemplu, dacă *p* ar fi un pointer spre locaţia unde se memorează un element al listei, atunci *p* este ***valid*** dacă este diferit de pointerul nul sau de orice altă adresă care nu reprezintă adresa de memorare a unui element al listei. În cazul în care *p* ar fi rangul (numărul de ordine al) elementului în listă, atunci *p* este ***valid*** dacă nu depăşeşte numărul de elemente din listă.

Ca urmare, dacă ne gândim la o listă liniară în care operaţiile de acces/inserare/ştergere să se facă pe baza unei ***poziţii*** generice în listă, se ajunge la următorul tip abstract de date.

Lista vidă o vom nota în ceea ce urmează cu ϕ.

`Tipul Abstract de Date LISTA`

domeniu

$\mathcal{L}$={*l* | *l* este o listă cu elemente de tip `TElement`}

operaţii (interfaţa minimală)

creează(*l*)
descriere: se creează o listă vidă
pre: adevărat
post: $l \in \mathcal{L}$, $l = \phi$

adaugăSfarsit (*l*, *e*)
descriere: se adaugă un element la sfârşitul listei
pre: $l \in \mathcal{L}$, $e \in$ `TElement`
post: $l' \in \mathcal{L}$, *l'* este *l* în care a fost adăugat *e* la sfârşit

adaugăInceput(*l*, *e*)
descriere: se adaugă un element la începutul listei
pre: $l \in \mathcal{L}$, $e \in$ `TElement`
post: $l' \in \mathcal{L}$, *l'* este *l* în care a fost adăugat *e* la început

valid(*l*, *p*)
descriere: funcţie care verifică dacă o poziţie în listă este validă
pre: **l**$\in \mathcal{L}$, *p* e o *poziţie* în *l*
post: *valid*= adevărat dacă *p* este o poziţie validă în *l*
fals în caz contrar

adaugăÎnainte(*l*, *p*, *e*)
descriere: se adaugă un element înaintea unei anumite poziţii în listă

pre: $l \in \mathcal{L}$, $e \in$ `TElement`, p e o *poziţie* în l, ***valid***(l, p)

post: $l' \in \mathcal{L}$, l' este l în care a fost inserat e înainte de poziţia p

adaugăDupă(l, p, e)

descriere: se adaugă un element după o anumită poziţie în listă

pre: $l \in \mathcal{L}$, $e \in$ `TElement`, p e o *poziţie* în l, ***valid***(l, p)

post: $l' \in \mathcal{L}$, l' este l în care a fost inserat e după poziţia p

şterge (l, p, e)

descriere: se şterge elementul din listă situat pe o anumită poziţie

pre: $l \in \mathcal{L}$, $e \in$ `TElement`, p e o *poziţie* în l, ***valid***(l, p)

post: $e \in$ `TElement`, $l' \in \mathcal{L}$, l' este l din care a fost șters elementul de pe poziţia p, e este elementul şters

element (l, p, e)

descriere: accesarea elementului din listă de pe o anumită poziţie

pre: $l \in \mathcal{L}$, $e \in$ `TElement`, p e o *poziţie* în l, ***valid***(l, p)

post: $e \in$ `TElement`, e este elementul de pe poziţia p din l

modifica (l, p, e)

descriere: modificarea elementului din listă de pe o anumită poziţie

pre: $l \in \mathcal{L}$, $e \in$ `TElement`, p e o *poziţie* în l, ***valid***(l, p)

post: $l' \in \mathcal{L}$, l' *este* l în care s-a înlocuit elementul de pe poziţia p cu e

prim(l)

descriere: funcţie care returnează poziţia primului element în listă

pre: $l \in \mathcal{L}$

post: *prim*= poziţia primului element din l sau o poziţie care nu e validă dacă l e vidă

ultim(l)

descriere: funcţie care returnează poziţia ultimului element în listă

pre: $l \in \mathcal{L}$

post: *ultim*= poziţia ultimului element din l sau o poziţie care nu e validă dacă l e vidă

următor(l, p)

descriere: funcţie care returnează poziţia în listă următoare unei poziţii date

pre: $l \in \mathcal{L}$, p e o *poziţie* în l, ***valid***(l, p)

post: *urmator*= poziţia din l care urmează poziţiei p sau o poziţie care nu e validă dacă p e poziţia ultimului element din listă

precedent(l, p)

descriere: funcţie care returnează poziţia în listă precedentă unei poziţii date

pre: $l \in \mathcal{L}$, p e o *poziţie* în l, ***valid***(l, p)

post: *precedent*= poziţia din l care precede poziţia p sau o poziţie care nu e validă dacă p e poziţia primului element din listă

caută(l, e)

descriere: funcţie care caută un element în listă

pre: $l \in \mathcal{L}$, $e \in$ TElement

post: *caută* = prima poziţie pe care apare *e* în *l* sau o poziţie care nu e validă dacă $e \notin l$

apare(*l*, *e*)

descriere: funcţie care verifică apartenenţa unui element în listă

pre: $l \in \mathcal{L}$, $e \in$ TElement

post: *apare* = adevărat $e \in l$
fals contrar

vidă(*l*)

descriere: funcţie care verifică dacă lista este vidă

pre: $l \in \mathcal{L}$

post: *vidă* = adevărat în cazul în care *l* e lista vidă
fals în caz contrar

dim(l)

descriere: funcţie care returnează numărul de elemente din listă

pre: $l \in \mathcal{L}$

post: *dim*=numărul de elemente din listă

iterator(*l*, *i*)

descriere: se construieşte un iterator pe listă

pre: $l \in \mathcal{L}$

post: *i* este un iterator pe lista *l*

distruge(*l*)

descriere: distruge o listă

pre: $l \in \mathcal{L}$

post: lista *l* a fost distrusă

Reamintim modul în care va putea fi tipărită o listă (ca orice alt container care poate fi iterat) folosind iteratorul construit pe baza operaţiei ***iterator*** din interfaţa listei.

```
Subalgoritmul tipărire(l) este:
{pre: l este o listă}
{post: se tipăresc elementele listei}
     iterator(l, i)          {se obţine un iterator pe lista l}
     Câttimp valid(i) execută    {cât timp iteratorul e valid}
          element(i, e) {e este elementul curent referit de iterator}
          @ tipăreşte e      {se tipăreşte elementul curent}
          următor(i)         {iteratorul referă următorul element}
     sfcât
sf-tipărire
```

Observaţie

Menționăm faptul că nu este o modalitate unanim acceptată pentru specificarea operațiilor. Spre exemplu, pentru operația *adaugăSfarsit* din interfața **TAD Lista,** o altă modalitate corectă de specificare ar fi una dintre cele de mai jos:

adaugăSfarsit (*l*, *e*)

desc.: se adaugă un element la sfârşitul listei

pre: $l \in \mathcal{L}$, $e \in$ TElement

post: $l' \in \mathcal{L}$, $l' = l \cup \{e\}$, e este pe ultima poziție în l'

adaugăSfarsit (l, e)

descriere: se adaugă un element la sfârşitul listei

pre: $l \in \mathcal{L}$, $e \in$ TElement

post: $l \in \mathcal{L}$, l este modificată prin adăugarea lui e la sfârşit şi păstrarea celorlate elemente pe poziţiile lor

1.5.2. Dicţionare

Dicţionarele reprezintă containere conţinând elemente sunt forma unor perechi (cheie, valoare). Dicţionarele păstrează elemente în aşa fel încât ele să poată fi uşor localizate folosind chei. Operaţiile de bază pe dicţionare sunt căutare, adăugare şi ştergere elemente. Într-un dicţionar cheile sunt unice şi în general, o cheie are o unică valoare asociată.

Aplicaţii ale dicţionarelor sunt multiple. Spre exemplu:

- Informaţii despre conturi bancare: fiecare cont este un obiect identificat printr-un număr de cont (considerat cheia elementului) şi informaţii adiţionale (numele şi adresa deţinătorului contului, informaţii despre depozite, etc). Informaţiile adiţionale vor fi considerate ca fiind valoarea elementului.
- Informaţii despre abonaţi telefonici**:** fiecare abonat este un obiect identificat printr-un număr de telefon (considerat cheia elementului) şi informaţii adiţionale (numele şi adresa abonatului, informaţii auxiliare, etc). Informaţiile adiţionale vor fi considerate ca fiind valoarea elementului.
- Informaţii despre studenţi**:** fiecare student este un obiect identificat printr-un număr matricol (considerat cheia elementului) şi informaţii adiţionale (numele şi adresa studentului, informaţii auxiliare, etc). Informaţiile adiţionale vor fi considerate ca fiind valoarea elementului.

Dăm în continuare specificaţia Tipului Abstract de Date **Dicţionar**.

Tipul Abstract de Date DICŢIONAR

domeniu

$\mathcal{D}$={**d | d** este un dicţionar cu elemente **e = (c, v), c** de tip **TCheie, v** de tip **TValoare**}

operaţii (interfaţa minimală)

creează(d)

descriere: se creează un dicţionar vid

pre: true

post:$d \in \mathcal{D}$, d este dicţionarul vid (fără elemente)

adaugă(*d, c, v*)

descriere: se adaugă un element în dicţionar

pre: $d \in \mathcal{D}$, $c \in$ **TCheie,** $v \in$ **TValoare**

post: $d' \in \mathcal{D}$, $d' = d \cup \{c, v\}$ (se adaugă în dicţionar perechea (c, v))

caută(*d, c*)

descriere: se adaugă un element în dicţionar (după cheie)

pre: $d \in \mathcal{D}$, $c \in$ **TCheie**

post: *caută*= $v \in$ **TValoare** dacă $(c,v) \in d$

elementul nul al **TValoare** în caz contrar

şterge(*d, c*)

descriere: se adaugă un element în dicţionar (după cheie)

pre: $d \in \mathcal{D}$, $c \in$ **TCheie**

post: *şterge*= $v \in$ **TValoare** dacă $(c,v) \in d$, *d'* este *d* din care a fost șters perechea (c,v)

elementul nul al **TValoare** în caz contrar

dim(*d*)

descriere: funcţie care returnează numărul de elemente din listă

pre: $d \in \mathcal{D}$

post: ***dim***= dimensiunea dicţionarului *d* (numărul de elemente) $\in \mathcal{N}^*$

vid(*d*)

descriere: funcţie care verifică dacă dicţionarul este vid

pre: $d \in \mathcal{D}$

post: ***vid***= adevărat în cazul în care *d* e dicţionarul vid

fals în caz contrar

chei(*d, m*)

descriere: se determină mulţimea cheilor din dicţionar

pre: $d \in \mathcal{D}$

post: $m \in \mathcal{M}$, m este mulţimea cheilor din dicţionarul *d*

valori(*d, c*)

descriere: se determină colecţia valorilor din dicţionar

pre: $d \in \mathcal{D}$

post: $c \in \mathcal{C}ol$, *c* este colecţia valorilor din dicţionarul *d*

perechi(*d, m*)

descriere: se determină mulţimea perechilor (cheie, valoare) din dicţionar

pre: $d \in \mathcal{D}$

post: $m \in \mathcal{M}$, m este mulţimea perechilor (cheie, valoare) din dicţionarul *d*

iterator(*d, i*)

descriere: se creează un iterator pe dicţionar

pre: $d \in \mathcal{D}$

post: $i \in \mathcal{I}$, *i* este iterator pe dicţionarul *d*

distruge(*d*)

descriere: distruge un dicţionar

pre: $d \in \mathcal{D}$

post: dicţionarul *d* a fost distrus

Reamintim modul în care va putea fi tipărit un dicţionar (ca orice alt container care poate fi iterat) folosind iteratorul construit pe baza operaţiei ***iterator*** din interfaţa dicţionarului.

```
Subalgoritmul tipărire(d) este:
{pre: d este un dicţionar}
{post: se tipăresc elementele dicţionarului}
     iterator(d, i)          {se obţine un iterator pe dicţionarul d}
     Câttimp valid(i) execută    {cât timp iteratorul e valid}
          element(i, e) {e este elementul curent referit de iterator}
          @ tipăreşte e      {se tipăreşte elementul curent}
          următor(i)         {iteratorul referă următorul element}
     sfcât
sf-tipărire
```

1.6. Probleme propuse

1. Scrieţi un program într-unul din limbajele de programare Python, C++, Java, C# care:
   a. Defineşte o clasă **B** având un atribut ***b*** de tip întreg şi o metodă de tipărire care afişează atributul ***b*** la ieşirea standard.
   b. Defineşte o clasă **D** derivată din **B** având un atribut ***d*** de tip şir de caractere şi de asemenea o metodă de tipărire pe ieşirea standard care va afişa atributul ***b*** din clasa de bază şi atributul ***d***.
   c. Defineşte o funcţie care construieşte o listă conţinând: un obiect $\mathbf{o_1}$ de tip **B** având ***b*** egal cu 8; un obiect $\mathbf{o_2}$ de tip **D** având ***b*** egal cu 5 şi ***d*** egal cu ″D5″; un obiect $\mathbf{o_3}$ de tip **B** având ***b*** egal cu -3; un obiect $\mathbf{o_4}$ de tip **D** având ***b*** egal cu 9 şi ***d*** egal cu ″D9″.
   d. Defineşte o funcţie care primeşte o listă cu obiecte de tip **B** şi returnează o listă doar cu obiectele care satisfac proprietatea: ***b*****>6**.
   e. Pentru tipul de dată **listă** utilizat în program, scrieţi specificaţiile operaţiilor folosite.

   Se pot folosi biblioteci existente pentru structuri de date (Python, C++, Java, C#). Nu se cere implementare pentru operaţiile listei.

2. Scrieţi un program într-unul din limbajele de programare Python, C++, Java, C# care:
   a. Defineşte o clasă **B** având un atribut ***b*** de tip întreg şi o metodă de tipărire care afişează atributul ***b*** la ieşirea standard.
   b. Defineşte o clasă **D** derivată din **B** având un atribut ***d*** de tip şir de caractere şi de asemenea o metodă de tipărire pe ieşirea standard care va afişa atributul ***b*** din clasa de bază şi atributul ***d***.
   c. Defineşte o funcţie care construieşte un dicţionar conţinând: un obiect $\mathbf{o_1}$ de tip **B** având ***b*** egal cu 8; un obiect $\mathbf{o_2}$ de tip **D** având ***b*** egal cu 5 şi ***d*** egal cu ″D5″; un obiect $\mathbf{o_3}$ de tip **B** având ***b*** egal cu -3; un obiect $\mathbf{o_4}$ de tip **D** având ***b*** egal cu 9 şi ***d*** egal cu ″D9″. (*cheia* unui obiect din dicţionar este valoarea ***b***, iar *valoarea* asociată cheii este obiectul).
   d. Defineşte o funcţie care primeşte un dicţionar cu obiecte de tip **B** şi verifică dacă în dicţionar există un obiect care satisface proprietatea: ***b*****>6**.

e. Pentru tipul de dată **dicţionar** utilizat în program, scrieţi specificaţiile operaţiilor folosite.

Se pot folosi biblioteci existente pentru structuri de date (Python, C++, Java, C#). Nu se cere implementare pentru operaţiile dicţionarului.

3. Subiectul va prezenta **o diagramă de clase** şi o diagramă de **interacţiuni între obiecte** şi se va cere **scrierea unui program** care corespunde diagramelor.

Programul va putea fi scris în orice limbaj orientat pe obiecte, ex. Python, Java, C++ sau C#.

2. Baze de date

2.1. Baze de date relaţionale. Primele trei forme normale ale unei relaţii

2.1.1. Modelul relaţional

Modelul relaţional de organizare a bazelor de date a fost introdus de **E.F.Codd** în 1970 şi este cel mai studiat şi mai mult folosit model de organizare a bazelor de date. In continuare se va face o scurtă prezentare a acestui model.

Fie $A_1, A_2, ..., A_n$ o mulţime de atribute (coloane, constituanţi, nume de date, etc.) şi $D_i = Dom(A_i) \cup \{?\}$ domeniul valorilor posibile pentru atributul A_i, unde prin "**?**" s-a notat valoarea de "*nedefinit*" (*null*). Valoarea de *nedefinit* se foloseşte pentru a verifica dacă unui atribut i s-a atribuit o valoare sau el nu are valoare (sau are valoarea "nedefinit"). Această valoare nu are un anumit tip de dată, se pot compara cu această valoare atribute de diferite tipuri (numerice, şiruri de caractere, date calendaristice, etc.).

Plecând de la mulţimile astfel introduse, se poate defini o **relaţie de gradul n** sub forma următoare:

$$R \subseteq D_1 \times D_2 \times ... \times D_n,$$

şi poate fi considerată ca o mulţime de vectori cu câte **n** valori, câte o valoare pentru fiecare din atributele A_i. O astfel de relaţie se poate memora printr-un tabel de forma:

R	A_1	...	A_i	..	A_n
r_1	a_{11}	...	a_{1j}	...	a_{1n}
...	...	...	...	...	...
r_i	a_{i1}	...	a_{ij}	...	a_{in}
...	...	...	...	...	...
r_m	a_{m1}	...	a_{mj}	...	a_{mn}

unde liniile din acest tabel formează **elementele relaţiei**, sau **tupluri**, sau **înregistrări**, care în general sunt distincte, şi $a_{ij} \in D_j$, $j = 1,...,n$, $i = 1,...,m$. Deoarece modul în care se evidenţiază elementele relaţiei R de mai sus seamănă cu un tabel, relaţia se mai numeşte şi **tabel**. Pentru a pune în evidenţă numele relaţiei (tabelului) şi lista atributelor vom nota această relaţie cu:

$$R[A_1, A_2, ..., A_n].$$

Modelul relaţional al unei baze de date constă dintr-o **colecţie de relaţii** ce variază în timp (conţinutul relaţiilor se poate schimba prin operaţii de adăugare, ştergere şi actualizare).

