

Manual de Informatică pentru licență iulie și septembrie 2018

Specializarea Informatică

Tematica generală:

Partea 1. Algoritmă și programare

1. Căutari (secvențială și binară), interclasare, sortări (selecție, bubblesort, inserție, mergesort, quicksort). Metoda backtracking.
2. Concepte OOP în limbaje de programare (Python, C++, Java, C#): Clase și obiecte. Membrii unei clase și specificatorii de acces. Constructori și destructori.
3. Relații între clase. Clase derivate și relația de moștenire. Suprascrierea metodelor. Polimorfism. Legare dinamica. Clase abstracte și interfețe.
4. Diagrame de clase și de interacțiune între obiecte în UML: Pachete, clase și interfețe. Relații între clase și interfețe. Obiecte. Mesaje
5. Liste. Dicționare. Specificarea operațiilor caracteristice (fără implementări)
6. Identificarea structurilor și 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 relaționale. Primele trei forme normale ale unei relații.
2. Interogarea bazelor de date cu operatori din algebra relațională.
3. Interogarea bazelor de date relaționale cu SQL (Select).

Partea 3. Sisteme de operare

1. Structura sistemelor de fișiere Unix.
2. Procese Unix: creare, funcțiile fork, exec, exit, wait; comunicare prin pipe și FIFO.
3. Programare shell Unix. 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 ȘI PROGRAMARE	3
1.1. CĂUTĂRI ȘI SORTĂRI	3
1.1.1. Căutări	3
1.1.2. Interclasare	5
1.1.3. Sortări	6
1.1.4. Metoda backtracking	11
1.2. CONCEPTE OOP ÎN LIMBAJE DE PROGRAMARE	15
1.2.1. Noțiunea de clasă	15
1.3. RELAȚII ÎNTRE CLASE	24
1.3.1. Bazele teoretice	24
1.3.2. Declararea claselor derivate	24
1.3.3. Funcții virtuale	26
1.3.4. Clase abstracte	30
1.3.5. Interfețe	32
1.4. DIAGrame DE CLASE ȘI INTERACȚIUNI ÎNTRE OBIECTE ÎN UML: PACHETE, CLASE ȘI INTERFEȚE. RELAȚII ÎNTRE CLASE ȘI INTERFEȚE. OBIECTE. MESAJE	34
1.4.1. Diagrame de clase	37
1.4.2. Diagrame de interacțiune	44
1.5. LISTE ȘI DICȚIONARE	46
1.5.1. Liste	47
1.5.2. Dicționare	51
1.6. PROBLEME PROPUSE	53
2. BAZE DE DATE	55
2.1. BAZE DE DATE RELAȚIONALE. PRIMELE TREI FORME NORMALE ALE UNEI RELAȚII ...	55
2.1.1. Modelul relațional	55
2.1.2. Primele trei forme normale ale unei relații	58
2.2. INTEROGAREA BD CU OPERATORI DIN ALGEBRA RELAȚIONALĂ	65
2.3. INTEROGAREA BAZELOR DE DATE RELAȚIONALE CU SQL	68
2.4. PROBLEME PROPUSE	74
3. SISTEME DE OPERARE	77
3.1. STRUCTURA SISTEMELOR DE FIȘIERE UNIX	77
3.1.1. Structura Internă a Discului UNIX	77
3.1.2. Tipuri de fișiere și sisteme de fișiere	80
3.2. PROCESE UNIX	84
3.2.1. Principalele apeluri system de gestiune a proceselor	84
3.2.2. Comunicarea între procese prin pipe	88
3.2.3. Comunicarea între procese prin FIFO	90
3.3. INTERPRETOARE ALE FIȘIERELOR DE COMENZI	94
3.3.1. Funcționarea unui interpretor de comenzi shell	94
3.3.2. Programarea în shell	95
3.4. PROBLEME PROPUSE	97
4. BIBLIOGRAFIE GENERALĂ	99

1. Algoritmă și programare

1.1. Căutări și sortări

1.1.1. Căutări

Datele se află în memoria internă, într-un șir 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 poziția elementului căutat (dacă acesta există).

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

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

$$k_1 < k_2 < \dots < k_n.$$

Uneori este util să aflăm nu numai dacă există un articol cu cheia dorită ci și 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 $a, n, (k_i, i=1, n)$;

Precondiția: $n \in \mathbb{N}, n \geq 1$ și $k_1 < k_2 < \dots < k_n$;

Rezultate p ;

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

1.1.1.1. Căutare secvențială

O primă metodă este căutarea **secvențială**, 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: $\{n \in \mathbb{N}, n \geq 1$ și $k_1 < k_2 < \dots < k_n\}$
 $\{$ Se caută p astfel ca: $(p=1$ și $a \leq k_1)$ sau
 $\{ (p=n+1$ și $a > k_n)$ sau $(1 < p \leq n)$ și $(k_{p-1} < a \leq k_p)$.
 $\{$ Cazul "încă negăsit" $\}$

```
Fie p := 0;
Dacă a ≤ k1 atunci p := 1 altfel
  Dacă a > kn atunci p := n + 1 altfel
    Pentru i := 2; n execută
      Dacă (p = 0) și (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 $n-1$ comparații, întrucât contorul i va lua toate valorile de la 2 la n . Cele n chei împart axa reală în $n+1$ intervale. Tot atâtea comparații se vor efectua în $n-1$ din cele $n+1$ intervale în care se poate afla cheia căutată, deci complexitatea medie are același ordin de mărime ca și complexitatea în cel mai rău caz.

Evident că în multe situații 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 și k1 < k2 < ... < kn}
                                              {Se caută p astfel ca: p=1 și a ≤ k1} sau }
                                              {(p=n+1 și a > kn) sau (1 < p ≤ n) și (kp-1 < a ≤ kp)}

  Fie p:=1;
  Dacă a > k1 atunci
    Cât timp p ≤ n și a > kp execută p:=p+1 sfcât
  sfdacă
sf-CautSucc

```

În cel mai rău caz și acest algoritm face același număr de operații ca și subalgoritmul *Cautsecv*. În medie numărul operațiilor este jumătate din numărul mediu de operații efectuate de subalgoritmul *Cautsecv* deci complexitatea este aceeași.

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 relație se află cheia articolului aflat în mijlocul colecției cu cheia de căutare. În urma acestei verificări căutarea se continuă doar într-o jumătate a colecției. În acest mod, prin înjumătățiri succesive se micșorează volumul colecției rămase pentru căutare. Căutarea binară se poate realiza practic prin apelul funcției `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 și k1 < k2 < ... < kn}
                                              {Se caută p astfel ca: (p=1 și a ≤ k1) sau }
                                              {(p=n+1 și a > kn) sau (1 < p ≤ n) și (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

```

Funcția CăutareBinară(a, n, K, St, Dr) este:

```
Dacă  $St \geq Dr - 1$ 
    atunci CăutareBinară := Dr
    altfel  $m := (St+Dr) \text{ Div } 2$ ;
        Dacă  $a \leq k_m$ 
            atunci CăutareBinară := CăutareBinară( $a, n, K, St, m$ )
            altfel CăutareBinară:= CăutareBinară( $a, n, K, m, Dr$ )
        sfdacă
    sfdacă
sf-CăutareBinară
```

În funcția CăutareBinară descrisă mai sus, variabilele St și Dr reprezintă capetele intervalului de căutare, iar m reprezintă mijlocul acestui interval. Prin această metodă, într-o colecție având n elemente, rezultatul căutării se poate furniza după cel mult $\log_2 n$ comparații. Deci complexitatea în cel mai rău caz este direct proporțională cu $\log_2 n$. Fără a insista asupra demonstrației, menționăm că ordinul de mărime al complexității medii este același.

Se observă că funcția CăutareBinarăNerec se apelează recursiv. Se poate înlătura ușor recursivitatea, așa cum se poate vedea în următoarea funcție:

```
Funcția CăutareBinarăNerec( $a, n, K, St, Dr$ ) este:
    Cât timp  $Dr - St > 1$  execută
         $m := (St+Dr) \text{ Div } 2$ ;
        Dacă  $a \leq k_m$  atunci  $Dr := m$  altfel  $St := m$  sfdacă
    sfcât
    CăutareBinarăNerec := Dr
sf- CăutareBinarăNerec
```

1.1.2. Interclasare

Fiind date două secvențe de date, ordonate crescător (sau descrescător) după o cheie, se cere să se obțină o colecție care să fie de asemenea ordonată crescător (respectiv descrescător) după aceeași cheie și care să fie formată din elementele secvențelor date. Acest lucru se poate obține direct (fără o sortare a secvenței finale) prin parcurgerea secvențială a celor două secvențe, simultan cu generarea secvenței cerute. Prin compararea a două elemente din secvențele de intrare se va decide care element va fi adăugat în secvența de ieșire.

Menționăm că sunt două tipuri de interclasări: (1) *interclasare cu păstrarea dublurilor* (în acest caz se păstrează în colecția rezultat toate elementele din cele două secvențe inițiale, chiar dacă elementele se repetă) și (2) *interclasare fără păstrarea dublurilor* (în acest caz se păstrează în secvența rezultat toate elementele distincte din cele două secvențe).

În continuare vom prezenta *interclasare cu păstrarea dublurilor*. Lăsăm ca exercițiu varianta de *interclasare fără păstrarea dublurilor*

Deci ne interesează un algoritm de rezolvare a problemei ce are următoarea specificare:

Date $m, (x_i, i=1,m), n, (y_i, i=1,n)$;

Precondiția: $\{x_1 \leq x_2 \leq \dots \leq x_m\}$ și $\{y_1 \leq y_2 \leq \dots \leq y_n\}$

Rezultate $k, (z_i, i=1,k)$;

Postcondiția: $\{k=m+n\}$ și $\{z_1 \leq z_2 \leq \dots \leq z_k\}$ și (z_1, z_2, \dots, z_k) este o permutare a valorilor $(x_1, \dots, x_m, y_1, \dots, y_n)$

O soluție posibilă ar fi depunerea componentelor vectorului X și a componentelor vectorului Y în vectorul Z , realizând astfel a doua parte din postcondiție. Ordonând apoi componentele vectorului Z obținem soluția dorită. Acest algoritm, deși corect, este ineficient. Este important ca la o singură trecere prin vectorii X și Y să se obțină vectorul Z . Acest lucru este realizat de următorul algoritm de interclasare:

```
Subalgoritmul Interclasare(m, X, n, Y, k, Z) este:      {X are cele m componente}
                                                    {ordonate crescător. La fel Y cu n componente}
                                                    {Cele m+n valori se depun în Z, tot ordonate crescător}
Fie i:=1; j:=1; k:=0;
Câttimp (i<=m) și (j<=n) execută                      {Există componente}
  Dacă  $x_i \leq y_j$ 
    atunci Cheamă ADAUGĂ( $x_i, k, Z$ )                    {și în X}
        i:=i+1
    altfel Cheamă ADAUGĂ( $y_j, k, Z$ )                    {și în Y}
        j:=j+1
  sfdacă
sfcât
Câttimp (i<=m) execută                                  {Există componente}
  Cheamă ADAUGĂ( $x_i, k, Z$ )                              {numai în X}
  i:=i+1                                                {avansează în vectorul X}
sfcât
Câttimp (j<=n) execută                                  {Există componente}
  Cheamă ADAUGĂ( $y_j, k, Z$ )                              {numai în Y}
  j:=j+1                                                {avansează în vectorul Y}
sfcât
sf-Interclasare
```

Aici s-a folosit subalgoritmul ADAUGĂ(val, k, Z) care adaugă valoarea val la sfârșitul vectorului Z având k elemente, subalgoritm dat în continuare.

```
Subalgoritmul ADAUGĂ(val, k, Z) este:                  {Aduagă val}
  k:=k+1;                                              {la finalul vectorul Z cu}
  z_k:=val;                                           {k componente}
sf-ADAUGĂ
```

Complexitatea subalgoritmului *Interclasare* descris anterior este $\theta(m+n)$. Spațiul suplimentar de memorare necesar pentru subalgoritmul de interclasare este $\theta(1)$.

1.1.3. Sortări

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

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

Date n, K ;

$\{K=(k_1, k_2, \dots, k_n)\}$

Precondiția: $k_i \in \mathbb{R}, i=1, n$

Rezultate K' ;

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

Deci $k_1 \leq k_2 \leq \dots \leq k_n$.

1.1.3.1. Sortare prin selecție

O primă tehnică numită "*Selecție*" se bazează pe următoarea idee: se determină poziția elementului cu cheie de valoare minimă (respectiv maximă), după care acesta se va interschimba cu primul element. Acest procedeu se repetă pentru subcolecția 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 $k_1 \leq k_2 \leq \dots \leq k_n$ }

```
Pentru i := 1; n-1 execută
  Fie ind := i;
  Pentru j := i + 1; n execută
    Dacă  $k_j < k_{ind}$  atunci ind := j sfdacă
  sfpentru
  Dacă  $i < ind$  atunci t :=  $k_i$ ;  $k_i := k_{ind}$ ;  $k_{ind} := t$  sfdacă
sfpentru
sf-Selectie
```

Se observă că numărul de comparații este:

$$(n-1)+(n-2)+\dots+2+1=n(n-1)/2$$

indiferent de natura datelor. Deci complexitatea medie, dar și în cel mai rău caz este $O(n^2)$.

1.1.3.2. Bubble sort

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

Subalgoritmul *BubbleSort*(n, K) este:

```
Repetă
  Fie kod := 0; {Ipoteza "este ordine"}
  Pentru i := 2; n execută
    Dacă  $k_{i-1} > k_i$  atunci
      t :=  $k_{i-1}$ ;
       $k_{i-1} := k_i$ ;
```

```

        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 $(n-1)+(n-2)+ \dots +2+1 = n(n-1)/2$ comparații, deci complexitatea lui este $O(n^2)$.

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 ki-1 > ki atunci
            t := ki-1;
            ki-1 := ki;
            ki := t;
            kod := 1                                {N-a fost ordine!}
        sfdacă
    sfpentru
    s := s + 1
    pânăcând kod = 0 sfrep                        {Ordonare}
sf-BubbleSort

```

1.1.3.3. Sortarea prin inserție

Ideea de bază a acestei metode de sortare este că, în timpul parcurgerii elementelor, inserăm elementul curent la poziția corectă în secvența de elemente deja sortată. Astfel, secvența conținând elementele deja prelucrate este păstrată sortată, iar la finalul parcurgerii, întreaga secvență va fi sortată. Acest algoritm se numește *Sortare prin inserție (InsertionSort)*.

Subalgoritm SortareInserție(n, K) este:

```

Pentru i:=2; n execută
    Fie ind:=i-1; a:=ki;
    Cât timp ind>0 și a<kind execută
        kind+1 := kind ;
        ind:=ind-1
    sfcât
    kind+1:=a
    sfpentru
sf-SortareInserție

```

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

1.1.3.4. Sortare prin interclasare (Merge Sort)

Vom folosi în cele ce urmează o variantă ușor modificată a subalgoritmului *Interclasare*, (descriș în Secțiunea 1.1.2) pentru a interclasa subsecvențe. Această versiune va interclasa $X[sx, \dots, dx]$ și $Y[sy, \dots, dy]$ în $Z[1, \dots, k]$.