O **bază de date relaţională** constă din trei părţi:

1. **Datele** (relaţii sau tabele, legături între tabele) şi descrierea acestora;

2. **Reguli de integritate** (pentru a memora numai valori corecte în relaţii);
3. **Operatori de gestiune** a datelor.

Exemplul 1. STUDENTI [NUME, ANUL_NASTERII, ANUL_DE_STUDIU],

cu următoarele valori posibile:

NUME	ANUL_NASTERII	ANUL_DE_STUDIU
Pop Ioan	1985	2
Barbu Ana	1987	1
Dan Radu	1986	3

Exemplul 2. CARTE [AUTORI, TITLU, EDITURA, AN_APARITIE],

cu valorile:

AUTORI	TITLU	EDITURA	AN_APARITIE
Date, C.J.	An Introduction to Database Systems	Addison-Wesley Publishing Comp.	2004
Ullman, J., Widom, J.	A First Course in Database Systems	Addison-Wesley + Prentice-Hall	2011
Helman, P.	The Science of Database Management	Irwin, SUA	1994
Ramakrishnan, R.	Database Management Systems	McGraw-Hill	2007

Pentru fiecare relaţie se poate preciza un atribut sau o colecţie de atribute, din cadrul relaţiei, numit **cheie**, cu rol de identificare a elementelor relaţiei (cheia ia valori diferite pentru înregistrări diferite din relaţie, deci fiecare înregistrare se poate identifica prin valoarea cheii). Dacă se dă câte o valoare pentru atributele din cheie, se poate determina linia (una singură) în care apar aceste valori. Vom presupune că nici o submulţime de atribute din cheie nu este cheie. Deoarece toate elementele relaţiei sunt diferite, o cheie există totdeauna (în cel mai rău caz cheia este formată din toate atributele relaţiei). Pentru exemplul 1 se poate alege NUME ca şi cheie (atunci în baza de date nu pot exista doi studenţi cu acelaşi nume), iar pentru exemplul 2 se poate alege grupul de atribute {AUTORI, TITLU, EDITURA, AN_APARITIE} ca şi cheie, sau să se introducă un nou atribut (de exemplu COTA) pentru identificare.

Pentru anumite relaţii pot fi alese mai multe chei. Una dintre chei (un atribut simplu sau un atribut compus din mai multe atribute simple) se alege **cheie principală** (**primară**), iar celelalte se vor considera **chei secundare**. Sistemele de gestiune a bazelor de date nu permit existenţa a două elemente distincte într-o relaţie cu aceeaşi valoare pentru oricare cheie (principală sau secundară), deci precizarea unei chei constituie o **restricţie** pentru baza de date.

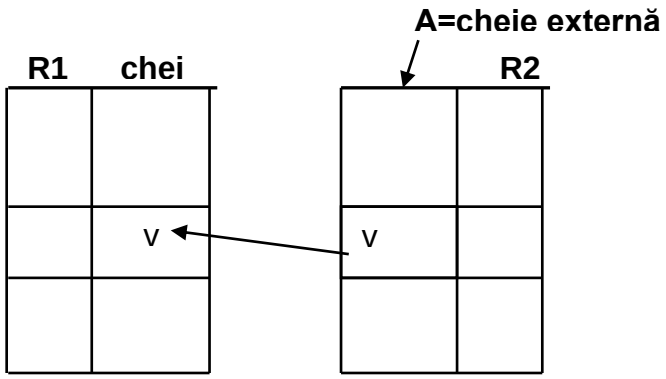
Exemplul 3. ORAR [ZI, ORA, SALA, PROFESOR, CLASA, DISCIPLINA],

cu orarul pe o săptămână. Se pot alege ca şi chei următoarele mulţimi de atribute:

{ZI, ORA, SALA}; {ZI, ORA, PROFESOR}; {ZI, ORA, CLASA}.

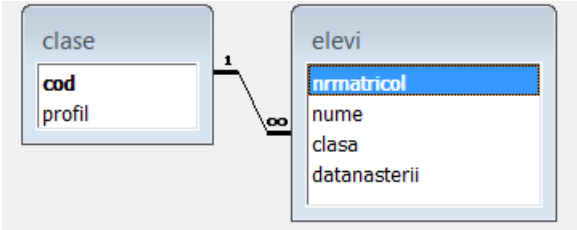
Valorile unor atribute dintr-o relaţie pot să apară în altă relaţie. Plecând de la o relaţie **R2** se pot căuta înregistrările dintr-o relaţie **R1** după valorile unui astfel de atribut (simplu sau compus). In relaţia **R2** se stabileşte un atribut **A**, numit **cheie externă**. Valorile

atributului **A** se caută printre valorile cheii din relaţia **R1**. Cele două relaţii **R1** şi **R2** nu este obligatoriu să fie distincte.



Exemplu:

CLASE [cod, profil]
ELEVI [nrmatricol, nume, clasa, datanasterii].

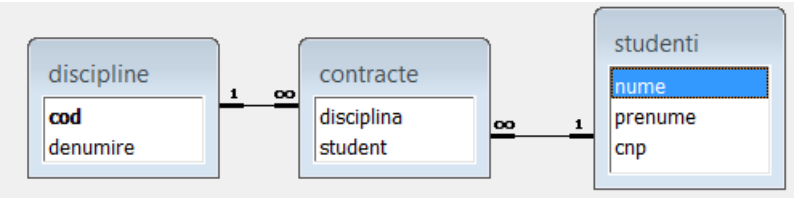


Legătura o putem stabili între relaţia **CLASE** (considerată ca părinte pentru legătură) şi relaţia **ELEVI** (ca membru pentru legătură) prin egalitatea **CLASE.cod=ELEVI.clasa**. Unei anumite clase (memorată în relaţia **CLASE**), identificată printr-un cod, îi corespund toţi elevii din clasa cu codul respectiv.

Prin **cheie externă** se pot memora **legături 1:n** între entităţi: la o clasă corespund oricâţi elevi, iar unui elev îi este asociată cel mult o clasă.

Cheia externă se poate folosi şi pentru a memora **legături <u>m:n</u>** între entităţi.

Fie două entităţi: **discipline** şi **studenţi**. La o disciplină sunt "inscrişi" mai mulţi studenţi, iar un student are asociate mai multe discipline. Varianta de memorare cuprinde o relaţie intermediară.



Pentru ca valorile dintr-o bază de date să fie corecte, la definirea bazei de date se pot preciza anumite **restricţii de intergritate** (ele sunt verificate de sistemul de gestiune a bazei de date la modificarea datelor din tabele). Aceste restricţii se referă la o coloană, la un tabel, la o legătură între două tabele:

- **restricţii asociate coloanei:**
  - Not Null - coloana nu poate să primească valori nedefinite
  - Primary Key - coloana curentă se defineşte cheia primară
  - Unique - valorile coloanei sunt unice

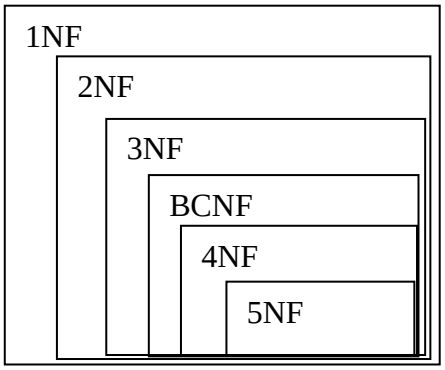
- Check(condiţie) - se dă condiţia pe care trebuie să o îndeplinească valorile coloanei (condiţii simple, care au valoarea *true* sau *false*)
- Foreign Key REFERENCES tabel_parinte [(nume_coloana)] [On Update *actiune*] [On Delete *actiune*] - coloana curentă este cheie externă

- **restricţii asociate tabelului:**
  - Primary key(lista coloane) - definirea cheii primare pentru tabel
  - Unique(lista coloane) - valorile sunt unice pentru lista de coloane precizată
  - Check(condiţie) - pentru a preciza condiţia pe care trebuie să o îndeplinească valorile unei linii
  - Foreign Key nume_cheie_externa(lista_coloane) REFERENCES tabel_parinte [(lista_coloane)] [On Update *actiune*] [On Delete *actiune*] - se defineşte cheia externă

2.1.2. Primele trei forme normale ale unei relaţii

In general anumite date se pot reprezenta în mai multe moduri prin relaţii (la modelul relaţional). Pentru ca aceste date **să se poată prelucra cât mai simplu** (la o operaţie de actualizare a datelor să nu fie necesare teste suplimentare) este necesar ca relaţiile în care se memorează datele să verifice anumite condiţii (să aibă un anumit nivel de normalizare).

Până în prezent se cunosc mai multe **forme normale** pentru relaţii, dintre care cele mai cunoscute sunt: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF. Avem următoarele incluziuni pentru relaţii în diferite forme normale:



Dacă o relaţie nu este de o anumită formă normală, atunci ea se poate descompune în mai multe relaţii de această formă normală.

Definiţie. Pentru descompunerea unei relaţii se foloseşte operatorul de **proiecţie**. Fie $R[A_1, A_2, ..., A_n]$ o relaţie şi $\alpha = \{A_{i_1}, A_{i_2}, ..., A_{i_p}\}$ o submulţime de atribute, $\alpha \subset \{A_1, A_2, ..., A_n\}$. Prin **proiecţia** relaţiei **R** pe **α** se înţelege relaţia:

$$R'[A_{i_1}, A_{i_2}, ..., A_{i_p}] = \prod_{\alpha}(R) = \prod_{\{A_{i_1}, A_{i_2}, ..., A_{i_p}\}}(R),,$$

unde:

$$\forall r = (a_1, a_2, ..., a_n) \in R \Rightarrow \prod_{\alpha}(r) = r[\alpha] = (a_{i_1}, a_{i_2}, ..., a_{i_p}) \in R',$$

şi toate elementele din R' sunt distincte.

Definiţie. Pentru compunerea relaţiilor se foloseşte operatorul de **join natural**. Fie $R[\alpha,\beta]$, $S[\beta,\gamma]$ două relaţii peste mulţimile de atribute α,β,γ, $\alpha\cap\gamma=\emptyset$. Prin **joinul natural** al relaţiilor **R** şi **S** se înţelege relaţia:

$$R*S[\alpha,\beta,\gamma]=\left\{\left(\prod_{\alpha}(r),\prod_{\beta}(r),\prod_{\gamma}(s)\right)\middle| r\in R, s\in S \quad si \quad \prod_{\beta}(r)=\prod_{\gamma}(s)\right\}$$

O relaţie R se poate descompune în mai multe relaţii noi $R_1,R_2,...,R_m$. Această descompunere este **bună** dacă $R=R_1*R_2*...*R_m$, deci datele din R se pot obţine din datele memorate în relaţiile $R_1,R_2,...,R_m$ şi nu apar date noi prin aceste operaţii de compunere.

Exemplu de descompunere care **nu este bună**: fie relaţia:

ContracteStudiu[Student ,CadruDidactic, Disciplina],

şi două noi relaţii obţinute prin proiecţia acestei relaţii: **SC**[Student, CadruDidactic] şi **CD**[CadruDidactic, Disciplina]. Presupunem că pentru relaţia iniţială avem următoarele valori:

R	**Student**	**CadruDidactic**	**Disciplina**
r_1	s1	c1	d1
r_2	s2	c2	d2
r_3	s1	c2	d3

Folosind definiţia proiecţiei se obţin următoarele valori pentru cele două relaţii obţinute din R şi pentru joinul natural al acestor relaţii:

SC	**Student**	**CadruDidactic**
r_1	s1	c1
r_2	s2	c2
r_3	s1	c2

CD	CadruDidactic	Disciplina
r_1	c1	d1
r_2	c2	d2
r_3	c2	d3

SC*CD	**Student**	**CadruDidactic**	**Disciplina**
r_1	s1	c1	d1
r_2	s2	c2	d2
?	s2	c2	d3
?	s1	c2	d2
r_3	s1	c2	d3

Se observă că în relaţia **SC*CD** se obţin înregistrări suplimentare faţă de relaţia iniţială**,** deci descompunerea sugerată **nu este bună**.

Observaţie. Prin **atribut simplu** vom înţelege un atribut oarecare din relaţie, iar prin **atribut compus** vom înţelege o mulţime de atribute (cel puţin două) din relaţie.

Este posibil ca în diverse aplicaţii practice să apară atribute (simple sau compuse) ce iau mai multe valori pentru un element din relaţie. Aceste atribute formează un **atribut repetitiv**.

Exemplul 4. Fie relaţia:

STUDENT [NUME, ANULNASTERII, GRUPA, DISCIPLINA, NOTA],

cu atributul NUME ca şi cheie. In acest exemplu perechea {DISCIPLINA, NOTA} este un grup repetitiv. Putem avea următoarele valori în această relaţie:

NUME	ANULNASTERII	GRUPA	DISCIPLINA	NOTA
Pop Ioan	1998	221	Baze de date	10
			Sisteme de operare	9
			Probabilităţi	8
Mureşan Ana	1999	222	Baze de date	8
			Sisteme de operare	7
			Probabilităţi	10
			Proiect individual	9

Exemplul 5. Fie relaţia:

CARTE [Cota, NumeAutori, Titlu, Editura, AnApariţie, Limba, CuvinteCheie],

cu atributul **Cota** ca şi cheie şi atributele repetitive **NumeAutori** şi **CuvinteCheie**. Atributul **Cota** poate avea o semnificaţie efectivă (să existe o cotă asociată la fiecare carte) sau să fie introdus pentru existenţa cheii (valorile să fie distincte, eventual pot să fie generate automat).

Grupurile repetitive crează foarte multe greutăţi în memorarea diverselor relaţii şi din această cauză se încearcă **evitarea** lor, fără însă a pierde date. Dacă **R[A]** este o relaţie, unde **A** este mulţimea atributelor, iar **α** formează un grup repetitiv (atribut simplu sau compus), atunci **R** se poate descompune în două relaţii fără ca **α** să fie atribut repetitiv. Dacă **C** este o cheie pentru relaţia **R**, atunci cele două relaţii în care se descompune relaţia **R** sunt**:**

$$R'[C \cup \alpha] = \prod_{C \cup \alpha}(R) \text{ şi } R''[A - \alpha] = \prod_{A-\alpha}(R).$$

Exemplul 6. Relaţia STUDENT din exemplul 4 se descompune în următoarele două relaţii:

DATE_GENERALE [NUME, ANULNASTERII, GRUPA],
REZULTATE [NUME, DISCIPLINA, NOTA].

Exemplul 7. Relaţia CARTE din exemplul 5 se descompune în următoarele trei relaţii (în relaţia CARTE există două grupuri repetitive):

CARTI [Cota, Titlu, Editura, AnApariţie, Limba],
AUTORI [Cota, NumeAutor],
CUVINTE_CHEIE [Cota, CuvântCheie].

Observaţie. Dacă o carte nu are autori sau cuvinte cheie asociate, atunci ea va avea câte o înregistrare în relaţiile AUTORI sau CUVINTE_CHEIE în care al doilea atribut are valoarea **null**. Dacă se doreşte eliminarea acestor înregistrări, atunci relaţia **CARTE** nu se va putea obţine din cele trei relaţii numai prin join natural (sunt necesari operatori de join extern).

Definiţie. O relaţie este de **prima formă normală (1NF)** dacă ea **nu conţine grupuri** (de atribute) **repetitive**.

Sistemele de gestiune a bazelor de date relaţionale permit descrierea numai a relaţiilor ce se află în 1NF. Există şi sisteme ce permit gestiunea relaţiilor non-1NF (exemplu Oracle, unde o coloană poate fi un *obiect* sau o "*colecţie*" de date, sau mai recent bazele de date NoSQL).

Următoarele forme normale ale unei relaţii utilizează o noţiune foarte importantă, şi anume **dependenţa funcţională** dintre diverse submulţimi de atribute. Stabilirea dependenţelor funcţionale este o sarcină a administratorului bazei de date şi depinde de semnificaţia (semantica) datelor ce se memorează în relaţie. Operaţiile de actualizare a datelor din relaţie (înserare, ştergere, modificare) nu trebuie să modifice dependenţele funcţionale (dacă pentru relaţie există astfel de dependenţe).