```
Subalgoritm InterclasareSubSecv (sx, dx, X, sy, dy, Y, k, Z) este:
{X are componentele sx, ..., dx ordonate crescător. Y are sy, ..., dy ordonate}
{crescător. Toate aceste valori se adaugă în Z, ordonat crescător.}
  Fie i:=sx; j:=sy; k:=0;
  Cât timp (i<=dx) și (j<=dy) execută {Sunt componente de procesat}
    Dacă  $x_i \leq y_j$ 
      atunci Cheamă ADAUGĂ( $x_i, k, Z$ ) {în X}
      i:=i+1
    altfel Cheamă ADAUGĂ( $y_j, k, Z$ ) {în Y}
    j:=j+1
  sfdacă
sfcât
Cât timp (i<=dx) execută {Sunt componente}
  Cheamă ADAUGĂ( $x_i, k, Z$ ) {doar în X}
  i:=i+1
sfcât
Cât timp (j<=dy) execută {Sunt componente}
  Cheamă ADAUGĂ( $y_j, k, Z$ ) {doar în Y}
  j:=j+1
sfcât
sf-InterclasareSubSecv
```

Algoritmul *Sortare-Interclasare* pentru sortarea unei secvențe S cu n elemente se folosește strategia *divide-et-impera*:

1. Dacă S are cel puțin două elemente, fie $S1$ and $S2$ subsecvențele din S , fiecare conținând aproximativ jumătatea numărului de elemente din S ($S1$ conține primele $\frac{n}{2}$ elemente și $S2$ conține restul elementelor).
2. Sortează secvențele $S1$ și $S2$ folosind *Sortare-Interclasare*.
3. Înlouiește elementele din S cu rezultatul interclasării secvențelor sortate $S1$ și $S2$.
Subalgoritmul *SortareInterclasare* este descriș mai jos.

```
Subalgoritm SortareInterclasare (n, A) este:
  Cheamă SortareInterclasareRec (1, n, A);
sf-SortareInterclasare
```

```
Subalgoritm SortareInterclasareRec (Stânga, Dreapta, A) este:
{Sortare prin interclasare a secvenței  $A_{Stânga}, A_{Stânga+1}, \dots, A_{Dreapta}$ }
  Dacă Stânga < Dreapta
    atunci
      Fie  $m := (Stânga + Dreapta) \div 2$ ;
      Cheamă SortareInterclasareRec (Stânga, m, A);
      Cheamă SortareInterclasareRec (m+1, Dreapta, A);
      Cheamă InterclasareSubSecv (Stânga, m, A, m+1, Dreapta, A, k, C);
      Fie  $A[Stânga \dots Dreapta] = C[1 \dots k]$ ;
```

sfdacă
sf-SortareInterclasareRec

Subalgoritmul de mai sus sortează recursiv cele două părți ale unei secvențe și apoi le interclasează într-un vector extern C . La final, vectorul C este re-copiat în subsecvența A . Complexitatea timp pentru *SortareInterclasare* este $\theta(n \cdot \log_2 n)$.

Se observă faptul că subalgoritmul *SortareInterclasare* folosește spațiu suplimentare de memorare $\theta(n)$ necesar pentru interclasarea celor două subșiruri.

1.1.3.5. Quicksort

O metodă mai performantă de ordonare, care va fi prezentată în continuare, se numește "*QuickSort*" și 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 subșir precizat prin limita inferioară și limita superioară a indicilor acestuia. Apelul procedurii pentru ordonarea întregului șir este: $\text{QuickSort}(n, K, 1, n)$, unde n reprezintă numărul de articole ale colecției date. Deci

Subalgoritmul *SortareRapidă*(n, K) este:

Cheamă $\text{QuickSort}(n, K, 1, n)$
sf-SortareRapidă

Procedura $\text{QuickSort}(n, K, St, Dr)$ va realiza ordonarea subșirului $k_{St}, k_{St+1}, \dots, k_{Dr}$. Acest subșir va fi rearanjat astfel încât k_{St} să ocupe poziția lui finală (când șirul este ordonat). Dacă i este această poziție, șirul va fi rearanjat astfel încât următoarea condiție să fie îndeplinită:

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

Odată realizat acest lucru, în continuare va trebui doar să ordonăm subșirul $k_{St}, k_{St+1}, \dots, k_{i-1}$ prin apelul recursiv al procedurii $\text{QuickSort}(n, K, St, i-1)$ și apoi subșirul k_{i+1}, \dots, k_{Dr} prin apelul $\text{QuickSort}(n, K, i+1, Dr)$. Desigur ordonarea acestor două subșiruri (prin apelul recursiv al procedurii) mai este necesară doar dacă acestea conțin cel puțin două elemente.

Procedura QuickSort este prezentată în continuare :

Subalgoritmul $\text{QuickSort}(n, K, St, Dr)$ este:

Fie $i := St; j := Dr; a := k_i$;
Repetă
 Cât timp $k_j \geq a$ și $(i < j)$ execută $j := j - 1$ sfcât
 $k_i := k_j$;

```

    Cât timp  $k_i \leq a$  și  $(i < j)$  execută  $i := i + 1$  sf-cât
     $k_j := k_i$ ;
    până-când  $i = j$  sf-rep
    Fie  $k_i := a$ ;
    Dacă  $St < i-1$  atunci Cheamă QuickSort( $n, K, St, i - 1$ ) sf-dacă
    Dacă  $i+1 < Dr$  atunci Cheamă QuickSort( $n, K, i + 1, Dr$ ) sf-dacă
sf-QuickSort

```

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

1.1.4. Metoda backtracking

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

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

Problema 1. (Generarea permutărilor) Fie n un număr natural. Determinați permutările numerelor $1, 2, \dots, n$.

O soluție pentru generarea permutărilor, în cazul particular $n = 3$, ar putea fi:

```

Subalgoritmul Permutărilor este:
Pentru  $i_1 := 1; 3$  execută
    Pentru  $i_2 := 1; 3$  execută
        Pentru  $i_3 := 1; 3$  execută
            Fie posibil :=  $(i_1, i_2, i_3)$ 
            Dacă componentele vectorului posibil sunt distincte
                atunci
                    Tipărește posibil
            sf-dacă
        sf-pentru
    sf-pentru
sf-Permutărilor

```

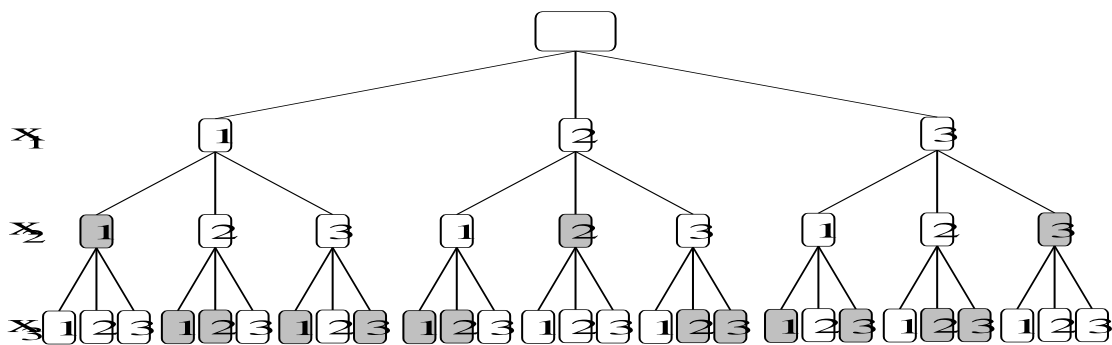


Figura 1.1 Reprezentare grafică a produsului cartezian $\{1, 2, 3\}^3$

Observații privind subalgoritmul *Permutări 1*:

- Pentru n oarecare nu putem descrie un algoritm care să conțină n cicluri în textul sursă.
- Numărul total de vectori verificați este 3^3 , iar în general n^n . Vectorii *posibil* verificați sunt reprezentați 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 îmbunătățire a acestor algoritmi ar consta în a verifica anumite condiții 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 promițător* (pentru a fi o permutare), în schimb vectorul (1, 1, ...) nu este. Vectorul (1, 3, ...) satisface anumite *condiții de continuare* (pentru a ajunge la soluție) - are componente distincte. Nodurile hașurate din Figura 1.1 constituie valori care nu conduc la o soluție.

Vom descrie un algoritm general pentru metoda Backtracking după care vom particulariza acest algoritm pentru problemele enunțate la începutul secțiunii. Pentru început vom face câteva observații și notații privind metoda Backtracking aplicată unei probleme în care soluțiile se reprezintă pe vectori, nu neapărat de lungime fixă:

- spațiul de căutare a soluțiilor (spațiul soluțiilor posibile): $S = S_1 \times S_2 \times \dots \times S_n$;
- *posibil* este vectorul pe care se reprezintă soluțiile;
- $posibil[1..k] \in S_1 \times S_2 \times \dots \times S_k$ este un vector care poate conduce sau nu la o soluție; k reprezintă indice pentru vectorul *posibil*, respectiv nivel în arborele care redă grafic procesul de căutare (Figura 1.2).
- $posibil[1..k]$ este promițător dacă satisface condiții care pot conduce la o soluție;
- *soluție*($n, k, posibil$) funcție care verifică dacă vectorul (promițător) $posibil[1..k]$ este soluție a problemei.

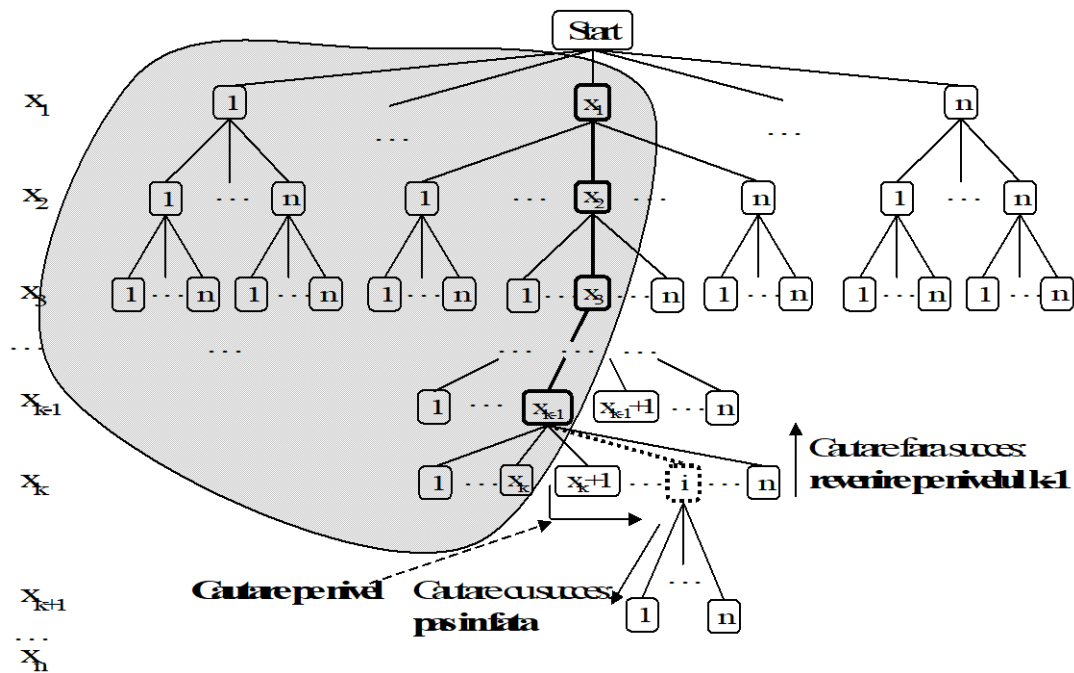


Figura 1.2. Spațiul soluțiilor posibile pentru generarea permutărilor

Procesul de căutare poate fi urmărit în algoritmul care urmează:

```

Algoritmul Backtracking este: {varianta nefinisată}
Fie k := 1
@Inițializează căutarea pe nivelul k (= 1)
Cât timp k > 0 execută {posibil[1..k-1] este promițător}
  @Caută (secvențial) pe nivelul k o valoare v, pentru a completa în
  continuare vectorul posibil[1..k-1] astfel încât posibil[1..k] să
  fie promițător
  Dacă căutarea este cu succes
    atunci Fie posibil[k] := v {posibil[1..k] este promițător}
    Dacă soluție(n, k, posibil)
      atunci {o soluție! (rămânem pe nivelul k)}
      Tiparește posibil[1..k]
    altfel {e doar un vector promițător}
    @Initializeaza cautarea pe nivelul k+1
    Fie k := k + 1 {pas în față (pe nivelul k+1)}
  sfdacă
  altfel {pas în spate (revenire pe nivelul k-1)}
  k := k - 1
sfdacă
sfcat
sf-Backtracking
  
```

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

condiții-continuare(k, posibil, v)

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

Apoi, pentru a inițializa căutarea la nivelul *j* avem nevoie de a alege un element fictiv din mulțimea S_j , activitate realizată de funcția

```
init(j)
```

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

```
urmator(j, v, nou)
```

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

```
Algoritmul Backtracking este: {versiune finală}
Fie k := 1;
posibil[1] := init(1);
Cât timp k > 0 execută {posibil[1..k-1] este promițător}
    Fie Găsit := false; v := posibil[k];
    Cât timp Urmator(k, v,urm) și not Găsit execută
        Fie v := urm;
        Dacă condiții-continuare(k, posibil, v) atunci
            Găsit := true
        sfdacă
    sfcât
    Dacă Găsit
        atunci Fie posibil[k] := v; {posibil[1..k] este promițător}
            Dacă soluție(n, k, posibil)
                atunci {o soluție! (rămânem pe nivelul k)}
                    Tiparește posibil[1..k]
                altfel {e doar un vector promițător}
                    Fie k := k + 1; {pas în față (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 și funcțiile *condiții-continuare* și *soluție* sunt dependente de problema care se rezolvă. De exemplu, pentru generarea permutărilor funcțiile menționate sunt:

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

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

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

```

    sfcât
    conditii-continuare:=kod
sf-conditii

Funcția soluții(n, k, posibil) este:
    Soluții := (k = n)
sf-solutii

```

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

Subalgoritmul Backtracking(k, posibil) este:

```

{posibil[1..k] este promițător}
Dacă soluție(n, k, posibil) atunci
    {o soluție! terminare apel recursiv, astfel}
    Tipareste posibil[1..k]
    {rămânem pe același nivel}
altfel
    Pentru fiecare v valoare posibilă pentru posibil[k+1] execută
        Dacă condiții-continuare(k + 1, posibil, v) atunci
            posibil[k + 1] := v
            Backtracking(k + 1, posibil)
            {pas în față}
        sfdacă
    sfpentru
sfdacă
{terminare apel Backtracking(k, posibil)}
sf-Backtracking {deci, pas în spate (revenire)}

```

cu apelul inițial **Cheamă** Backtracking(0, posibil).