Definiţie. Fie $R[A_1, A_2, ..., A_n]$ o relaţie şi $\alpha, \beta \subset \{A_1, A_2, ..., A_n\}$ două submulţimi de atribute. Atributul (simplu sau compus) β este **dependent funcţional** de atributul α (simplu sau compus), notaţie: $\alpha \to \beta$, dacă şi numai dacă fiecare valoare a lui α din **R** are asociată o **valoare precisă şi unică** pentru β (această asociere este valabilă "tot timpul" existenţei relaţiei **R**). O valoare oarecare a lui α poate să apară în mai multe linii ale lui **R** şi atunci fiecare dintre aceste linii conţine aceeaşi valoare pentru atributul β, deci:

$$\prod_{\alpha}(r) = \prod_{\alpha}(r') \text{ implică } \prod_{\beta}(r) = \prod_{\beta}(r').$$

Valoarea α din implicaţia (dependenţa) $\alpha \to \beta$ se numeşte ***determinant***, iar β este ***determinat***.

Observaţie. Dependenţa funcţională se poate folosi ca o **proprietate (restricţie)** pe care baza de date trebuie să o îndeplinească pe perioada existenţei acesteia: se adaugă, elimină, modifică elemente în relaţie numai dacă dependenţa funcţională este verificată.

Existenţa unei dependenţe funcţionale într-o relaţie înseamnă că anumite asocieri de valori se memorează de mai multe ori, deci apare o **redundanţă**. Pentru exemplificarea unor probleme care apar vom lua relaţia următoare, care memorează rezultatele la examene pentru studenţi:

Exemplul 8. EXAMEN [<u>NumeStudent, Disciplina</u>, Nota, CadruDidactic],

unde cheia este {*NumeStudent, Disciplina*}. Deoarece unei discipline îi corespunde un singur cadru didactic, iar unui cadru didactic pot să-i corespundă mai multe discipline, putem cere ca să fie îndeplinită restricţia (dependenţa) *{Disciplina}→{CadruDidactic}*.

Examen	NumeStudent	Disciplina	Nota	CadruDidactic
1	Alb Ana	Matematică	10	Rus Teodor
2	Costin Constantin	Istorie	9	Popa Horea
3	Alb Ana	Istorie	8	Popa Horea
4	Enisei Elena	Matematică	9	Rus Teodor
5	Frişan Florin	Matematică	10	Rus Teodor

Dacă păstrăm o astfel de dependenţă funcţională, atunci pot apare următoarele probleme:

- **Risipă de spaţiu**: aceleaşi asocieri se memorează de mai multe ori. Legătura dintre disciplina de *Matematică* şi profesorul *Rus Teodor* este memorată de trei ori, iar dintre disciplina *Istorie* şi profesorul *Popa Horea* se memorează de două ori.
- **Anomalii la actualizare**: schimbarea unei date ce apare într-o asociere implică efectuarea acestei modificări în toate asocierile (fără a se şti câte astfel de asocieri există), altfel baza

de date va conţine erori (va fi inconsistentă). Dacă la prima înregistrare se schimbă valoarea atributului *CadruDidactic* şi nu se face aceeaşi modificare şi la înregistrările 4 şi 5, atunci modificarea va introduce o eroare în relaţie.

- **Anomalii la înserare**: la adăugarea unei înregistrări trebuie să se cunoască valorile atributelor, nu se pot folosi valori nedefinite pentru atributele implicate în dependenţele funcţionale.
- **Anomalii la ştergere**: la ştergerea unor înregistrări se pot şterge şi asocieri (între valori) ce nu se pot reface. De exemplu, dacă se şterg înregistrările 2 şi 3, atunci asocierea dintre *Disciplina* şi *CadruDidactic* se pierde.

Anomaliile de mai sus apar datorită existenţei unei dependenţe funcţionale între mulţimi de atribute. Pentru a elimina situaţiile amintite trebuie ca aceste dependenţe (asocieri) de valori să se păstreze într-o relaţie separată. Pentru aceasta este necesar ca relaţia iniţială să se descompună, fără ca prin descompunere să se piardă date sau să se introducă date noi prin compunerea de relaţii (trebuie ca descompunerea "să fie bună"). O astfel de descompunere se face în momentul proiectării bazei de date, când se pot stabili dependenţele funcţionale.

Observaţii. Se pot demonstra uşor următoarele proprietăţi simple pentru dependenţele funcţionale:

1. Dacă C este o cheie pentru $R[A_1, A_2, ..., A_n]$, atunci $C \to \beta, \forall \beta \subset \{A_1, A_2, ..., A_n\}$.
2. Dacă $\beta \subseteq \alpha$, atunci $\alpha \to \beta$, numită **dependenţa funcţională trivială** sau **reflexivitatea**.
$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\beta \subset \alpha}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \alpha \to \beta$$
3. Dacă $\alpha \to \beta$, atunci $\gamma \to \beta, \forall \gamma$ cu $\alpha \subset \gamma$.
$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \underset{\alpha \subset \gamma}{\Rightarrow} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \gamma \to \beta$$
4. Dacă $\alpha \to \beta$ şi $\beta \to \gamma$, atunci $\alpha \to \gamma$, care este proprietatea de **tranzitivitate** a dependenţei funcţionale.
$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \underset{\beta \to \gamma}{\Rightarrow} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \to \gamma$$
5. Dacă $\alpha \to \beta$ şi $\gamma \subset A$, atunci $\alpha\gamma \to \beta\gamma$, unde $\alpha\gamma = \alpha \cup \gamma$.
$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left| \begin{array}{c} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{array} \right. \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

Definiţie. Un atribut A (simplu sau compus) se numeşte **prim** dacă există o cheie C şi $A \subset C$ (C este o cheie compusă, sau A este chiar o cheie a relaţiei). Dacă un atribut nu este inclus în nici o cheie, atunci se numeşte **neprim**.

Definiţie. Fie $R[A_1, A_2, ..., A_n]$ şi $\alpha, \beta \subset \{A_1, A_2, ..., A_n\}$. Atributul β este **complet dependent funcţional** de α dacă β este dependent funcţional de α (deci $\alpha \to \beta$) şi nu este dependent funcţional de nici o submulţime de atribute din α ($\forall \gamma \subset \alpha, \delta \to \beta$ nu este adevărat).

Observaţie. Dacă atributul β **nu** este **complet dependent funcţional** de α (deci este dependent de o submulţime a lui α), atunci α este un **atribut compus**.

Definiţie. O relaţie este de a **doua formă normlă (2NF)** dacă:

- **este de prima formă normală**,

- orice **atribut neprim** (simplu sau compus) (deci care nu este inclus într-o cheie) **este complet dependent funcţional de oricare cheie a relaţiei**.

Observaţie. Dacă o relaţie este de prima formă normală (1NF) şi nu este de a doua formă normală (2NF), atunci **are o cheie compusă** (dacă o relaţie nu este de a doua formă normală, atunci există o dependenţă funcţională $\alpha \to \beta$ cu α atribut inclus într-o cheie).

Pentru a preciza modul de **descompunere** pentru cazul general, fie $R[A_1, A_2,...,A_n]$ o relaţie şi $C \subset A = \{A_1, A_2,...,A_n\}$ o cheie. Presupunem că există $\beta \subset A$, $\beta \cap C = \emptyset$ (β este un atribut necheie), β dependent funcţional de $\alpha \subset C$ (β este complet dependent funcţional de o submulţime strictă de atribute din cheie). Dependenţa $\alpha \to \beta$ se poate elimina dacă relaţia **R** se descompune în următoarele două relaţii:

$$R'[\alpha \cup \beta] = \prod_{[\alpha \cup \beta]}(R) \text{ şi } R''[A - \beta] = \prod_{A-\beta}(R).$$

Vom analiza relaţia din exemplul 8:

EXAMEN [<u>NumeStudent, Disciplina</u>, Nota, CadruDidactic],

unde cheia este {*NumeStudent, Disciplina*} şi există dependenţa funcţională (restricţia) *{Disciplina}→{CadruDidactic}*. De aici deducem că atributul *CadruDidactic* nu este complet dependent funcţional de o cheie, deci relaţia EXAMEN nu este de a doua formă normală. Eliminarea acestei dependenţe funcţionale se poate face prin descompunerea relaţiei în următoarele relaţii:

APRECIERI [NumeStudent, Disciplina, Nota];
STAT_FUNCTII [Disciplina, CadruDidactic].

Exemplul 9. Presupunem că pentru memorarea contractelor de studiu se foloseşte relaţia:

CONTRACTE[Nume, Prenume, <u>CNP, CodDisciplina</u>, DenumireDisciplina].

Cheia relaţiei este *{CNP,CodDisciplina}*. In relaţie mai există două dependenţe funcţionale:

{CNP} →{Nume, Prenume} şi *{CodDisciplina}→{DenumireDisciplina}*. Pentru eliminarea acestor dependenţe se descompune relaţia în următoarele:

STUDENTI [CNP, Nume, Prenume],
INDRUMATORI [CodDisciplina, DenumireDisciplina],
CONTRACTE [CNP, CodDisciplina].

Pentru a treia formă normală este necesară noţiunea de **dependenţă tranzitivă**.

Definiţie. Un atribut **Z** este **tranzitiv dependent** de atributul **X** dacă $\exists$ **Y** încât $X \to Y$, $Y \to Z$, iar $Y \to X$ nu are loc şi Z nu este inclus în $X \cup Y$.

Definiţie. O relaţie este de a **treia formă normală (3NF)** dacă este **2NF** şi **orice atribut neprim nu este tranzitiv dependent de oricare cheie a relaţiei**.

Dacă **C este o cheie** şi β un atribut tranzitiv dependent de cheie, atunci există un atribut α care verifică: $C \to \alpha$ (dependenţă care este verificată totdeauna) şi $\alpha \to \beta$. Deoarece relaţia este 2NF, obţinem că β este complet dependent de **C**, deci $\alpha \not\subset C$. De aici deducem că o relaţie ce este 2NF şi nu este 3NF are o dependenţă $\alpha \to \beta$, iar α este atribut neprim. Această dependenţă se poate elimina prin descompunerea relaţiei R în mod asemănător ca la eliminarea dependenţelor de la 2NF.

Exemplul 10. Rezultatele obţinute de absolvenţi la lucrarea de licenţă sunt trecute în relaţia:

LUCRARI_LICENTA [<u>NumeAbsolvent</u>, Nota, CadruDidIndr, Departament].

Aici se memorează numele cadrului didactic îndrumător şi denumirea departamentului la care se află acesta. Deoarece se introduc date despre absolvenţi, câte o înregistrare pentru un absolvent, putem să stabilim că *NumeAbsolvent* este cheia relaţiei. Din semnificaţia atributelor incluse în relaţie se observă următoarea dependenţă funcţională:

$\{CadruDidIndr\} \rightarrow \{Departament\}$.

Din existenţa acestei dependenţe funcţionale se deduce că relaţia **nu este de 3NF**. Pentru a elimina dependenţa funcţională, relaţia se poate descompune în următoarele două relaţii:

REZULTATE [NumeAbsolvent, Nota, CadruDidIndr]
INDRUMATORI [CadruDidIndr, Departament].

Exemplul 11. Presupunem că adresele unui grup de persoane se memorează în următoarea relaţie:

ADRESE [<u>CNP</u>, Nume, Prenume, CodPostal, LocalitateDomiciliu, Strada, Nr].

Cheia relaţiei este {CNP}. Deoarece la unele localităţi codul poştal se stabileşte la nivel de stradă, sau chiar poţiuni de stradă, există dependenţa funcţională:

$\{CodPostal\} \rightarrow \{LocalitateDomiciliu\}$.

Deoarece există această dependenţă funcţională, deducem că relaţia ADRESE nu este de a treia formă normală, deci este necesară descompunerea ei.

Exemplul 12. Să considerăm următoarea relaţie care memorează o eventuală planificare a studenţilor pentru examene:

PLANIFICARE_EX [Data, Ora, Cadru_did, Sala, Grupa],

cu următoarele restricţii (cerinţe care trebuie respectate) şi care se transpun în definirea de chei sau de dependenţe funcţionale:

1. Un student dă maximum un examen într-o zi, deci {*Grupa, Data*} este cheie.
2. Un cadru didactic are examen cu o singură grupă la o anumită oră, deci {*Cadru_did, Data, Ora*} este cheie.
3. La un moment dat într-o sală este planificat cel mult un examen, deci {*Sala, Data, Ora*} este cheie.
4. Intr-o zi cadrul didactic nu schimbă sala, în sala respectivă pot fi planificate şi alte examene, dar la alte ore, deci există următoarea dependenţă funcţională:

   $\{Cadru\_did, Data\} \rightarrow \{Sala\}$

Toate atributele din această relaţie apar în cel puţin o cheie, deci nu există atribute neprime. Având în vedere definiţia formelor normale precizate până acuma, putem spune că relaţia **este în 3NF**. Pentru a elimina şi dependenţele funcţionale de tipul celor pe care le avem în exemplul de mai sus s-a introdus o nouă formă normală:

Definiţie. O relaţie este în **3NF Boyce-Codd,** sau **BCNF,** dacă orice determinant (pentru o dependenţă funcţională) este cheie, deci nu există dependenţe funcţionale $\alpha \rightarrow \beta$ astfel încât α să nu fie cheie.

Pentru a elimina dependenţa funcţională amintită mai sus trebuie să facem următoarea descompunere pentru relaţia **PLANIFICARE_EX**:

PLANIFICARE_EX [Data, Cadru_did, Ora, Student],

REPARTIZARE_SALI [Cadru_did, Data, Sala].

După această descompunere nu mai există dependenţe funcţionale, deci relaţiile sunt de tipul BCNF, dar a dispărut cheia asociată restricţiei precizate la punctul 3 de mai sus: {*Sala, Data, Ora*}. Dacă se mai doreşte păstrată o astfel de restricţie, atunci ea trebuie verificată altfel (de exemplu, prin program).

2.2. Interogarea BD cu operatori din algebra relaţională

Pentru a **explica** limbajul de interogare (cererea de date) bazat pe algebra relaţiilor vom preciza la început **tipurile de condiţii** ce pot apare în cadrul diferiţilor operatori relaţionali.

1. Pentru a verifica dacă un atribut îndeplineşte o condiţie simplă se face compararea acestuia cu o anumită **valoare**, sub forma:

nume atribut ***operator_relaţional*** **valoare**

2. O relaţie cu o singură coloană poate fi considerată ca o mulţime. Următoarea condiţie testează dacă o anumită valoare aparţine sau nu unei mulţimi:

$$\text{nume\_atribut} \left\{ \begin{array}{c} IS\ IN \\ IS\ NOT\ IN \end{array} \right\} \text{relaţie\_cu\_o\_coloană}$$

3. Două relaţii (considerate ca mulţimi de înregistrări) se pot compara prin operaţiile de egalitate, diferit, incluziune, neincluziune. Intre două relaţii cu acelaşi număr de coloane şi cu aceleaşi tipuri de date pentru coloane (deci între două mulţimi comparabile) putem avea condiţii de tipul următor:

$$\text{relaţie} \left\{ \begin{array}{c} IS\ IN \\ IS\ NOT\ IN \\ = \\ <> \end{array} \right\} \text{relaţie}$$

4. Tot *condiţie* este şi oricare din construcţiile următoare:

(condiţie)
NOT condiţie
condiţie1 AND condiţie2
condiţie1 OR condiţie2

unde condiţie, condiţie1, condiţie2 sunt condiţii de tipurile 1-4.

In primul tip de condiţie apare construcţia *'valoare'*, care poate fi una din tipurile următoare. Pentru fiecare construcţie se ia în *valoare* o anumită relaţie curentă, care rezultă din contextul în care apare aceasta.

- **nume_atribut** - care precizează valoarea atributului dintr-o înregistrare curentă. Dacă precizarea numai a numelui atributului crează ambiguitate (există mai multe relaţii curente care conţin câte un atribut cu acest nume), atunci se va face o calificare a atributului cu numele relaţiei sub forma: **relaţie.atribut**.

- **expresie** - dacă se evaluează expresia, iar dacă apar şi denumiri de atribute, atunci acestea se iau dintr-o înregistrare curentă.
- **COUNT(*) FROM relaţie** - precizează numărul de înregistrări din relaţia specificată.
- $\left\{\begin{matrix} COUNT \\ SUM \\ AVG \\ MAX \\ MIN \end{matrix}\right\}\left([DISTINCT]\, nume\_atribut\right)$ - care determină o valoare plecând de la toate înregistrările din relaţia curentă. La determinarea acestei valori se iau toate valorile atributului precizat ca argument (din toate înregistrările), sau numai valorile distincte, după cum lipseşte sau apare cuvântul DISTINCT. Valorile astfel determinate sunt: numărul de valori (pentru COUNT), suma acestor valori (apare SUM, valorile trebuie să fie numerice), valoarea medie (apare AVG, valorile trebuie să fie numerice), valoarea maximă (apare MAX), respectiv valoarea minimă (apare MIN).

In continuare se vor preciza **operatorii** care se pot folosi pentru **interogarea bazelor de date relaţionale**.

- **Selecţia** (sau proiecţia orizontală) a unei relaţii **R** - determină o nouă relaţie ce are aceeaşi schemă cu a relaţiei **R**. Din relaţia **R** se iau numai înregistrările care îndeplinesc o condiţie **c**. Notaţia pentru acest operator este: $\sigma_c(R)$.
- **Proiecţia** (sau proiecţia verticală) - determină o relaţie nouă ce are atributele precizate printr-o mulţime **α** de atribute. Din fiecare înregistrare a unei relaţii R se determină numai valorile atributelor incluse în mulţimea **α**. Mulţimea **α** de atribute se poate extinde la o **mulţime de expresii** (în loc de o mulţime de atribute), care precizează coloanele relaţiei care se construieşte. Notaţia pentru acest operator este: $\prod_{\alpha}(R)$.
- **Produsul cartezian** a două relaţii: $\mathbf{R_1 \times R_2}$ - care determină o relaţie nouă ce are ca atribute concatenarea atributelor din cele două relaţii, iar fiecare înregistrare din R_1 se concatenează cu fiecare înregistrare din R_2.
- **Reuniunea**, **diferenţa** şi **intersecţia** a două relaţii: $R_1 \cup R_2$, $R_1 - R_2$, $R_1 \cap R_2$. Cele două relaţii trebuie să aibă *aceeaşi schemă*.
- Există mai mulţi operatori **join**.

  Joinul condiţional sau **theta join**, notat prin $R_1 \otimes_{\Theta} R_2$ - care determină acele înregistrări din produsul cartezian al celor două relaţii care îndeplinesc o anumită condiţie. Din definiţie se observă că avem: $R_1 \otimes_{\Theta} R_2 = \sigma_{\Theta}(R_1 \times R_2)$.

  Joinul natural, notat prin $R_1 * R_2$, care determină o relaţie nouă ce are ca atribute reuniunea atributelor din cele două relaţii, iar înregistrările se obţin din toate perechile de înregistrări ale celor două relaţii care au aceleaşi valori pentru atributele comune. Dacă cele două relaţii au schemele $R_1[\alpha], R_2[\beta]$, şi $\alpha \cap \beta = \{A_1, A_2, ..., A_n\}$, atunci joinul natural se poate calcula prin construcţia următoare:

$$R_1 * R_2 = \prod_{\alpha \cup \beta}\left(R_1 \otimes_{R_1.A_1 = R_2.A_1 \text{ and } ... \text{ and } R_1.A_n = R_2.A_n} R_2\right)$$

Joinul extern stânga, notat (în acest material) prin $R_1 \triangleright_C R_2$, determină o relaţie nouă ce are ca atribute concatenarea atributelor din cele două relaţii, iar înregistrările se obţin astfel: se iau înregistrările care se obţin prin joinul condiţional $R_1 \otimes_c R_2$, la care se adaugă înregistrările din R_1 care nu s-au folosit la acest join condiţional combinate cu valoarea *null* pentru toate atributele corespunzătoare relaţiei R_2.

Joinul extern dreapta, notat prin $R_1 \triangleleft_C R_2$, se obţine asemănător ca joinul extern stânga, dar la înregistrările din $R_1 \otimes_c R_2$ se adaugă înregistrările din R_2 care nu s-au folosit la acest join condiţional combinate cu valoarea *null* pentru toate atributele corespunzătoare relaţiei R_1.

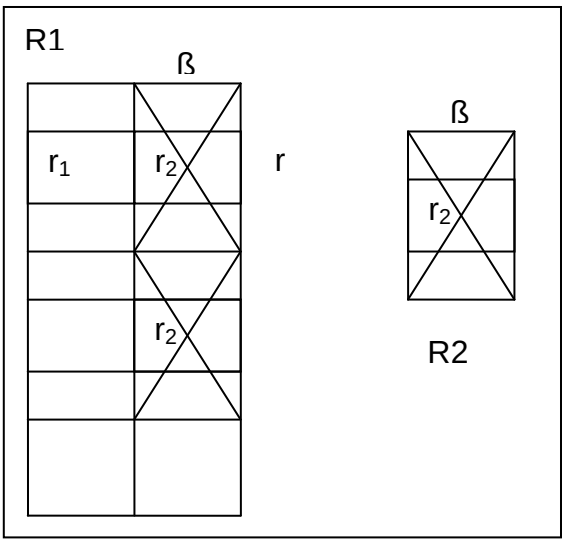
- **Câtul** pleacă de la două relaţii $R_1[\alpha], R2[\beta]$,

  cu $\beta \subset \alpha$, şi se notează prin $R_1 \div R_2[\alpha - \beta]$. Deducem că atributele din cât sunt date de mulţimea $\alpha - \beta$. O înregistrare $r \in R_1 \div R_2$ dacă $\forall r_2 \in R_2$, $\exists r_1 \in R_1$ ce îndeplineşte condiţiile:

  1. $\prod_{\alpha-\beta}(r_1) = r$;

  2. $\prod_{\beta}(r_1) = r_2$.

  Semnificaţia relaţiei cât se vede şi din figura alăturată. O înregistrare r_1 aparţine câtului dacă în relaţia R_1 apar toate concatenările dintre această înregistrare şi fiecare înregistrare din R_2.



O problemă importantă legată de operatorii descrişi mai sus constă în determinarea unei **submulţimi independente** de operatori. O mulţime M de operatori este **independentă** dacă eliminând un operator oarecare *op* din M se diminuează puterea mulţimii, adică va exista o relaţie obţinută cu operatori din M şi care nu se poate obţine cu operatori din mulţimea M - {*op*}.

Pentru limbajul de interogare descris mai sus, o mulţime independentă este formată din submulţimea: $\{\sigma, \prod, \times, \cup, -\}$. Ceilalţi operatori se obţin după regulile următoare (unele expresii au fost deja deduse mai sus):

- $R_1 \cap R_2 = R_1 - (R_1 - R_2)$;
- $R_1 \otimes_c R_2 = \sigma_c(R_1 \times R_2)$;
- $R_1[\alpha], R2[\beta]$, şi $\alpha \cap \beta = \{A_1, A_2, ..., A_n\}$, atunci

  $$R_1 * R_2 = \prod_{\alpha \cup \beta} \left(R_1 \otimes_{R_1.A_1 = R_2.A_1 \text{ and } ... \text{ and } R_1.A_n = R_2.A_n} R_2 \right);$$

- Fie $R_1[\alpha], R2[\beta]$, şi $R_3[\beta]$ = (null, ... , null), $R_4[\alpha]$ = (null, ... , null).

  $$R_1 \triangleright_C R_2 = (R_1 \otimes_c R_2) \cup [R_1 - \prod_{\alpha}(R_1 \otimes_C R_2)] \times R_3.$$

$$R_1 \triangleleft_C R_2 = (R_1 \otimes_C R_2) \cup R_4 \times [R_2 - \prod_{\beta}(R_1 \otimes_C R_2)].$$

- Dacă $R_1[\alpha], R2[\beta]$, cu $\beta \subset \alpha$, atunci $r \in R_1 \div R_2$ dacă $\forall r_2 \in R_1 \div R_2$, $\exists r_1 \in R_1$ ce îndeplineşte condiţiile: $\prod_{\alpha-\beta}(r_1) = r$ şi $\prod_{\beta}(r_1) = r_2$.

  De aici deducem că **r** este din $\prod_{\alpha-\beta}(R_1)$. In $\left(\prod_{\alpha-\beta}(R_1)\right) \times R_2$ sunt toate elementele ce au o parte în $\prod_{\alpha-\beta}(R_1)$ şi a doua parte în R_2. Din relaţia astfel obţinută vom elimina pe R_1 şi rămân acele elemente ce au o parte în $\prod_{\alpha-\beta}(R_1)$ şi nu au cealaltă parte în $\prod_{\beta}(R_1)$. De aici obţinem:

  $$R_1 \div R_2 = \prod_{\alpha-\beta}(R_1) - \prod_{\alpha-\beta}\left(\left(\prod_{\alpha-\beta}(R_1)\right) \times R_2 - R_1\right).$$

La lista de operatori relaţionali amintiţi mai sus se pot aminti câteva instrucţiuni utile la rezolvarea unor probleme:

- **Atribuirea**: unei variabile (relaţii) R îi vom atribui o relaţie dată printr-o expresie construită cu operatorii de mai sus. In instrucţiune se poate preciza, pentru R, şi denumirea coloanelor.

  R[lista] := expresie

- **Eliminarea duplicărilor** unei relaţii: $\delta(R)$
- **Sortarea** înregistrărilor dintr-o relaţie: $s_{\{lista\}}(R)$
- **Gruparea:** $\gamma_{\{lista\,1\}\ group\ by\ \{lista\,2\}}(R)$, care este o extensie pentru proiecţie. Inregistrările din R sunt grupate după coloanele din *lista2*, iar pentru un grup de înregistrări cu aceleaşi valori pentru *lista2* se evaluează *lista1* (unde pot apare *funcţii de grupare*).

2.3. Interogarea bazelor de date relaţionale cu SQL

Pentru gestiunea bazelor de date relaţionale s-a construit limbajul **SOL** (Structured Query Language), ce permite gestiunea componentelor unei baze de date (tabel, index, utilizator, procedură memorată, etc.).

Scurt istoric:

- 1970 - E.F. Codd formalizează modelul relaţional
- 1974 - la IBM (din San Jose) se defineşte limbajul SEQUEL (Structured English Query Language)
- 1975 - se defineşte limbajul SQUARE (Specifying Queries as Relational Expressions).
- 1976 - la IBM se defineşte o versiune modificată a limbajului SEQUEL, cu numele SEQUEL/2. După o revizuire devine SQL
- 1986 - SQL devine standard ANSI (American National Standards Institute)
- 1987 - SQL este adoptată de ISO (International Standards Organization)
- 1989 - se publică extensia SQL89 sau SQL1
- 1992 - se face o revizuire şi se obţine SQL2 sau SQL92

- 1999 - se complectează SQL cu posibilităţi de gestiune orientate obiect, rezultând SQL3 (sau SQL1999)
- 2003 - se adaugă noi tipuri de date şî noi funcţii, rezultând SQL2003.

Comanda SELECT este folosită pentru interogarea bazelor de date (obţinerea de informaţii). Această comandă este cea mai complexă din cadrul sistemelor ce conţin comenzi SQL. Comanda permite obţinerea de date din mai multe surse de date. Ea are, printre altele, funcţiile de selectie, proiectie, produs cartezian, join şi reuniune, intersecţie şi diferenţă din limbajul de interogare a bazelor de date relaţionale bazat pe algebra relaţiilor. Sintaxa comenzii este dată în continuare.

SELECT [**ALL** | **DISTINCT** | **TOP n [PERCENT]**] { ***** | **exp [AS câmp]** [**exp [AS câmp]**]... }

FROM sursa1 [alias] [,sursa2 [alias]]...

[WHERE condiţie]

[GROUP BY lista_câmpuri [HAVING condiţie]]

[[**UNION [ALL]** | **INTERSECT** | **EXCEPT**] **comanda_SELECT**]

[**ORDER BY** { **câmp** | **nrcâmp** } [{ **ASC** | **DESC** }] [**,ORDER BY** { **câmp** | **nrcâmp** } [{ **ASC** | **DESC** }]]...]

Această comandă selectează date din **sursele** de date precizate în clauza **FROM**. Pentru precizarea (calificarea) câmpurilor (dacă este necesar, deci dacă folosirea numai a numelui câmpului produce ambiguitate, adică există mai multe câmpuri cu acest nume în sursele de date) se poate folosi numele tabelului sau un nume sinonim (**alias local** numai în comanda SELECT) stabilit în **FROM** după numele sursei de date. Dacă se defineşte un "**alias**", atunci calificarea se face numai cu el (nu se va mai face cu numele tabelului).

O construcţie numită **sursa** poate fi:

1. un **tabel** sau **view** din baza de date
2. (**instrucţiune_select**)
3. **expresie_join**, sub forma:
   - **sursa1 [alias]** ***operator_join*** **sursa2 [alias] ON** ***condiţie_legatură***
   - **(expresie_join)**

O condiţie_elementară de legătură dintre două surse de date (precizate prin **expresie_tabel)** este de forma:

[alias_sursa1.]câmp1 ***operator*** **[alias_sursa2.]câmp2**

unde **operator** poate fi: **=, <>, >, >=, <, <=.** Cei doi termeni ai comparatiei trebuie să aparţină la tabele diferite.

Condiţiile de legătură dintre două surse de date sunt de forma:

- **cond_elementara [AND cond_elementara] ...**
- **(condiţie)**

O **expresie join** are ca rezultat un tabel şi este de forma:

Sursa1 $\left[\left\{ \begin{array}{c} \textbf{INNER} \\ \textbf{LEFT [OUTER]} \\ \textbf{RIGHT [OUTER]} \\ \textbf{FULL [OUTER]} \end{array} \right\} \right]$ **JOIN Sursa2 ON conditie**

Joinul condiţional, din algebra relaţională, notat prin **Sursa1** $\otimes_c$ **Sursa2**, este precizat prin **Sursa1 INNER JOIN sursa2 ON condiţie,** şi determină acele înregistrări din produsul cartezian al celor două surse care îndeplinesc condiţia din **ON**.

Joinul extern stânga, precizat prin **Sursa1 LEFT [OUTER] JOIN sursa2 ON condiţie,** determină o sursă nouă ce are ca atribute concatenarea atributelor din cele două surse, iar înregistrările se obţin astfel: se iau înregistrările care se obţin prin joinul condiţional **Sursa1** $\otimes_c$ **Sursa2**, la care se adaugă înregistrările din **sursa1** care nu s-au folosit la acest join condiţional combinate cu valoarea *null* pentru toate atributele corespunzătoare din **Sursa2**.

Joinul extern dreapta, precizat prin **Sursa1 RIGHT [OUTER] JOIN sursa2 ON condiţie,** determină o sursă nouă ce are ca atribute concatenarea atributelor din cele două surse, iar înregistrările se obţin astfel: se iau înregistrările care se obţin prin joinul condiţional **Sursa1** $\otimes_c$ **Sursa2**, la care se adaugă înregistrările din **sursa2** care nu s-au folosit la acest join condiţional combinate cu valoarea *null* pentru toate atributele corespunzătoare din **Sursa1**.

Joinul extern total, precizat prin **Sursa1 FULL [OUTER] JOIN sursa2 ON condiţie,** se obţine prin reuniunea rezultatelor obţinute de joinul extern stânga şi joinul extern dreapta.

Alte tipuri de expresii join:

- Sursa1 **JOIN** Sursa2 **USING** (lista_coloane)
- Sursa1 **NATURAL JOIN** Sursa2
- Sursa1 **CROSS JOIN** Sursa2

Dacă în clauza **FROM** apar mai multe **surse de date** (care se vor evalua la un tabel), atunci între un astfel de tabel - pe care îl vom numi **tabel principal**, şi celelalte tabele este indicat să existe anumite **legături** (stabilite prin **condiţii**). Plecând de la fiecare înregistrare a tabelului principal se determină înregistrările din celelalte tabele asociate prin astfel de legături (deci înregistrările ce verifică o condiţie). Dacă legătura (condiţia) nu se stabileşte, atunci se consideră că ea asociază toate înregistrările din celelalte tabele pentru fiecare înregistrare a tabelului principal (se consideră că valoarea condiţiei este **true** atunci când ea lipseşte). Această condiţie de legătură dintre sursele de date se precizează prin:

FROM sursa1[, sursa2] ... WHERE condiţie_legătură

Folosind sursele de date din FROM şi eventuala condiţie de legătură (dacă există mai multe surse de date) va rezulta un **tabel_rezultat**, cu coloanele obţinute prin concatenarea coloanelor din sursele de date, iar înregistrările sunt determinate după cum sunt explicate mai sus.

In **tabel_rezultat** se pot păstra toate înregistrările obţinute din sursele de date, sau se poate face o **filtrare** prin utilizarea unei condiţii de filtrare. Aceasta condiţie de filtrare va fi trecută în clauza **WHERE** în continuarea condiţiei de legătură. Cu o condiţie de filtrare condiţia din WHERE este de forma:

WHERE condiţie_filtrare

WHERE condiţie_legătură AND condiţie_filtrare

Condiţia de filtrare din clauza WHERE poate fi construită după următoarele reguli.

Condiţiile elementare de filtrare pot fi de una din formele următoare:

- **expresie operator_relational expresie**
- **expresie [NOT] BETWEEN valmin AND valmax**

  pentru a verifica dacă valoarea unei expresii este cuprinsă între două valori (*valmin* şi *valmax*) sau nu este cuprinsă între aceste valori (apare **NOT**)

- **câmp (NOT) LIKE şablon**

  După **LIKE** apare un *şablon* (ca un şir de caractere) ce precizează o mulţime de valori. In funcţie de sistemul de gestiune folosit, există un caracter în şablon ce precizează locul unui singur caracter necunoscut în câmp, sau un caracter în şablon ce precizează un şir neprecizat de caractere în câmp.

- **expresie** [**NOT**] **IN** { **valoare** [**valoare**]... / (**subselectie**) }

  Se verifică dacă valoarea expresiei apare (sau nu - cu **NOT**) într-o listă de valori sau într-o subselecţie. O **subselecţie** este o sursă de date generată cu comanda **SELECT** şi care are numai un singur câmp - cu valori de acelaşi tip cu valorile expresiei. Această condiţie corespunde testului de "**apartenenţă**" al unui element la o mulţime.

- **câmp operator_relational** { **ALL** / **ANY** / **SOME** } **(subselectie)**

  Valorile câmpului din stânga operatorului relaţional şi valorile singurului câmp din subselecţie trebuie să fie de acelaşi tip. Se obţine valoarea *adevărat* pentru condiţie dacă valoarea din partea stângă este în relaţia dată de operator pentru:

  - toate valorile din subselectie (apare **ALL**),
  - cel puţin o valoare din subselectie (apare **ANY** sau **SOME**).