1.2. Concepte OOP în limbaje de programare

1.2.1. Noțiunea de clasă

1.2.1.1. Realizarea protecției datelor prin metoda programării modulare

Dezvoltarea programelor prin programare procedurală înseamnă folosirea unor funcții și proceduri pentru scrierea programelor. În limbajul C lor le corespund funcțiile care returnează o valoare sau nu. Însă în cazul aplicațiilor mai mari ar fi de dorit să putem realiza și o protecție corespunzătoare a datelor. Acest lucru ar însemna că numai o parte a funcțiilor să aibă acces la datele problemei, acelea care se referă la datele respective. Programarea modulară oferă o posibilitate de realizare a protecției datelor prin folosirea clasei de memorie static. Dacă într-un fișier se declară o dată aparținând clasei de memorie statică în afara funcțiilor, atunci ea poate fi folosită începând cu locul declarării până la sfârșitul modulului respectiv, dar nu și 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 funcții corespunzătoare pentru inițializarea vectorului, eliberarea zonei de memorie ocupate și ridicarea la pătrat, respectiv afișarea elementelor vectorului. O posibilitate de implementare a modului este prezentată în fișierul **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 obținând un program obiect. Un exemplu de program principal este prezentat în fișierul **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 deși 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 și 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 și operațiile (funcțiile) 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 și declararea sau definirea funcțiilor 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();
};
```

Funcțiile declarate sau definite în interiorul structurii vor fi numite *funcții membru* iar datele *date membru*. Dacă o funcție membru este definită în interiorul structurii (ca și funcția *distr* din exemplul de mai sus), atunci ea se consideră funcție *inline*. Dacă o funcție membru se definește în afara structurii, atunci numele funcției se va înlocui cu numele tipului abstract urmat de operatorul de rezoluție (::) și numele funcției membru. Astfel funcțiile *init*, *lapatrat* și *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;
}
```

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

1.2.1.3. Declararea claselor

Un tip abstract de date clasă se declară ca și o structură, dar cuvântul cheie `struct` se înlocuiește cu `class`. Ca și în cazul structurilor referirea la tipul de dată clasă se face cu numele după cuvântul cheie `class` (numele clasei). Protecția datelor se realizează cu modificatorii de protecție: *private*, *protected* și *public*. După modificatorul de protecție se pune caracterul `:`. Modificatorul *private* și *protected* reprezintă date protejate, iar *public* date neprotejate. Domeniul de valabilitate a modificatorilor de protecție este până la următorul modificator din interiorul clasei, modificatorul implicit fiind *private*. Menționăm că și în cazul structurilor putem să folosim modificatori de protecție, 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* și *d* au fost declarate ca date de tip *private* (protejate), iar funcțiile membru au fost declarate publice (neprotejate). Bineînțeles, o parte din datele membru pot fi declarate publice, și unele funcții membru pot fi declarate protejate, dacă natura problemei cere acest lucru. În general, datele membru protejate pot fi accesate numai de funcțiile membru ale clasei respective și eventual de alte funcții numite *funcții prietene* (sau funcții *friend*).

O altă observație importantă referitoare la exemplul de mai sus este că inițializarea datelor membru și eliberarea zonei de memorie ocupată s-a făcut prin funcții membru specifice.

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

```
nume_clasă listă_de_obiecte;
```

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

```
vector v;
```

Inițializarea obiectelor se face cu o funcție membru specifică numită *constructor*. În cazul distrugerii unui obiect se apelează automat o altă funcție 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 și 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 funcție inline. Definirea funcțiilor membru care sunt declarate, dar nu sunt definite în interiorul clasei se face ca și în cazul tipurilor abstracte de date de tip *struct*, folosind operatorul de rezoluție.

1.2.1.4. Membrii unei clase. Pointerul *this*

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

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

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

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

În interiorul funcțiilor membru însă referirea la datele respectiv funcțiile 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 funcție membru, și folosește acest pointer pentru identificarea datelor și funcțiilor 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 funcției membru *afiseaza* se utilizează de exemplu data membru *d*, atunci ea este interpretată de către compilator ca și *this->d*.

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

1.2.1.5. Constructorul

Inițializarea obiectelor se face cu o funcție membru specifică numită constructor. Numele constructorului trebuie să coincidă cu numele clasei. O clasă poate să aibă mai mulți constructori. În acest caz aceste funcții membru au numele comun, ceea ce se poate face datorită posibilității de supraîncărcare a funcțiilor. Bineînțeles, în acest caz numărul și/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 mulți constructori, având ca date membru numele și prenumele unei persoane, și cu o funcție membru pentru afișarea numelui complet.

Fișierul **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();
};
```

Fișierul **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;
}

```

Fișierul `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 copiere

    D.afiseaza();
    return 0;
}

```

Observăm prezența a doi constructori specifici: *constructorul implicit* și *constructorul de copiere*. Dacă o clasă are constructor fără parametri atunci el se va numi *constructor implicit*. *Constructorul de copiere* se folosește la inițializarea obiectelor folosind un obiect de același tip (în exemplul de mai sus o persoană cu numele și 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 conțină 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 definiți de programator, sau obiectul care se inițializează cu un constructor implicit, sau cu toți parametrii implicați.

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

Fișierul `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 implicați
    } //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* și *sotie* au fost declarați ca și referințe la tipul *persoana*. Dacă ar fi fost declarați ca parametri formali de tip *persoana*, atunci în cazul declarației:

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

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

1.2.1.6. Destructorul

Destructorul este funcția membru care se apelează în cazul distrugerii obiectului. Destructorul obiectelor globale se apelează automat la sfârșitul funcției *main* ca parte a funcției *exit*. Deci, nu este indicată folosirea funcției *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 și un mod explicit de apelare a destructorului, în acest caz numele destructorului trebuie precedat de numele clasei și operatorul de rezoluție.

Numele destructorului începe cu caracterul `~` după care urmează numele clasei. Ca și în cazul constructorului, destructorul nu returnează o valoare și nu este permisă nici folosirea cuvântului cheie `void`. Apelarea destructorului în diferite situații este ilustrată de următorul exemplu. Fișierul `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. Relații între clase

1.3.1. Bazele teoretice

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

În viața de zi cu zi însă ne întâlnim nu numai cu obiecte separate, dar și 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: *moștenirea (inheritance)*. Acest lucru înseamnă că se moștenesc toate datele și funcțiile membru ale clasei de bază de către clasa derivată, dar se pot adăuga elemente noi (date membru și funcții membru) în clasa derivată. În cazul în care o clasă derivată are mai multe clase de bază se vorbește despre *moștenire multiplă*.

O altă proprietate importantă a obiectelor care aparțin clasei derivate este că funcțiile membru moștenite pot fi supraîncărcate. Acest lucru înseamnă că o operație referitoare la obiectele care aparțin ierarhiei are un singur identificator, dar funcțiile care descriu această operație pot fi diferite. Deci, numele funcției și lista parametrilor formali este aceeași în clasa de bază și în clasa derivată, dar descrierea funcțiilor diferă între ele. Astfel, în clasa derivată funcțiile membru pot fi specifice clasei respective, deși operația se identifică prin același nume. Această proprietate se numește *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 și funcții membru noi
};
```

unde `lista_claselor_de_bază` este de forma:

`elem_1, elem_2, ..., elem_n`

și `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* și *private* se numesc și de această dată *modificatori de protecție*. 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 protecție 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* și *public* devin de tip *protected*, respectiv *private* dacă modificatorul de protecție referitor la clasa de bază este *protected* respectiv *private*, și rămân neschimbate dacă modificatorul de protecție referitor la clasa de bază este *public*. Din acest motiv în general datele membru se declară de tip *protected* și modificatorul de protecție referitor la clasa de bază este *public*. Astfel datele membru pot fi accesate, dar rămân protejate și în clasa derivată.

1.3.3. Funcții virtuale

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

Fișierul `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 execuție se obține 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 funcției *main* s-a apelat funcția membru *functia_2* moștenită de la clasa de bază, dar funcția membru *functia_1* apelată de *functia_2* s-a determinat încă în faza de compilare. În consecință, deși funcția membru *functia_1* s-a supraîncărcat în clasa derivată nu s-a apelat funcția supraîncărcată ci funcția membru a clasei de bază.

Acest neajuns se poate înlătura cu ajutorul introducerii noțiunii de funcție membru *virtuală*. Dacă funcția membru este *virtuală*, atunci la orice apelare a ei, determinarea funcției membru corespunzătoare a ierarhiei de clase nu se va face la compilare ci la execuție, în funcție de natura obiectului pentru care s-a făcut apelarea. Această proprietate se numește *legare dinamică*, iar dacă determinarea funcției membru se face la compilare, atunci se vorbește de *legare statică*.

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

Acest lucru se poate realiza declarând *functia_1* ca funcție membru *virtuală*. Astfel, pentru orice apelare a funcției membru *functia_1*, determinarea celui exemplar al funcției membru din ierarhia de clase care se va executa, se va face la execuție și nu la compilare. Ca urmare, funcția membru *functia_1* se determină prin *legare dinamică*.

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

Dacă o funcție 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 obținut prin execuție 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 funcția membru *functia_1* a clasei derivate.

Prezentăm în continuare un alt exemplu în care apare necesitatea introducerii funcțiilor membru *virtuale*. Să se definească clasa *fracție* referitoare la numerele raționale, având ca date membru numărătorul și numitorul fracției. Clasa trebuie să aibă un constructor, valoarea implicită pentru numărător fiind zero iar pentru numitor unu, precum și două funcții membru: *produs* pentru a calcula produsul a două fracții și *inmulteste* pentru înmulțirea obiectului curent cu fracția dată ca și parametru. De asemenea, clasa *fracție* trebuie să aibă și o funcție membru pentru afișarea unui număr rațional. Folosind clasa *fracție* ca și clasă de bază se va defini clasa derivată *fracție_scrie*, pentru care se va supraîncărca funcția *produs*, astfel încât concomitent cu efectuarea înmulțirii să se afișeze pe *stdout* operația respectivă. Funcția *inmulteste* nu se va supraîncărca, dar operația efectuată trebuie să se afișeze pe dispozitivul standard de ieșire și în acest caz. Fișierul `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 execuție se obține:

```

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

```

Observăm că rezultatul nu este cel dorit, deoarece operația de înmulțire s-a afișat numai o singură dată, și anume pentru expresia `r1 = p1.produs(q1)`. În cazul expresiei `r2 = p1.inmulteste(q1)` însă nu s-a afișat operația de înmulțire. Acest lucru se datorează faptului că funcția membru *inmulteste* nu s-a supraîncărcat în clasa derivată. Deci s-a apelat funcția moștenită de la clasa *fractie*. În interiorul funcției *inmulteste* s-a apelat funcția membru *produs*, dar deoarece această funcție membru s-a determinat încă în faza de compilare, rezultă că s-a apelat funcția referitoare la clasa *fractie* și nu cea referitoare la clasa derivată *fractie_scrie*. Deci, afișarea operației s-a efectuat numai o singură dată.

Soluția este, ca și în exemplul anterior, declararea unei funcții membru virtuale, și anume funcția *produs* se va declara ca funcție 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 obținem:

```

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 afișarea operației s-a făcut de două ori, pentru ambele expresii. Funcțiile virtuale, ca și alte funcții membru de fapt, nu trebuie neapărat supraîncărcate în clasele derivate. Dacă nu sunt supraîncărcate atunci se moștenește funcția membru de la un nivel superior.

Determinarea funcțiilor membru virtuale corespunzătoare se face pe baza unor tabele construite și gestionate în mod automat. Obiectele claselor care au funcții membru virtuale conțin și un pointer către tabela construită. De aceea gestionarea funcțiilor membru virtuale necesită mai multă memorie și un timp de execuție mai îndelungat.

1.3.4. Clase abstracte

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

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

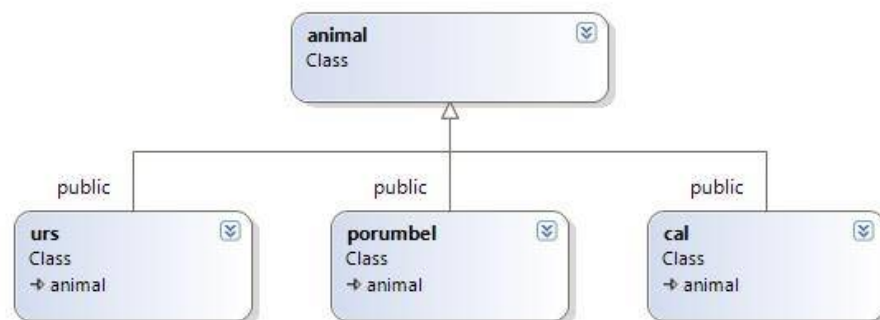


Figura 1.3. Ierarhie de clase referitoare la animale

Funcția virtuală pură este o funcție membru care este declarată, dar nu este definită în clasa respectivă. Ea *trebuie* definită într-o clasă derivată. Funcția membru virtuală pură se declară în modul următor. Antetul obișnuit al funcției este precedat de cuvântul cheie *virtual*, și antetul se termină cu $= 0$. După cum arată numele și declarația ei, funcția membru virtuală pură este o funcție virtuală, deci selectarea exemplarului funcției din ierarhia de clase se va face în timpul execuției programului.

Clasele care conțin cel puțin o funcție membru virtuală pură se vor numi *clase abstracte*.

Deoarece clasele abstracte conțin funcții membru care nu sunt definite, nu se pot crea obiecte aparținând claselor abstracte. Dacă funcția virtuală pură nu s-a definit în clasa derivată atunci și clasa derivată va fi clasă abstractă și ca atare nu se pot defini obiecte aparținând acelei clase.

Să considerăm exemplul de mai sus și 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. Afișarea acestui rezultat se va face de către o funcție membru a clasei *animal* care nu se supraîncarcă în clasele derivate. Fișierul **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 deși clasa *animal* este clasă abstractă, este utilă introducerea ei, pentru că multe funcții membru pot fi definite în clasa de bază și moștenite fără modificări în cele trei clase derivate.

1.3.5. Interfețe

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

În continuare introducem o clasă abstractă *Vehicul*, care nu conține numai funcții membru virtuale pure, și două clase derivate din această clasă abstractă. Fișierul `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 și interacțiuni între obiecte în UML: Pachete, clase și interfețe. Relații între clase și interfețe. Obiecte. Mesaje

Limbajul de modelare unificat UML (Unified Modelling Language) [29] definește un set de *elemente de modelare* și *notații grafice* asociate acestora. Elementele de modelare pot fi folosite pentru descrierea oricărui sistem software. În particular, UML conține elemente ce pot fi folosite și pentru cele orientate pe obiecte.

Această secțiune conține câteva elemente de bază folosite pentru descrierea **structurii** și **comportamentului** unui sistem software orientat pe obiecte - *diagrame de clase* și de *interacțiuni între obiecte*. Aceste elemente corespund selecției facute în [30], capitolele 3 și 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 și (C) care este **legătura dintre modele și codul scris**.

Pentru a răspunde pe scurt acestor întrebări, vom da exemple ce se referă la o aplicație folosită de un casier pentru înregistrarea vânzărilor la un punct de vânzare într-un magazin. Aplicația este numită **POS** (**P**oint **o**f **S**ale) și 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 și 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 și **nu cum furnizează acest comportament**. Ele se mai numesc și **modele ale domeniului** (sau **modele de afaceri - business models**) și descriu spațiul problemei. Din perspectivă structurală, diagramele de clase sunt folosite pentru a defini **conceptele domeniului**. Diagramele de interacțiune sunt rar folosite în cazul acestor

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

Primul model din **Figura 1.4** prezintă un extras din modelul conceptual pentru aplicația 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 funcționează 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 și de interacțiuni între obiecte constituie două instrumente de bază pentru descrierea detaliată a sistemului (**proiectare detaliată**). Desigur, se folosesc și alte elemente de modelare, structurale și comportamentale, dar acestea nu sunt discutate aici, de exemplu - colaborări, mașini cu stări, activități, etc.

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

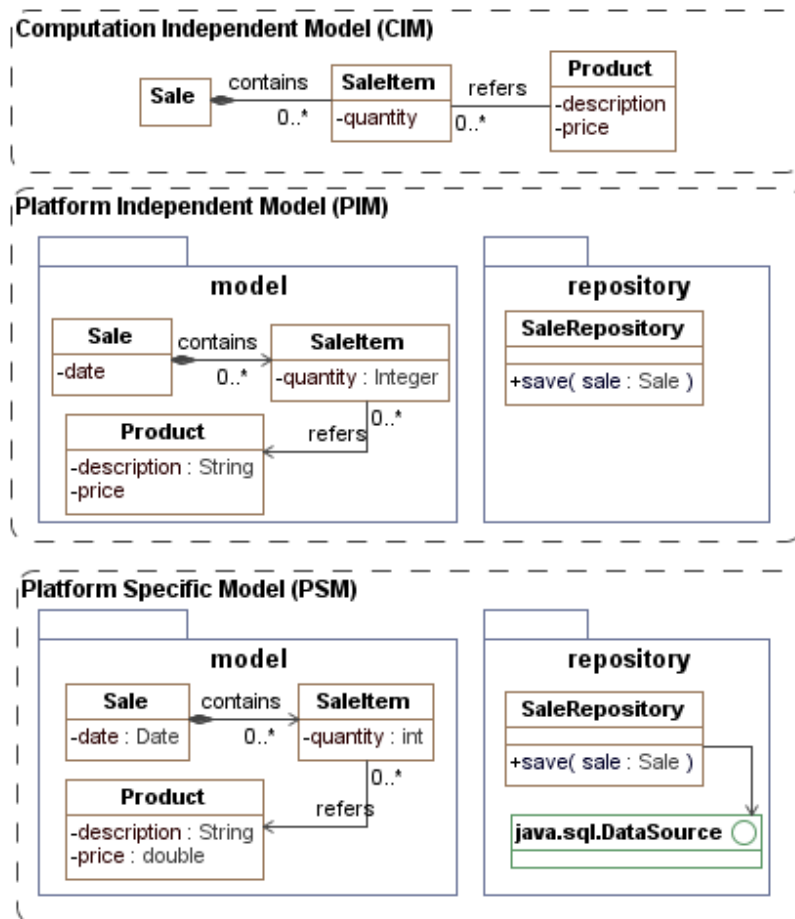


Figura 1.4 Tipuri de modele

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 și de interacțiune 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 și 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 **corespondențe între elementele modelelor și elementele platformei alese**.

B. Procese de dezvoltare și 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] și crează modele PIM, opțional derivate din modele CIM. Apoi, din modele PIM, generează cod, folosind în mod opțional un model intermediar 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 aplicații orientate pe servicii folosesc limbaje/extensii ale UML independente de platformă, de exemplu SoaML¹.

Există **procesele dirijate de modele care nu folosesc modele PIM ci direct modele PSM**. Acestea se bazează pe specificații elaborate pentru anumite platforme, de exemplu arhitecturi bazate pe componente ce oferă servicii, SCA².

Procesele de dezvoltare mai elaborate de tipul RUP³ (pentru sisteme mari), recomandă folosirea tuturor modelelor **CIM, PIM și PSM**, în contextul folosirii instrumentelor CASE.

Procesele de dezvoltare de tip agil nu recomandă în general folosirea instrumentelor CASE, de exemplu dezvoltarea dirijată de teste⁴ sau dezvoltarea agilă dirijată de modele⁵, dar recomandă construirea unor modele înainte de a începe codificarea. **Modelele sunt de fapt schițe** (scrise pe hârtie sau pe tablă) și sunt folosite pentru a comunica ideea despre proiectarea sistemului.

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

O ultimă și recentă categorie de procese de dezvoltare care propune folosirea modelelor **PIM și generarea directă și 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 așteptăm la adoptarea unui **stil nou de dezvoltare** în care vom construi doar **modele** și vom scrie **cod într-un limbaj textual⁷ definit pe elementele din aceste modele.** Astfel, modelele PSM și codul scris în limbaje ca și Java, C++ sau C# vor fi lăsate în grija instrumentelor CASE care le vor genera automat.

C. Corespondența dintre modele și cod

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

Ținând cont și de modelele executabile amintite anterior (care sunt la nivel PIM), în secțiunile care urmează vom discuta numai **corespondențele dintre modele PIM și limbajele C++, Java și C#.** Modelele PSM conțin în plus față de cele PIM și tipuri de date specifice anumitor limbaje, astfel corespondențele dintre modele PSM și cod sunt aceleași, doar că sunt prezente în modele și 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 și relațiile dintre acestea. Elementele structurale selectate în această secțiune sunt (a) **tipuri de obiecte:** clase, interfețe, enumerări; (b) **gruparea elementelor** folosind pachete și (c) **relații între aceste elemente:** asocieri, generalizări, realizări și dependențe.

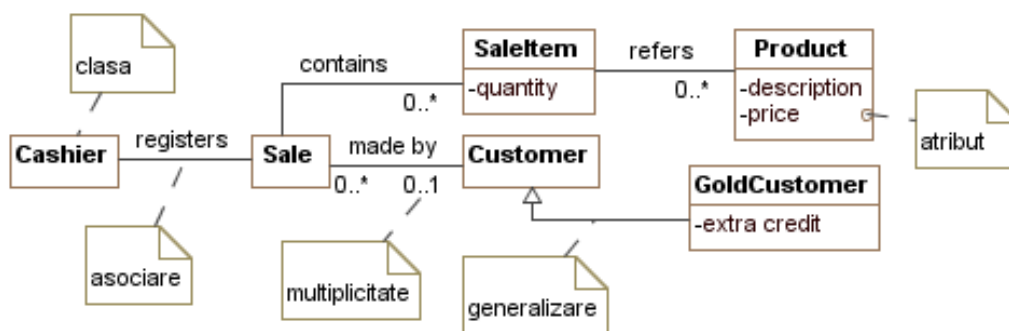


Figura 1.5 Model conceptual

Figura 1.5 prezintă un model conceptual inițial pentru POS. Clasele sunt folosite pentru a identifica conceptele acestui domeniu. Acolo unde nu e relevant, compartimentul cu atributele claselor este ascuns. Proprietățile claselor sunt definite prin atribute și 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 și sunt folosite pentru a genera modele PIM. Ca și modele CIM, ele pot să nu conțină 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) și dependențe între acestea - a se vedea **Figura 1.6** pentru *POS*. Pachetele sunt definite cu responsabilități coezive, de exemplu interacțiunea cu utilizatorul (*ui*), fațadă peste domeniu (*service*), entități (*model*) și depozite de obiecte sau obiecte de acces la date (*repository*, *inmemory repository*).

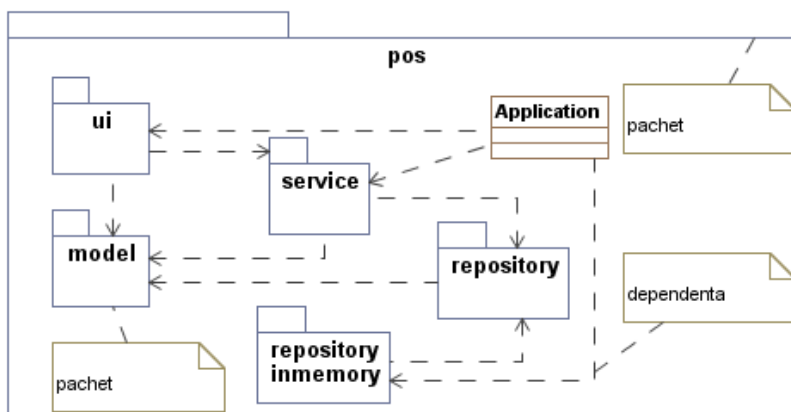


Figura 1.6 Arhitectură stratificată

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 responsabilității unice (Single Responsibility), conform căruia obiectele ar trebui să aibă o singură responsabilitate, iar obiectele cu responsabilități înrudite ar trebui grupate logic.

Dependența 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. Dependențele 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* și *model*, *service* depinde de *model* și *repository*, însă *service* nu depinde de implementarea concretă pentru *repository*, anume *repository inmemory*. Inversarea acestei ultime dependențe urmează un alt principiu SOLID, anume inversarea dependențelor (Dependency Inversion). **Figura 1.7** prezintă detalii privind această inversare a dependențelor dintre *service* și *repository inmemory*. În loc ca *StoreService* să fie dependent de implementarea concretă *InmemorySaleRepository*, a fost introdusă interfața *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_\(object-oriented_design\)](http://en.wikipedia.org/wiki/SOLID_(object-oriented_design))

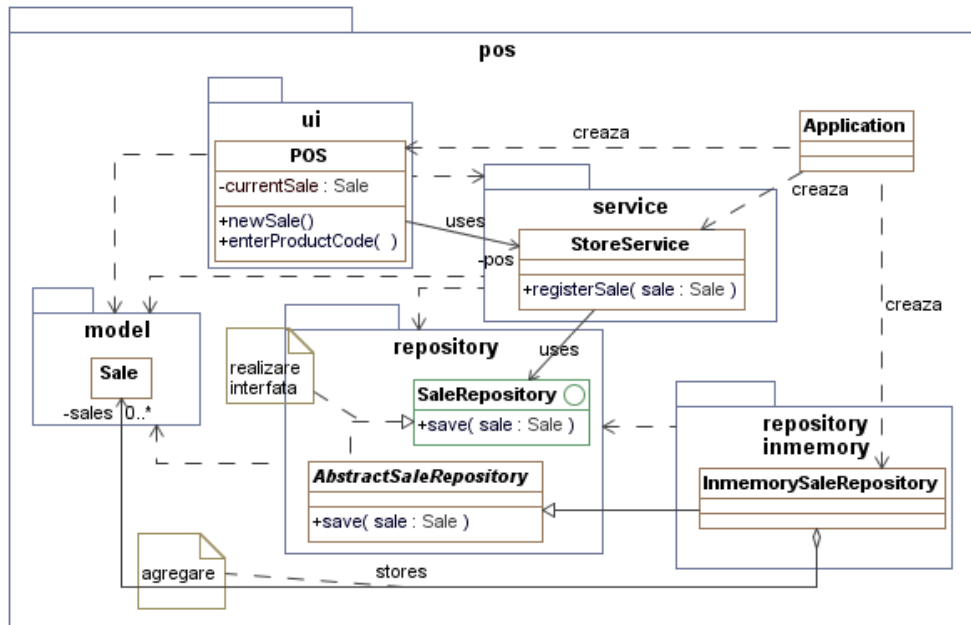


Figura 1.7 Arhitectură stratificată - justificare dependențe

La nivel PIM sau PSM, diagramele de clase sunt folosite pentru a rafina entitățile și relațiile dintre acestea - a se vedea **Figura 1.8**. Aceste elemente vor fi discutate în subsecțiunile care urmează.

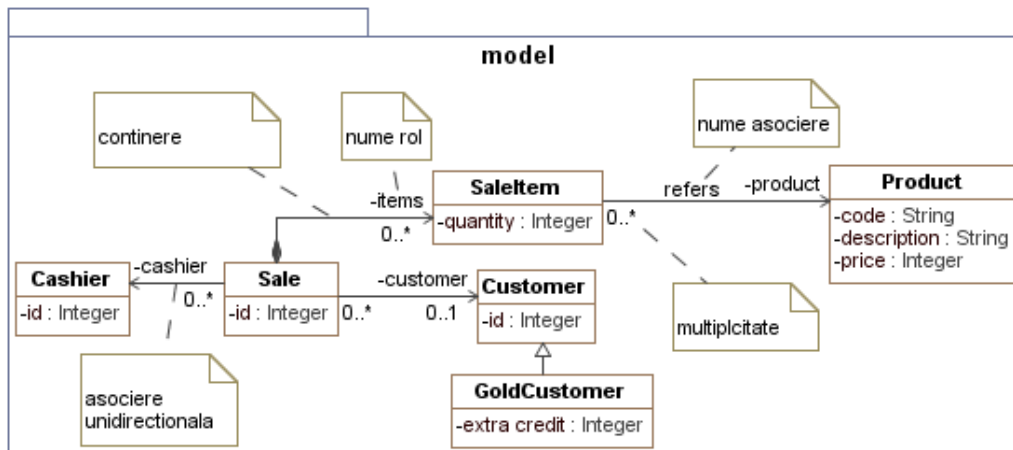


Figura 1.8 Entitățile POS

A. Pachete

Pachetele UML [29, 30] grupează elemente din model și oferă un spațiu de nume pentru elementele grupate. Din perspectiva elementelor discutate în acest document, pachetele pot conține tipuri de date și 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 și spațiilor de nume C++ și C#. Pachetele UML sunt referite folosind operatorul de rezoluție ::,

la fel ca și în C++ și 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 preponderență pachetele unui sistem sunt folosite pentru a descrie arhitectura acestuia - a se vedea **Figura 1.6** pentru sistemul *POS*. Dependentele între pachete indică un sumar al dependențelor dintre elementele conținute și elementele din alte pachete. Din perspectivă arhitecturală, buna gestionare a acestor dependențe este crucială în procesul de construire și întreținere a sistemului.

B. Clase

O clasă UML [29, 30] reprezintă o mulțime de