  Condiţii echivalente:

  "expresie IN (subselecţie)" echivalent cu "**expresie = ANY (subselecţie)"**

  "expresie NOT IN (subselecţie)" echivalent cu "**expresie <> ALL (subselecţie)"**

- **[NOT] EXISTS (subselectie)**

  Cu **EXISTS** se obtine valoarea *adevărat* dacă în subselecţie există cel puţin o înregistrare, şi *fals* dacă subselecţia este vidă. Prezenţa lui **NOT** inversează valorile de adevăr.

- O **condiţie de filtrare** poate fi:
  - condiţie elementară
  - (condiţie)
  - **not** condiţie
  - condiţie1 **and** condiţie2
  - condiţie1 **or** condiţie2

O condiţie elementară poate avea una din valorile: **true, false, null**. Valoarea null se obţine dacă unul din operanzii utilizaţi are valoarea null. Valorile de adevăr pentru operatorii not, and, or sunt date în continuare:

	true	false	null
not	false	true	null

and	true	false	null
true	true	false	null
false	false	false	false
null	null	false	null

or	true	false	null
true	true	true	true
false	true	false	null
null	true	null	

Din această succesiune de valori se pot selecta toate câmpurile din toate tabelele (apare "*" după numele comenzii), sau se pot construi câmpuri ce au ca valoare rezultatul unor **expresii**. Câmpurile cu aceeaşi denumire în tabele diferite se pot califica prin numele sau alias-ul tabelelor sursă. Numele câmpului sau expresiei din tabelul rezultat este stabilit automat de sistem (în functie de expresia ce-l generează), sau se poate preciza prin clauza **AS** ce urmează expresiei (sau câmpului). In acest fel putem construi valori pentru o nouă înregistrare într-un **tabel_final**.

Expresiile se precizează cu ajutorul operanzilor (câmpuri, rezultatul unor funcţii) şi a operatorilor corespunzători tipurilor de operanzi.

In tabelul final se pot include toate sau numai o parte din înregistrări, după cum e precizat printr-un predicat ce apare în faţa listei de coloane::

- ALL - toate înregistrările
- DISTINCT - numai înregistrările distinte
- TOP n - primele n înregistrări
- TOP n PERCENT - primele n% înregistrări

Inregistrările din "**tabelul final**" se pot **ordona** crescător (ASC) sau descrescător (DESC) după valorile unor câmpuri, precizate în clauza **ORDER BY**. Câmpurile se pot preciza prin nume sau prin poziţia lor (numărul câmpului) în lista de câmpuri din comanda **SELECT** (precizarea prin poziţie este obligatorie atunci când se doreşte sortarea după valorile unei expresii). Ordinea câmpurilor din această clauză precizează prioritatea cheilor de sortare.

Mai multe înregistrări consecutive din "tabelul final* pot fi **grupate** într-o singură înregistrare, deci un grup de înregistrări se înlocuieşte cu o singură înregistrare. Un astfel de grup este precizat de **valorile comune** ale câmpurilor ce apar în clauza **GROUP BY**. "**Tabelul nou**" se sortează (automat de către sistem) creascător după valorile câmpurilor din **GROUP BY**. Inregistrările consecutive din fişierul astfel sortat, ce au aceeaşi valoare pentru toate câmpurile din **GROUP BY**, se înlocuiesc cu o singură înregistrare. Prezenţa unei astfel de înregistrări poate fi condiţionată de valoarea *adevărat* pentru o condiţie ce se trece în clauza **HAVING**.

Pentru grupul de înregistrări astfel precizat (deci pentru o mulţime de valori) se pot folosi următoarele funcţii:

- **AVG** $\left(\left[\left\{\begin{matrix}\textbf{ALL}\\ \textbf{DISTINCT}\end{matrix}\right\}\right]\textbf{câmp}\right)$ sau **AVG([ALL]) expresie)**

  Pentru grupul de înregistrări se iau toate valorile (cu **ALL**, care este şi valoarea implicită) sau numai valorile distincte (apare **DISTINCT**) ale câmpului sau expresiei

numerice precizate şi din aceste valori se determină **valoarea medie**.

- **COUNT** $\left(\left\{ \left[\left\{ \begin{matrix} \textbf{ALL} \\ \textbf{DISTINCT} \end{matrix} \right\} \right] \textbf{câmp} \right\} \right)$

  Această funcţie determină **numărul** de înregistrări din grup (apare '*'), numărul de valori ale unui câmp (apare **ALL**, identic cu '*'), sau numărul de înregistrări distincte din grup (cu **DISTINCT**).

- **SUM** $\left(\left[\left\{ \begin{matrix} \textbf{ALL} \\ \textbf{DISTINCT} \end{matrix} \right\} \right] \textbf{câmp} \right)$ sau **SUM([ALL]) expresie)**

  Pentru înregistrările din grup se face **suma** valorilor unui câmp sau ale unei expresii numerice (deci numărul de termeni este dat de numărul de înregistrări din grup) sau suma valorilor distincte ale câmpului.

- $\left\{ \begin{matrix} \textbf{MAX} \\ \textbf{MIN} \end{matrix} \right\} \left(\left[\left\{ \begin{matrix} \textbf{ALL} \\ \textbf{DISTINCT} \end{matrix} \right\} \right] \textbf{câmp} \right)$ sau $\left\{ \begin{matrix} \textbf{MAX} \\ \textbf{MIN} \end{matrix} \right\}$ **([ALL]) expresie)**

  Pentru fiecare înregistrare din grup se determină valoarea unei expresii sau câmp şi se află valoarea **maximă** sau **minimă** dintre aceste valori.

Cele cinci funcţii amintite mai sus (**AVG, COUNT, SUM, MIN, MAX**) pot apare atât în expresiile ce descriu câmpurile din fişierul rezultat, cât şi în clauza **HAVING**. Deoarece aceste funcţii se aplică unui grup de înregistrări, în comada **SELECT** acest grup trebuie generat de clauza **GROUP BY**. Dacă această clauză lipseşte, atunci **întregul "tabel final" constituie un grup**, deci tabelul rezultat va avea o singură înregistrare.

In general nu este posibilă selectarea câmpurilor singure (fără rezultat al funcţilor amintite) decât numai dacă au fost trecute în **GROUP BY**. Dacă totuşi apar, **şi această folosire nu produce eroare**, atunci se ia o valoare oarecare, pentru o înregistrare din grup.

Două tabele cu acelaşi număr de câmpuri (coloane) şi cu acelaşi tip pentru valorile câmpurilor aflate pe aceleaşi poziţii se pot **reuni** într-un singur tabel obţinut cu ajutorul operatorului **UNION**. Din tabelul rezultat obţinut se pot păstra toate înregistrările (apare **ALL**) sau numai cele distincte (nu apare **ALL**). Clauza **ORDER BY** poate apare numai pentru ultima selecţie. Nu se pot combina subselecţii prin clauza **UNION**.

Intre două rezultate (mulţimi de înregistrări) obţinute cu instrucţiuni SELECT se poate folosi operatorul INTERSEC sau EXCEPT (sau MINUS).

Clauzele din instrucţiunea SELECT trebuie să fie în ordinea: lista_expresii FROM ... WHERE ... HAVING ... ORDER BY ...

O comandă se poate memora în baza de date ca o componentă numită **view**. Definirea este:

CREATE VIEW nume_view AS comanda_SELECT

2.4. Probleme propuse

I.

a. **Se cere o bază de date relaţională**, cu tabele în 3NF, ce gestionează următoarele informaţii dintr-o firmă de soft:

- **activităţi**: cod activitate, descriere, tip activitate;
- **angajaţi**: cod angajat, nume, listă activităţi, echipa din care face parte, liderul echipei;

unde:

- o **activitate** este identificată prin "cod activitate";
- un **angajat** este identificat prin "cod angajat";
- un angajat face parte dintr-o singură **echipă**, iar echipa are un lider, care la rândul său este angajat al firmei;
- un angajat poate să participe la realizarea mai multor activităţi, iar la o activitate pot să participe mai mulţi angajaţi;

Justificaţi că tabelele obţinute sunt în 3NF.

b. Pentru baza de date de la punctul **a**, să se rezolve, folosind algebra relaţională **sau** Select-SQL, următoarele interogări:

b1. Numele angajaţilor care lucrează la cel puţin o activitate de tipul *"Proiectare"* şi **nu** lucrează la nici o activitate de tipul *"Testare"*.

b2. Numele angajaţilor care sunt liderii unei echipe cu cel puţin 10 angajaţi.

II.

a. Se cere o bază de date relaţională, cu tabele în 3NF, ce gestionează următoarele informaţii dintr-o facultate:

- **discipline**: cod, denumire, număr credite, lista studenţilor care au dat examen;
- **studenti**: cod, nume, data naşterii, grupa, anul de studiu, specializarea, lista disciplinelor la care a dat examene (inclusiv data examenului şi nota obţinută).

Justificaţi că tabelele sunt în 3NF.

b. Pentru baza de date de la punctul **a**, folosind algebra relaţională şi instructiuni SELECT-SQL, se cer studenţii (nume, grupa, nr.discipline promovate) ce au promovat în anul 2013 peste 5 discipline. Dacă un student are la o disciplină mai multe examene cu note de promovare, atunci disciplina se numără o singură dată.

III.

a. Se cere o bază de date relaţională, cu tabele în 3NF, ce gestionează următoarele informaţii despre absolvenţii înscrişi pentru examnul de licenţă: Nr.matricol, Cod şi denumire secţie absolvită, Titlul lucrării, Cod şi nume cadru didactic îndrumător, Cod şi denumire departament de care aparţine cadrul didactic îndrumător, Lista de resurse soft necesare pentru susţinerea lucrării (ex. C#.Net, C++, etc.), Lista de resurse hard necesare pentru susţinerea lucrării (Ram 8Gb, DVD Reader, etc.). Justificaţi că tabelele sunt în 3NF.

b. Pentru baza de date de la punctul **a**, folosind algebra relaţională şi instructiuni SELECT-SQL cel puţin o dată fiecare, se cer următoarele liste (explicaţi modul de obţinere a listelor):

i. Absolvenţii (nume, titlu lucrare, nume cadru didactic) pentru care cadrul didactic îndrumător aparţine de un departament dat prin denumire.
ii. Pentru departament se cere numărul de absolvenţi care au cadrul didactic îndrumător de la acest departament.
iii. Cadrele didactice care nu au îndrumat absolvenţi la lucrarea de licenţă.
iv. Numele absolvenţilor care au nevoie de urmatoarele două resurse soft: Oracle şi C#.

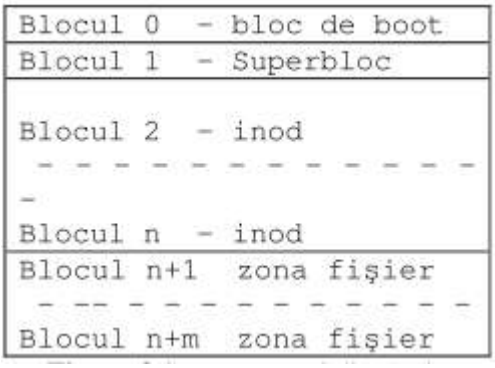
3. Sisteme de operare

3.1. Structura sistemelor de fişiere Unix

3.1.1. Structura Internă a Discului UNIX

3.1.1.1. Partiţii şi Blocuri

Un sistem de fişiere Unix este găzduit fie pe un periferic oarecare (hard-disc, CD, dischetă etc.), fie pe o partiţie a unui hard-disc. Partiţionarea unui hard-disc este o operaţie (relativ) independentă de sistemul de operare ce va fi găzduit în partiţia respectivă. De aceea, atât partţiilor, cât şi suporturilor fizice reale le vom spune generic, discuri Unix.



Structura unui disc UNIX

Un fişier Unix este o succesiune de octeţi, fiecare octet putând fi adresat în mod individual. Este permis atât accesul secvenţial, cât şi cel direct. Unitatea de schimb dintre disc şi memorie este blocul. La sistemele mai vechi acesta are 512 octeţi, iar la cele mai noi până la 4Ko, pentru o mai eficientă gestiune a spaţiului. Un sistem de fişiere Unix este o structură de date rezidentă pe disc. Aşa după cum se vede din figura de mai sus, un disc este compus din patru categorii de blocuri.

Blocul 0 conţine programul de încărcare al SO. Acest program este dependent de maşina sub care se lucrează.

Blocul 1 este numit şi superbloc. In el sunt trecute o serie de informaţii prin care se defineşte sistemul de fişiere de pe disc. Printre aceste informaţii amintim:

-numărul n de inoduri (detaliem imediat);
-numărul de zone definite pe disc;
-pointeri spre harta de biţi a alocării inodurilor;
-pointeri spre harta de biţi a spaţiului liber disc;
-dimensiunile zonelor disc etc.

Blocurile 2 la n, unde n este o constantă a formatării discului. Un inod (sau i-nod) este numele, în terminologia Unix, a descriptorului unui fişier. Inodurile sunt memorate pe disc sub forma unei liste (numităi-listă). Numărul de ordine al unui inod în cadrul i-listei se

reprezintă pe doi octeţi şi se numeşte i-număr. Acest i-număr constituie legătura dintre fişier şi programele utilizator.

Partea cea mai mare a discului este rezervată zonei fişierelor. Alocarea spaţiului pentru fişiere se face printr-o variantă elegantă de indexare. Informaţiile de plecare pentru alocare sunt fixate în inoduri.

3.1.1.2. Directori şi I-noduri

Structura unei intrări într-un fişier director este ilustrată în figura de mai jos

Numele fişierului (practic oricât de lung)	inumăr

Structura unei intrări în director

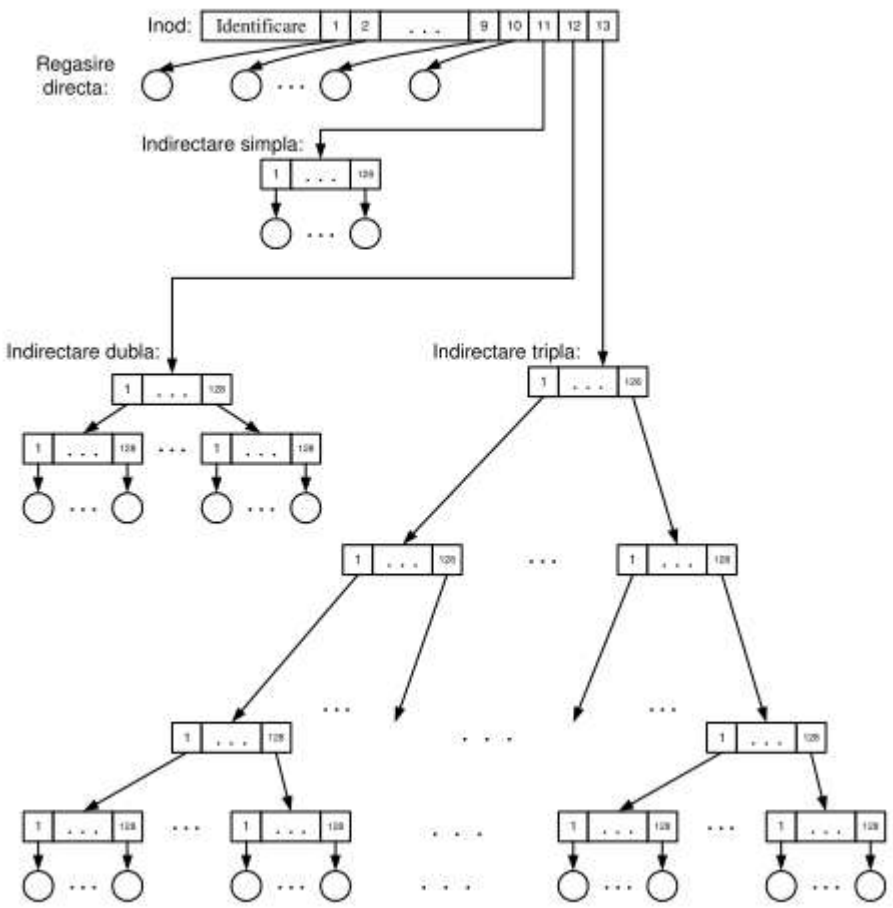
Deci, în director se află numele fişierului şi referinţa spre inodul descriptor al fişierului.
Un inod are, de regulă, între 64 şi 128 de octeţi şi el conţine informaţiile din tabelul următor:

mode	Drepturile de acces şi tipul fişierului.
link count	Numărul de directoare care conţin referiri la acest inumăr, adică numărul de legături spre acest fişier.
user ID	Numărul (UID) de identificare a proprietarului.
group ID	Numărul (GID) de identificare a grupului.
size	Numărul de octeți (lungimea) fişierului.
access time	Momentul ultimului acces la fişier.
mod time	Momentul ultimei modificări a fişierului.
inode time	Momentul ultimei modificări a structurii inodului.
block list	Lista adreselor disc pentru primele blocuri care aparţin fişierului.
indirect list	Referințe către celelalte blocuri care aparţin fişierului.

3.1.1.3. Schema de alocare a blocurilor disc pentru un fişier

Fiecare sistem de fişiere Unix are câteva constante proprii, printre care amintim: lungimea unui bloc, lungimea unui inod, lungimea unei adrese disc, câte adrese de prime blocuri se înregistrează direct în inod şi câte referinţe se trec în lista de referinţe indirecte. Indiferent de valorile acestor constante, principiile de înregistrare / regăsire sunt aceleaşi.

Pentru fixarea ideilor, vom alege aceste constante cu valorile întâlnite mai frecvent la sistemele de fişiere deja consacrate. Astfel, vom presupune că un bloc are lungimea de 512 octeţi. O adresă disc se reprezintă pe 4 octeţi, deci într-un bloc se pot înregistra 128 astfel de se adrese. In inod trec direct primele 10 adrese de blocuri, iar lista de adrese indirecte are 3 elemente. Cu aceste constante, în figura de mai jos este prezentată structura pointerilor spre blocurile ataşate unui fişier Unix.



Structura unui inod şi accesul la blocurile unui fişier

In inodul fişierului se află o listă cu 13 intrări, care desemnează blocurile fizice aparţinând fişierului.

• Primele 10 intrări conţin adresele primelor 10 blocuri de câte 512 octeţi care aparţin fişierului.

• Intrarea nr. 11 conţine adresa unui bloc, numit bloc de indirectare simplă. El conţine adresele următoarelor 128 blocuri de câte 512 octeţi, care aparţin fişierului.

• Intrarea nr. 12 conţine adresa unui bloc, numit bloc de indirectare dublă. El conţine adresele a 128 blocuri de indirectare simplă, care la rândul lor conţin, fiecare, adresele a câte 128 blocuri, de 512 octeţi fiecare, cu informaţii aparţinând fişierului.

• Intrarea nr. 13 conţine adresa unui bloc, numit bloc de indirectare triplă. In acest bloc sunt conţinute adresele a 128 blocuri de indirectare dublă, fiecare dintre acestea conţinând adresele a câte 128 blocuri de indirectare simplă, iar fiecare dintre acestea conţine adresele a câte 128 blocuri, de câte 512 octeţi, cu informaţii ale fişierului.

In figura de mai sus am ilustrat prin cercuri blocurile de informaţie care aparţin fişierului, iar prin dreptunghiuri blocurile de referinţe, în interiorul acestora marcând referinţele. Numărul de accese necesare pentru a obţine direct un octet oarecare este cel mult 4. Pentru fişiere mici acest număr este şi mai mic. Atât timp cât fişierul este deschis, inodul lui lui este prezent în memoria internă. Tabelul următor dă numărul maxim de accese la disc pentru a obţine, în acces direct orice octet dintr-un fişier, în funcţie de lungimea fişierului.

Lungime maximă (blocuri)	Lungime maximă (octeţi)	Accese indirecte	Accese la informaţie	Total accese
10	5120	-	1	1
10+128 = 138	70656	1	1	2
10+128+128^2 = 16522	8459264	2	1	3
10+128+128^2+128^3= 2113674	1082201088	3	1	4

La sistemele Unix actuale lungimea unui bloc este de 4096 octeţi care poate înregistra 1024 adrese, iar în inod se înregistrează direct adresele primelor 12 blocuri. In aceste condiţii, tabelul de mai sus se transformă în:

Lungime maximă (blocuri)	Lungime maximă (octeţi)	Accese indirecte	Accese la informaţie	Total accese
12	49152	-	1	1
12+1024 = 1036	4243456	1	1	2
12++1024+1024 ^2 = 1049612	4299210752	2	1	3
12+1024+1024 ^2+1024^3 = 1073741824	4398046511104 (peste 5000Go)	3	1	4

3.1.2. Tipuri de fişiere şi sisteme de fişiere

In cadrul unui sistem de fişiere, apelurile sistem Unix gestionează opt tipuri de fişiere şi anume:

1. Normale (obişnuite)
2. Directori
3. Legături hard (hard links)
4. Legături simbolice (symbolic links)
5. Socketuri (sockets)
6. FIFO - pipe cu nume (named pipes)
7. Periferice caracter
8. Periferice bloc

Pe lângă aceste opt tipuri, mai există încă patru entităţi, pe care apelurile sistem le văd, din punct de vedere sintactic, tot ca şi fişiere. Aceste entităţi sunt gestionate de nucleul Unix, au suportul fizic tot în nucleu şi folosite la comunicări între procese. Aceste entităţi sunt:

9. Pipe (anonymous pipes)
10. Segmente de memorie partajată
11. Cozi de mesaje
12. Semafoare

Fişierele obişnuite sunt privite ca şiruri de octeţi, accesul la un octet putându-se face fie secvenţial, fie direct prin numărul de ordine al octetului.

Fişierele directori. Un fişier director se deosebeşte de un fişier obişnuit numai prin informaţia conţinută în el. Un director conţine lista de nume şi adrese pentru fişierele subordonate lui. Uzual, fiecare utilizator are un director propriu care punctează la fişierele lui obişnuite, sau la alţi subdirectori definiţi de el.

Fişierele speciale. In această categorie putem include, pentru moment, ultimele 6 tipuri de fişiere. In particular, Unix priveşte fiecare dispozitiv de I/O ca şi un fişier de tip special. Din punct de vedere al utilizatorului, nu există nici o deosebire între lucrul cu un fişier disc normal şi lucrul cu un fişier special.

Fiecare director are două intrări cu nume speciale şi anume:

"." (punct) denumeşte generic (punctează spre) însuşi directorul respectiv;

".." (două puncte succesive), denumeşte generic (punctează spre) directorul părinte.

Fiecare sistem de fişiere conţine un director principal numit **`root`** sau **`/`**.

In mod obişnuit, fiecare utilizator foloseşte un *director curent*, ataşat utilizatorului la intrarea în sistem. Utilizatorul poate să-şi schimbe acest director (`cd`), poate crea un nou director subordonat celui curent, (`mkdir`), să şteargă un director (`rmdir`), să afişeze *calea* de acces de la `root` la un director sau fişier (`pwd`) etc.

Apariţia unui mare număr de distribuitori de Unix a condus, inevitabil, la proliferarea unui număr oarecare de "*sisteme de fişiere extinse*" proprii acestor distribuitori. De exemplu:

- Solaris utilizează sistemul de fişiere `ufs`;
- Linux utilizează cu precădere sistemul de fişiere `ext2` şi mai nou, `ext3`;
- IRIX utilizează `xfs`
- etc.

Actualele distribuţii de Unix permit utilizatea unor sisteme de fişiere proprii altor sisteme de operare. Printre cele mai importante amintim:

- Sistemele FAT şi FAT32 de sub MS-DOS şi Windows 9x;
- Sistemul `NTFS` propriu Windows NT şi 2000.

Din fericire, aceste extinderi sunt transparente pentru utilizatorii obişnuiţi. Totuşi, se recomandă prudenţă atunci când se efectuează altfel de operaţii decât citirea din fişierele create sub alte sisteme de operare decât sistemul curent. De exemplu, modificarea sub Unix a unui octet într-un fişier de tip `doc` creat cu Word sub Windows poate uşor să compromită fişierul aşa încât el să nu mai poată fi exploatat sub Windows!

Administratorii sistemelor Unix trebuie să ţină cont de sistemele de fişiere pe care le instalează şi de drepturile pe care le conferă acestora vis-a-vis de userii obişnuiţi.

Principiul structurii arborescente de fişiere este acela că orice fişier sau director are un singur părinte. Automat, pentru fiecare director sau fişier există o singură cale (path) de la rădăcină la directorul curent. Legătura între un director sau fişier şi părinte o vom numi *legătură naturală*. Evident ea se creează odată cu crearea directorului sau fişierului respectiv.

3.1.2.1. Legături hard şi legături simbolice

In anumite situaţii este utilă partajarea unei porţiuni a structurii de fişiere între mai mulţi utilizatori. De exemplu, o bază de date dintr-o parte a structurii de fişiere trebuie să fie accesibilă mai multor utilizatori. Unix permite o astfel de partajare prin intermediul *legăturilor suplimentare*. O legătură suplimentară permite referirea la un fişier pe alte căi decât pe cea naturală. Legăturile suplimentare sunt de două feluri: *legături hard* şi *legături simbolice (soft).*

Legăturile hard sunt identice cu legăturile naturale şi ele pot fi create numai de către administratorul sistemului. O astfel de legătură este o intrare într-un director care punctează spre o substructură din sistemul de fişiere spre care punctează deja legătura lui naturală. Prin aceasta, substructura este văzută ca fiind descendentă din două directoare diferite! Deci, printr-o astfel de legătură un fişier primeşte efectiv două nume. Din această cauză, la parcurgerea unei structuri arborescente, fişierele punctate prin legături hard apar duplicate. Fiecare duplicat apare cu numărul de legături către el.

De exemplu, dacă există un fişier cu numele `vechi`, iar administratorul dă comanda:

```
#ln  vechi  linknou
```

atunci în sistemul de fişiere se vor vedea două fişiere identice: `vechi` si `linknou`, fiecare dintre ele având marcat faptul că sunt două legături spre el.

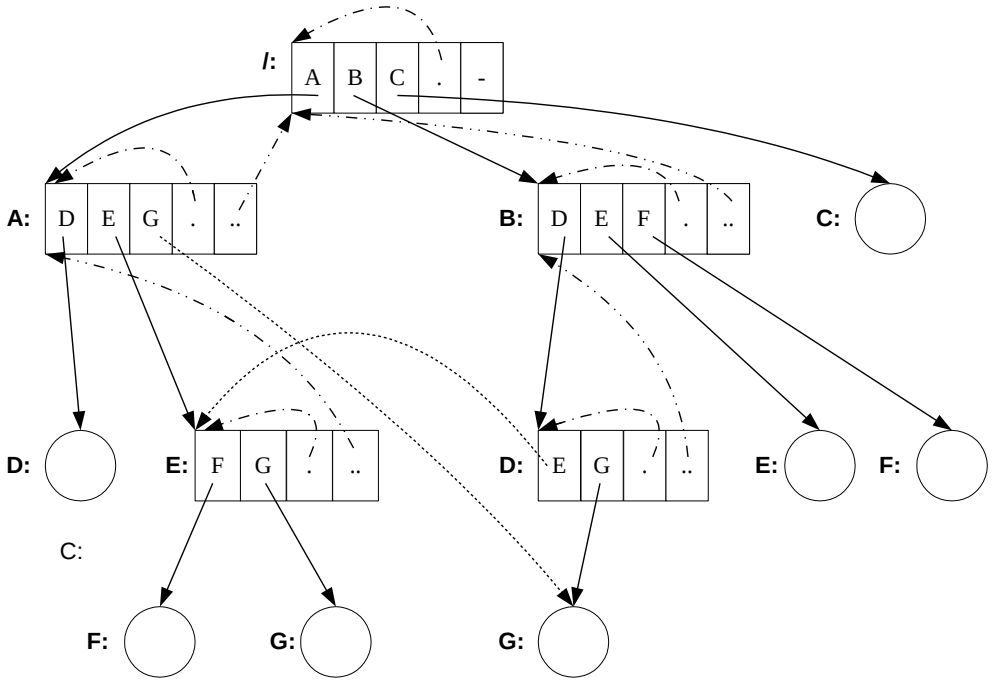
Legăturile hard pot fi făcute numai în interiorul aceluiaşi sistem de fişiere (detalii puţin mai târziu).

Legăturile simbolice sunt intrări speciale într-un director, care punctează (referă) un fişier (sau director) oarecare în structura de directori. Această intrare se comportă ca şi un subdirector al directorului în care s-a creat intrarea.

In forma cea mai simplă, o legătură simbolică se creează prin comanda:

```
ln  - s  caleInStructuraDeDirectori  numeSimbolic
```

După această comandă, `caleInStructuraDeDirectori` va avea marcată o legătură în plus, iar numeSimbolic va indica (numai) către această cale. Legăturile simbolice pot fi utilizate şi de către userii obişnuiţi. De asemenea, ele pot puncta şi înafara sistemului de fişiere.



Structura arborescentă împreună cu legăturile simbolice sau hard conferă sistemului de fişiere Unix o structură de graf aciclic. In exemplul din figura de mai sus este prezentat un exemplu simplu de structură de fişiere. Prin literele mari A, B, C, D, E, F, G am indicat nume de fişiere obişnuite, nume de directori şi nume de legături. Este evident posibil ca acelaşi nume să apară de mai multe ori în structura de directori, graţie structurii de directori care elimină ambiguităţile. Fişierele obişnuite sunt reprezentate prin cercuri, iar fişierele directori prin dreptunghiuri.

Legăturile sunt reprezentate prin săgeţi de trei tipuri:

- linie continuă – legăturile naturale;
- linie întreruptă – spre propriul director şi spre părinte;
- linie punctată – legături simbolice sau hard.

In exemplul de mai sus există 12 noduri - fişiere obişnuite sau directori. Privit ca un arbore, deci considerând numai legăturile naturale, el are 7 ramuri şi 4 nivele.

Să presupunem că cele două legături (desenate cu linie punctata) sunt simbolice. Pentru comoditate, vom nota legătura simbolică cu ultima literă din specificarea căii. Crearea celor două legături se poate face, de exemplu, prin succesiunea de comenzi:

```
cd  /A
ln  -s  /A/B/D/G  G          Prima legătură
cd  /A/B/D
ln  -s  /A/E  E          A doua legătură
```

Să presupunem acum că directorul curent este B. Vom parcurge arborele în ordinea director urmat de subordonaţii lui de la stânga spre dreapta. Următoarele 12 linii indică toate cele 12 noduri din structură. Pe aceeaşi linie apar, atunci când este posibil, mai multe specificări ale aceluiaşi nod. Specificările care fac uz de legături simbolice sunt subliniate. Cele mai lungi 7 ramuri vor fi marcate cu un simbol # în partea dreaptă.

/	..				
/A	../A				
/A/D	../A/D				#
/A/E	../A/E	D/E	./D/E		
/A/E/F	../A/E/F	D/E/F	./D/E/F		#
/A/E/G	../A/E/G	D/E/G	./D/E/G		#
/B	.				
/B/D	D	./D			
/B/D/G	D/G	./D/G	/A/G	../A/G	#
/B/E	E	./E			#
/B/F	F	./F			#
/C	../C				#

Ce se întâmplă cu ştergerea în cazul legăturilor multiple? De exemplu, ce se întâmplă când se execută una dintre următoarele două comenzi?

```
rm  D/G
rm  /A/G
```

Este clar că fişierul trebuie să rămână activ dacă este şters numai de către una dintre specificări.

Pentru aceasta, în descriptorul fişierului respectiv există un câmp numit *contor de legare.* Acesta are valoarea 1 la crearea fişierului şi creşte cu 1 la fiecare nouă legătură. La ştergere, se radiază legătura din directorul părinte care a cerut ştergerea, iar contorul de legare scade cu 1. Abia dacă acest contor a ajuns la zero, fişierul va fi efectiv şters de pe disc şi blocurile ocupate de el vor fi eliberate.

3.2. Procese Unix

Procese Unix: creare, funcţiile fork, exec, exit, wait; comunicare prin pipe şi FIFO

3.2.1. Principalele apeluri system de gestiune a proceselor

In secţiunile din acest subcapitol vom prezenta cele mai importante apeluri sistem pentru lucrul cu procese: `fork, exit, wait` şi `exec*`. Incepem cu `fork()`, apelul sistem pentru crearea unui proces.

3.2.1.1. Crearea proceselor Unix. Apelul fork

In sistemul de operare Unix un proces se creează prin apelul funcţiei sistem `fork()`. La o funcţionare normală, efectul acesteia este următorul*: se copiază imaginea procesului într-o zonă de memorie liberă, această copie fiind noul proces creat, în prima fază identic cu procesul iniţial*. Cele două procese îşi continuă execuţia în paralel cu instrucţiunea care urmează apelului `fork`.

Procesul nou creat poartă numele de *proces fiu*, iar procesul care a apelat funcţia `fork`() se numeşte *proces părinte*. Exceptând faptul că au spaţii de adrese diferite, procesul fiu diferă de procesul părinte doar prin identificatorul său de proces PID, prin identificatorul procesului părinte PPID şi prin valoarea returnată de apelul `fork`(). *La derulare normală, un apel `fork()` întoarce în procesul părinte (procesul care a lansat apelul `fork()`) pid-ul noului proces fiu, iar în procesul fiu, întoarce valoarea 0.*

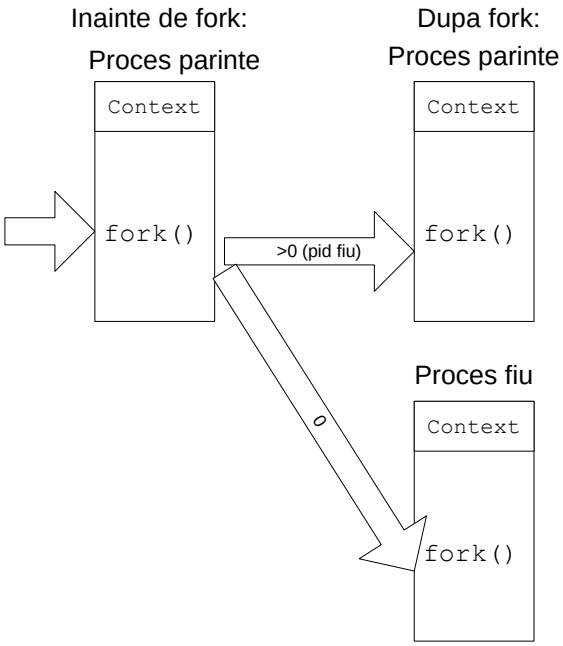


Figura 3.1 Mecanismul fork

In figura de mai sus am ilustrat acest mecanism, unde săgeţile indică instrucţiunea care se execută în mod curent în proces.

In caz de eşec, `fork` întoarce valoarea –1 şi desigur, setează corespunzător variabila `errno`. Eşecul apelului `fork` poate să apară dacă:

- nu există memorie suficientă pentru efectuarea copiei imaginii procesului părinte;
- numărul total de procese depăşeste o limită maximă admisă.

Acest comportament al lui `fork` permite descrierea uşoară a două secvenţe de instrucţiuni care să se deruleze în paralel, sub forma:

```
if ( fork() == 0 )
{ - - - instrucţiuni ale procesului fiu - - - }
else
{ - - - instrucţiuni ale procesului tată - - - }
```

Programul următor ilustrează utilizarea lui fork:

```
main(){
   int pid,i;
   printf("\nInceputul programului:\n");
   if ((pid=fork())<0) err_sys("Nu pot face fork()\n");
   else if (pid==0){//Suntem in fiu
      for (i=1;i<=10;i++){
```

```
         sleep(2);      //dormim 2 secunde
         printf("\tFIUL(%d) al PARINTELUI(%d):3*%d=%d\n",
                                getpid(),getppid(),i,3*i);
      }
      printf("Sfarsit FIU\n");
   }
   else if (pid>0){//Suntem in parinte
      printf("Am creat FIUL(%d)\n",pid);
      for (i=1;i<=10;i++){
         sleep(1);      //dormim 1 secunda
         printf("PARINTELE(%d):2*%d=%d\n",getpid(),i,2*i);
      }
      printf("Sfarsit PARINTE\n");
   }
}
```

In mod intenţionat, am făcut astfel încât procesul fiu să aştepte mai mult decât părintele (în cazul calculelor complexe apar adesea situaţii în care operaţiile unuia dintre procese durează mai mult în timp). Ca urmare, părintele va termina mai repede execuţia. Rezultatele obţinute sunt:

```
Inceputul programului:
Am creat FIUL(20429)
PARINTELE(20428): 2*1=2
         FIUL(20429) al PARINTELUI(20428): 3*1=3
PARINTELE(20428): 2*2=4
PARINTELE(20428): 2*3=6
         FIUL(20429) al PARINTELUI(20428): 3*2=6
PARINTELE(20428): 2*4=8
PARINTELE(20428): 2*5=10
         FIUL(20429) al PARINTELUI(20428): 3*3=9
PARINTELE(20428): 2*6=12
PARINTELE(20428): 2*7=14
         FIUL(20429) al PARINTELUI(20428): 3*4=12
PARINTELE(20428): 2*8=16
PARINTELE(20428): 2*9=18
         FIUL(20429) al PARINTELUI(20428): 3*5=15
PARINTELE(20428): 2*10=20
Sfarsit PARINTE
         FIUL(20429) al PARINTELUI(1): 3*6=18
         FIUL(20429) al PARINTELUI(1): 3*7=21
         FIUL(20429) al PARINTELUI(1): 3*8=24
         FIUL(20429) al PARINTELUI(1): 3*9=27
         FIUL(20429) al PARINTELUI(1): 3*10=30
Sfarsit FIU
```

3.2.1.2. Execuţia unui program extern; apelurile exec

Aproape toate sistemele de operare şi toate mediile de programare oferă, într-un fel sau altul, mecanisme de lansare a unui program din interiorul altuia. Unix oferă acest mecanism prin intermediul familiei de apeluri sistem `exec*`. După cum se va vedea, utilizarea combinată de `fork – exec` oferă o mare elasticitate manevrării proceselor.

Apelurile sistem din familia `exec` lansează un nou program în cadrul aceluiaşi proces. *Apelului `exec` i se furnizează numele unui fişier executabil, iar conţinutul acestuia se suprapune peste programul procesului existent*, aşa cum se vede în Figura 3.2.

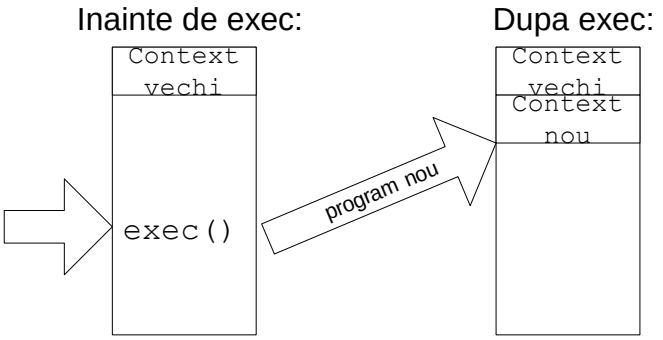


Figure 3.2 Mecanismul exec

In urma lui `exec` instrucţiunile aflate în programul curent nu se mai execută, în locul lor se lansează instrucţiunile noului program.

Unix oferă, în funcţie de trei criterii, şase astfel de apeluri. Criteriile sunt:

- Specificarea căii spre programul executabil ce va fi lansat: absolută sau relativă la directoarele indicate prin variabila de mediu PATH.
- Mediul este moştenit sau se creează un nou mediu.
- Specificarea argumentelor din linia de comandă se face printr-o listă explicită sau printr-un vector de pointeri spre aceste argumente.

Din cele opt posibile, s-au eliminat cele două cu cale relativă şi mediu nou. Prototipurile celor şase apeluri `exec*()` sunt:

```
int execv (char *fisier, char *argv[]);
int execl (char *fisier, char *arg0, …, char *argn,
NULL);
int execve(char *fisier, char *argv[], char *envp[]);
int execle(char *fisier, char *arg0, …, char *argn, NULL,
          char *envp[]);
int execvp(char *fisier, char *argv[]);
int execlp(char* fisier, char *arg0, …, char *argn,
NULL);
```

Semnificaţia parametrilor `exec` este următoarea:

- `fisier` - numele fişierului executabil care va înlocui programul curent. El trebuie să coincidă cu argumentul `argv[0]` sau `arg0`.
- `argv` este tabloul de pointeri, terminat cu un pointer NULL, care conţine argumentele liniei de comandă pentru noul program lansat în execuţie.
- `arg0, arg1, ... , argn, NULL` conţine argumentele liniei de comandă scrise explicit ca şi stringuri; această secvenţă trebuie terminată cu NULL.
- `envp` este tabloul de pointeri, terminat cu un pointer NULL, care conţine stringurile corespunzătoare noilor variabile de mediu sub forma “nume=valoare”.

3.2.1.3. Apelurile exit şi wait

Apelul sistem:

```
exit(int n)
```

provoacă terminarea procesului curent şi revenirea la procesul părinte (cel care l-a creat prin `fork`). Intregul *n* precizează *codul de retur* cu care se termină procesul. In cazul în care procesul părinte nu mai există, procesul este trecul în starea *zombie* şi este subordonat automat procesului special `init` (care are PID-ul 1).

Aşteptarea terminării unui proces se realizează folosind unul dintre apelurile sistem `wait()` sau `waitpid()`. Prototipurile acestora sunt:

```
pid_t wait(int *stare)
pid_t waitpid(pid_t pid, int *stare, int optiuni);
```

Apelul `wait()` suspendă execuţia programului până la terminarea unui proces fiu. Dacă fiul s-a terminat înainte de apelul `wait()`, apelul se termină imediat. La terminare, toate resursele ocupate de procesul fiu sunt eliberate.

3.2.2. Comunicarea între procese prin pipe

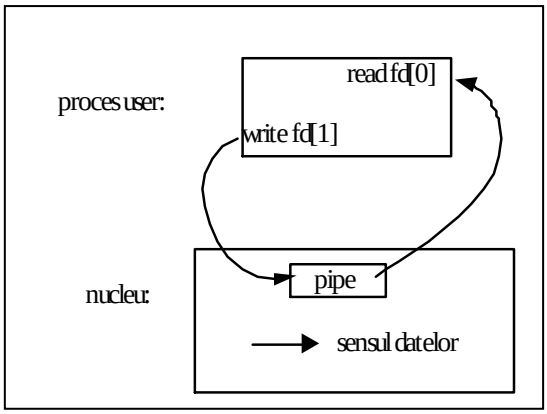
3.2.2.1. Conceptul de pipe

Conceptul a apărut prima dată sub Unix, pentru a permite unui proces fiu să comunice cu părintele său. De obicei procesul părinte redirectează ieşirea sa standard, `stdout`, către un *pipe*, iar procesul fiu îşi redirectează intrarea standard, `stdin`, din acelaşi *pipe*. In majoritatea sistemelor de operare se foloseşte operatorul "|" pentru specificarea acestui gen de conexiuni între comenzi ale sistemului de operare.

Un *pipe* Unix este un flux unidirecţional de date, gestionat de către nucleul sistemului. De fapt, în nucleu se rezervă un buffer de minimum 4096 octeţi în care octeţii sunt gestionaţi aşa cum am descris mai sus. Crearea unui *pipe* se face prin apelul sistem:

```
int pipe(int fd[2]);
```

Intregul `fd[0]` se comportă ca un întreg care identifică descriptorul pentru citirea din "fişierul" *pipe,* iar `fd[1]` ca un întreg care indică descriptorul pentru scriere în *pipe*. In urma creării, legătura user – nucleu prin acest *pipe* apare ca în figura de mai jos.



Evident, un pipe într-un singur proces nu are sens. Este însă esenţială funcţionarea lui `pipe` combinată cu `fork`. Astfel, dacă după crearea lui *`pipe`* se execută un `fork`, atunci legătura celor două procese cu *pipe* din nucleu apare ca în figura următoare.

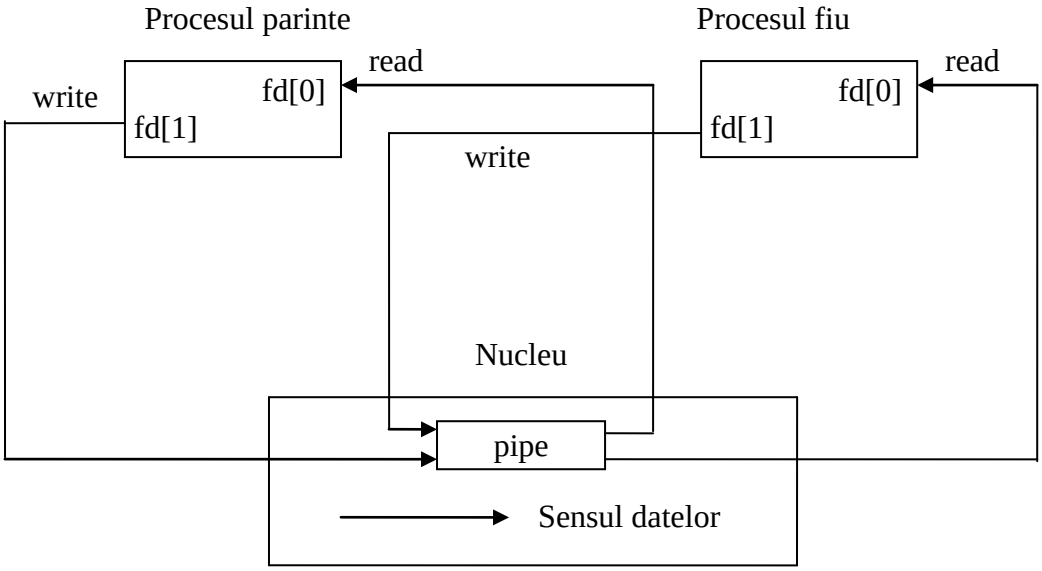


Figura 3.3 Un *pipe* leagă două procese înrudite

Asigurarea unidirecţionalităţii unui *pipe* cade exclusiv în sarcina programatorului. Astfel, pentru a se asigura sensul datelor în exemplul de mai sus, se impune ca înainte de a transmite prin *pipe*:

- In procesul părinte să se apeleze `close(fd[0]);`
- In procesul fiu să se apeleze `close(fd[1]);`

Natural, dacă se doreşte ordinea inversă, atunci se vor executa operaţiile `close(fd[1])` în procesul părinte şi `close(fd[0])` în procesul fiu.

3.2.2.2. Exemplu: implementarea who | sort prin pipe şi exec

Să considerăm acum comanda shell compusă:

```
$ who | sort
```

Noi vom prezenta realizarea conexiunii între cele două comenzi: `who | sort` prin *pipe*. Procesul părinte (care înlocuieşte procesul shell) generează doi fii, iar aceştia îşi redirectează corespunzător intrările / ieşirile. Primul dintre ele execută `who`, celălalt `sort`, iar părintele aşteaptă terminarea lor. Sursa este prezentată în programul de mai jos.

```
//whoSort.c
//Lanseaza in pipe comenzile shell: $ who | sort
#include <unistd.h>

main (){

      int p[2];
      pipe (p);
      if (fork () == 0) {         // Primul fiu
         dup2 (p[1], 1);          //redirectarea iesirii standard
         close (p[0]);
         execlp ("who", "who", 0);
      }
      else if (fork () == 0) {    // Al doilea fiu
         dup2 (p[0], 0);          // redirectarea intrarii standard
         close (p[1]);
         execlp ("sort", "sort", 0);//executie sort
      }
      else {                 // Parinte
         close (p[0]);
         close (p[1]);
         wait (0);
         wait (0);
      }
}
```

Observatie: pentru a se înţelege mai bine exemplul de mai sus, invităm cititorul să citească din manualele Unix prezentarea apelului sistem `dup2`. Aici apelul `dup2` are ca paramentru un descriptor *pipe*.

3.2.3. Comunicarea între procese prin FIFO

3.2.3.1. Conceptul de FIFO

Cel mai mare dezavantaj al lui `pipe` sub Unix este faptul că poate fi utilizat numai în procese înrudite: procesele care comunică prin *pipe* trebuie să fie descendenţi din procesul creator al lui *pipe*. Aceasta deoarece întregii descriptori de citire/scriere din/în *pipe* sunt unici şi sunt transmişi proceselor fiu ca urmare a apelului `fork()`.

In jurul anului 1985 (Unix System V), a apărut conceptul *FIFO (pipe cu nume)*. Acesta este un flux de date unidirecţional, accesat prin intermediul unui fişier rezident în sistemul de fişiere. Incepând cu Unix System V, există fişiere de tip *FIFO*. Spre deosebire de *pipe*, fişierele *FIFO* au nume şi ocupă un loc în sistemul de fişiere. Din această cauză, un *FIFO*

poate fi accesat de orice două procese, nu neapărat cu părinte comun. Atenţie însă: chiar dacă un *FIFO* există ca fişier în sistemul de fişiere, pe disc nu se stochează nici o dată care trece prin canalul *FIFO*, acestea fiind stocate şi gestionate în buffer-ele nucleului sistemului de operare!

Conceptual, canalele *pipe* şi *FIFO* sunt similare. Deosebirile esenţiale dintre ele sunt următoarele două:

- suportul pentru *pipe* este o porţiune din memoria RAM gestionată de nucleu, în timp ce *FIFO* are ca suport discul magnetic;
- toate procesele care comunică prin-un *pipe* trebuie să fie descendente ale procesului creator al canalului *pipe*, în timp ce pentru FIFO nu se cere nici o relaţie între procesele protagoniste.

Crearea unui fifo se poate face folosind unul dintre apelurile:

```
int mknod (char *numeFIFO, int mod, 0);
int mkfifo (char *numeFIFO, int mod);
```

sau folosind una dintre comenzile shell:

```
$ mknod numeFIFO  p
$ mkfifo numeFIFO
```

- Prin stringul `numeFIFO` este specificat numele "fişierului" de tip *FIFO*.
- Argumentul `mod`, în cazul apelurilor sistem, reprezintă drepturile de acces la acest fişier. In cazul apelului `mknod`, `mod` trebuie să specifice flagul S_IFIFO, pe lângă drepturile de acces la fişierul *FIFO* (se leagă prin operatorul '|'). Acest flag este definit în `<sys/stat.h>`.
- Pentru crearea unui *FIFO* cu apelul sistem `mknod`, cel de-al treilea parametru este ignorat (trebuie însă specificat, de aceea am pus 0).
- De remarcat că trebuie specificat "`p`" (de la pipe cu nume), ultimul parametru al comenzii shell `mknod`.

Menţionăm că cele două apeluri de mai sus, deşi sunt specificate de POSIX, nu sunt amândouă apeluri sistem pe toate implementările de Unix. Astfel, pe FreeBSD sunt prezente ambele apeluri sistem `mknod()` şi `mkfifo()`, dar pe Linux şi Solaris există numai apelul sistem `mknod()`, funcţia de bibliotecă `mkfifo()` fiind implementată cu ajutorul apelului sistem `mknod()`. Cele două comenzi shell sunt însă disponibile pe majoritatea implementărilor de Unix. Sub sistemele Unix mai vechi, comenzile `mknod` şi `mkfifo` sunt permise numai super-user-ului. Incepând cu Unix System V 4.3 ele sunt disponibile şi utilizatorului obişnuit.

Ştergerea (distrugerea) unui *FIFO* se poate face fie cu comanda shell `rm numeFIFO`, fie cu un apel sistem C `unlink()` care cere un descriptor pentru fişierul *FIFO*.

Odată ce *FIFO* este creat, el trebuie să fie deschis pentru citire sau scriere folosind apelul sistem `open`. Precizarea sau nu a flagului O_NDELAY la apelul sistem `open` are efectele indicate în tabelul următor.

Condiţii	normal	setat O_NDELAY
deschide *FIFO* read-only, dar nu există proces de scriere în *FIFO*	aşteaptă până când apare un proces care deschide *FIFO* pentru scriere	revine imediat fără a semnala eroare
deschide *FIFO* write-only, dar nu există proces de citire din *FIFO*	aşteaptă până apare un proces pentru citire	revine imediat cu semnalarea de eroare: variabila `errno` va deveni `ENXIO`
citire din *FIFO* sau din pipe, dar nu există date de citit	aşteaptă până când apar date în pipe sau *FIFO*, sau până când nu mai există proces deschis pentru scriere. Intoarce numărul de date citite dacă apar noi date, sau 0 dacă nu mai există proces de scriere	revine imediat, cu întoarcerea valorii 0
scrie în *FIFO* sau pipe, dar acesta este plin	aşteaptă până când se face spaţiu disponibil, apoi scrie atâtea date, cât îi permite spaţiul disponibil	revine imediat, cu întoarcerea valorii 0

Aceste reguli trebuie completate cu regulile de citire/scriere de la începutul capitolului despre comunicaţii prin fluxuri de octeţi. De asemenea, înainte de a fi folosit, un canal *FIFO* trebuie să fie în prealabil deschis pentru citire de un proces şi deschis pentru scriere de alt proces.

3.2.3.2. Exemplu: aplicare FIFO la comunicare client / server

Modelul de aplicaţie client / server este clasic în programare. In cele ce urmează vom ilustra o schemă de aplicaţii client / server bazate pe comunicaţii prin *FIFO*. Pentru a se asigura comunicarea bidirecţională se folosesc două *FIFO*-uri. Pentru partea specifică aplicaţiei, se folosesc metodele `client(int in, int out)` şi `server(int in, int out)`. Fiecare dintre ele primeşte ca şi parametri descriptorii de fişiere, presupuse deschise, prin care comunică cu partenerul.

In cele două programe care urmează este schiţată schema serverului şi a clientului. Cele două programe presupun că cele două canale FIFO sunt create, respectiv şterse, prin comenzi Unix.

Programul server:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include <unistd.h>

#include "server.c"

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

main() {
   int    readfd, writefd;
```

```
 - - - - - - - - - - - - - - -

   readfd = open (FIFO1, 0));

   writefd = open (FIFO2, 1));

   for ( ; ; ) {  // bucla de asteptare a cererilor
        server(readfd, writefd);
   }

 - - - - - - - - - - - - - - -

   close (readfd);
   close (writefd);
}
```

Programul client:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <stdio.h>
#include <unistd.h>

#include "client.c"

extern int errno;

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

main() {
   int    readfd, writefd;

 - - - - - - - - - - - - - - -

   writefd = open (FIFO1, 1));

   if ((readfd = open (FIFO2, 0));

   client(readfd, writefd);

 - - - - - - - - - - - - - - -

   close (readfd);
   close (writefd);
}
```

3.3. Interpretoare ale fişierelor de comenzi

3.3.1. Funcţionarea unui interpretor de comenzi shell

Un *interpretor de comenzi* (*shell)* este un program special care furnizează o interfaţă între nucleul sistemului de operare Unix (*kernel*-ul) şi utilizator. Din această perspectivă, a asigurării legăturii între utilizator şi sistem, un *shell* poate fi privit diferit:

1. *limbaj de comandă* care asigură interfaţa dintre calculator şi utilizator. In momentul în care un utilizator îşi deschide o sesiune de lucru, în mod implicit, un *shell* se instalează ca interpretor de comenzi. *Shell*-ul afişează la ieşirea standard (asociată de obicei unui terminal) un prompter, invitând astfel utilizatorul să introducă comenzi sau să lanseze în execuţie fişiere de comenzi, eventual parametrizate.

2. *limbaj de programare*, ce are ca element de bază (element primitiv) *comanda* Unix (similară semantic cu *instrucţiunea de atribuire* din limbajele de programare). Ca şi element primitiv de dirijare a succesiunii elementelor de bază este *valoarea codului de retur* al ultimei comenzi executate: valoarea 0 înseamnă *true*, valoare nenulă înseamnă *false* (corespondentul din limbajele de programare clasice este *condiţia*). *Shell*-urile dispun de conceptele de variabilă, constantă, expresie, structuri de control şi subprogram. Spre deosebire de alte limbaje de programare, expresiile cu care lucrează *shell*-urile sunt preponderent şiruri de caractere. In ceea ce priveşte cerinţele sintactice, acestea au fost reduse la minim prin eliminarea parantezelor de delimitare a parametrilor, a diferitelor caractere de separare şi terminare, a declaraţiilor de variabile, etc.

un shell lansat în execuţie la deschiderea unei sesiuni de lucru va rămâne activ până la închiderea respectivei sesiuni. Odată instalat, acesta lucrează conform algoritmului următor:

```
CâtTimp (nu s-a închis sesiunea)
   Afişează prompter;
   Citeşte linia de comandă;
   Dacă ( linia se termină cu '&' ) atunci
     Crează un proces şi-i dă spre execuţie comanda
     Nu aşteaptă ca execuţia să se termine
   Altfel
     Crează un proces şi-i dă spre execuţie comanda
     Aşteaptă să se termine execuţia comenzii
   SfDacă
SfCâtTimp
```

Este important să remarcăm, din algoritmul de mai sus, cele două moduri în care o comandă poate fi executată:

- modul *foreground* - execuţie la vedere. In acest gen de execuţie *sh* lansează execuţia comenzii, aşteaptă terminarea ei după care afişează din nou prompterul pentru o nouă comandă. Acesta este modul implicit de execuţie al oricărei comenzi Unix.
- modul *background* - execuţie în fundal, ascunsă. In acest gen de execuţie ***sh*** lansează procesul care va executa comanda, dar nu mai aşteaptă terminarea ei ci afişează imediat prompterul, oferind utilizatorului posibilitatea de a lansa imediat o nouă

comandă. Comada care se doreşte a fi lansată în background trebuie să se încheie cu caracterul special '**&**'.

Intr-o fereastră (sesiune) de lucru Unix se pot rula oricâte comenzi în background şi numai una în foreground. Iată, spre exemplu, trei astfel de comenzi, două lansate în background - o copiere de fişier (comanda `cp`) şi o compilare (comanda `gcc`) şi una în foreground - editare de fişier (comanda `vi`):

```
cp A B &
gcc x.c &
vi H
```

3.3.2. Programarea în shell

3.3.2.1. Scurtă prezentare a limbajului sh

In cele ce urmează vom prezenta gramatica limbajului sh – cel mai simplu shell de sub Unix. Vom pune în evidenţă principalele categorii sintactice, semantica / funcţionalitatea fiecărei astfel de categorii se deduce uşor din context.

Vom considera următoarele convenţii, pe care le folosim doar în scrierea regulilor gramaticii:

- O categorie gramaticală se poate defini prin una sau mai multe alternative de definire. Alternativele se scriu câte una pe linie, începând cu linia de după numele categoriei gramaticale, astfel:

  categorieGramaticală:
  alternativa_1 de definire
  ----
  alternativa_n de definire
- [ ]? Semnifică faptul că, construcţia dintre paranteze va apărea cel mult odată.
- [ ]+ Semnifică faptul că, construcţia dintre paranteze va apărea cel puţin odată.
- [ ]* Semnifică faptul că, construcţia dintre paranteze poate să apară de 0 sau mai multe ori.

Folosind aceste convenţii, sintaxa limbajului **`sh`** (în partea ei superioară, fără detalii) este descrisă în fig. 2.2.

Semnificaţia unora dintre elementele sintactice din fig. 2.2 este:

- *cuvânt:* secvenţă de caractere diferite de caracterele albe (spaţiu, tab)
- *nume:* secvenţă ce începe cu literă şi continuă cu litere, cifre, _ (underscore)
- *cifra:* cele 10 cifre zecimale

O *comanda* `sh` poate avea oricare dintre cele 9 forme prezentate. Una dintre modalităţile de definire este cea de *comandăElementară*, unde o astfel de comandă elementară este un şir de *elemente*, un element putând fi definit în 10 moduri distincte. O *legarePipe* este fie o singură comandă, fie un şir de comenzi separate prin caracterul special '**|**'. In sfârşit, *listaCom* este o succesiune de *legarePipe* separate şi eventual terminate cu simboluri speciale.

Se poate observa că, în conformitate cu gramatica de mai sus, **sh** acceptă şi construcţii fără semantică! De exemplu, ***comandă*** poate fi o ***comandăElementară***, care să conţină un singur ***element***, format din ***>&-;***. O astfel de linie este acceptată de **sh**, fiind corectă din punct de vedere sintactic, deşi nu are sens din punct de vedere semantic.

Shell-ul **sh** are un număr de 13 **cuvinte rezervate**. Lista acestora este următoarea:

```
if then else elif fi
case in esac
for while until do done
```

Structurile alternative **if** şi **case** sunt închise de construcţiile **fi**, respectiv **esac**, obţinute prin oglindirea cuvintelor de start. In cazul ciclurilor repetitive, sfârşitul acestora este indicat prin folosirea cuvântului rezervat **done**. Nu s-a folosit construcţia similară corespunzătoare lui **do**, deoarece **od** este numele unui comenzi clasice Unix.

Incheiem acest subcapitol cu prezentarea sintaxei unor **construcţii rezervate**, precum şi a unor **caractere cu semnificaţie specială** în shell-ul **sh**.

a) Construcţii sintactice:

|	legare pipe
&&	legare andTrue
||	legare orFalse
;	separator / terminator comandă
;;	delimitator case
(), {}	grupări de comenzi
< <<	redirectări intrare
> >>	redirectări ieşire
&*cifra*, **&-**	specifică intrare sau ieşire standard

b) Machete şi specificări generice:

*****	înlocuieşte orice şir de caractere
?	înlocuieşte orice caracter
[...]	înlocuieşte cu orice caracter din **...**

Observaţie: aceste machete şi specificări generice nu trebuie confundate cu convenţia propusă la începutul subcapitolului pentru scrierea gramaticii limbajului **sh**

comandă:
comandăElementară
(*listaCom* **)**
{ *listaCom* **}**
if *listaCom* **then** *listaCom* [**elif** *listaCom* **then** *listaCom*]* [**else** *listaCom*]? **fi**
case *cuvant* **in** [*cuvant* [| *cuvant*]* **)** *listaCom* **;;**]+ **esac**
for *nume* **do** *listaCom* **done**
for *nume* **in** [*cuvant*]+ **do** *listaCom* **done**
while *listaCom* **do** *listaCom* **done**
until *listaCom* **do** *listaCom* **done**

comandăElementară:
[*element*]+

listaCom:
legarePipe [*separator legarePipe*]* [*terminator*]?

legarePipe:
comanda [| *comanda*]*

element:
cuvânt
nume=cuvânt
>cuvânt
<cuvânt
>>cuvânt
<<cuvânt
>&cifra
<&cifra
<&-
>&-

separator:
&&
||
terminator

terminator:
;
&

3.4. Probleme propuse

I.

a. Descrieţi pe scurt funcţionarea apelului sistem fork şi valorile pe care le poate returna.

b. Ce tipăreşte pe ecran secvenţa de program de mai jos, considerând că apelul sistem fork se execută cu succes? Justificaţi răspunsul.

```
int main() {
    int n = 1;
    if(fork() == 0) {
        n = n + 1;
        exit(0);
    }
    n = n + 2;
    printf("%d: %d\n", getpid(), n);
    wait(0);
    return 0;
}
```

c. Ce tipăreşte pe ecran fragmentul de script shell de mai jos? Explicaţi funcţionarea primelor trei linii ale fragmentului.

```
1  for F in *.txt; do
2      K=`grep abc $F`
```

```
3       if [ "$K" != "" ]; then
4           echo $F
5       fi
6   done
```

II.

a. Se dă fragmentul de cod de mai jos. Indicaţi liniile care se vor tipări pe ecran în ordinea în care vor apărea, considerând că apelul sistem fork se execută cu succes? Justificaţi răspunsul.

```
int main() {
    int i;
    for(i=0; i<2; i++) {
        printf("%d: %d\n", getpid(), i);
        if(fork() == 0) {
            printf("%d: %d\n", getpid(), i);
            exit(0);
        }
    }
    for(i=0; i<2; i++) {
        wait(0);
    }

    return 0;
}
```

b. Explicaţi funcţionarea fragmentului de script shell de mai jos. Ce se întâmplă, dacă fişierul *raport.txt* lipseşte iniţial. Adăugaţi rândul de cod care lipseşte pentru generarea fişierului *raport.txt*.

```
more raport.txt
rm raport.txt
for f in *.sh; do
  if [ ! -x $f ]; then
     chmod 700 $f
  fi
done
mail -s "Raport fisiere afectate" admin@scs.ubbcluj.ro <raport.txt
```

4. Bibliografie generală

1. ***: Linux man magyarul, http://people.inf.elte.hu/csa/MAN/HTML/index.htm
2. A.S. Tanenbaum, A.S. Woodhull, *Operációs rendszerek*, 2007, Panem Kiadó.
3. Alexandrescu, *Programarea modernă in C++*. Programare generică si modele de proiectare aplicate, Editura Teora, 2002.
4. Angster Erzsébet: *Objektumorientált tervezés és programozás Java*, 4KÖR Bt, 2003.
5. Bartók Nagy János, Laufer Judit, *UNIX felhasználói ismeretek*, Openinfo
6. Bjarne Stroustrup: *A C++ programozási nyelv*, Kiskapu kiadó, Budapest, 2001.
7. Bjarne Stroustrup: *The C++ Programming Language Special Edition*, AT&T, 2000.
8. Boian F.M. Frentiu M., Lazăr I. Tambulea L. *Informatica de bază*. Presa Universitară Clujeana, Cluj, 2005
9. Boian F.M., Ferdean C.M., Boian R.F., Dragoş R.C., *Programare concurentă pe platforme Unix, Windows, Java*, Ed. Albastră, Cluj-Napoca, 2002
10. Boian F.M., Vancea A., Bufnea D., Boian R.,F., Cobârzan C., Sterca A., Cojocar D., *Sisteme de operare*, RISOPRINT, 2006
11. Bradley L. Jones: *C# mesteri szinten 21 nap alatt*, Kiskapu kiadó, Budapest, 2004.
12. Bradley L. Jones: *SAMS Teach Yourself the C# Language in 21 Days*, Pearson Education,2004.
13. Cormen, T., Leiserson, C., Rivest, R., *Introducere în algoritmi*, Editura Computer Libris Agora, Cluj, 2000
14. DATE, C.J., An Introduction to Database Systems (8th Edition), Addison-Wesley, 2004.
15. Eckel B., *Thinking in C++*, vol I-II, http://www.mindview.net
16. Ellis M.A., Stroustrup B., *The annotated C++ Reference Manual*, Addison-Wesley, 1995
17. Frentiu M., Lazăr I. *Bazele programării*. Partea I-a: Proiectarea algoritmilor
18. Herbert Schildt: Java. The Complete Reference, Eighth Edition, McGraw-Hill, 2011.
19. Horowitz, E., *Fundamentals of Data Structures in C++*, Computer Science Press, 1995
20. J. D. Ullman, J. Widom: *Adatbázisrendszerek - Alapvetés*, Panem kiado, 2008.
21. ULLMAN, J., WIDOM, J., A First Course in Database Systems (3rd Edition), Addison-Wesley + Prentice-Hall, 2011.
22. Kiadó Kft, 1998, http://www.szabilinux.hu/ufi/main.htm
23. Niculescu,V., Czibula, G., *Structuri fundamentale de date şi algoritmi. O perspectivă orientată obiect.*, Ed. Casa Cărţii de Stiinţă, Cluj-Napoca, 2011
24. RAMAKRISHNAN, R., Database Management Systems. McGraw-Hill, 2007, http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed.html
25. Robert Sedgewick: *Algorithms*, Addison-Wesley, 1984
26. Simon Károly: *Kenyerünk Java. A Java programozás alapjai*, Presa Universitară Clujeană, 2010.
27. Tâmbulea L., Baze de date, Facultatea de matematică şi Informatică, Centrul de Formare Continuă şi Invăţământ la Distanţă, Cluj-Napoca, 2003
28. V. Varga: *Adatbázisrendszerek (A relációs modelltől az XML adatokig)*, Editura Presa Universitară Clujeană, 2005, p. 260. ISBN 973-610-372-2
29. OMG. *UML Superstructure*, 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/
30. Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd Edition). Addison-Wesley Professional, 2003.

31. OMG. *MDA Guide Version 1.0.1*, 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf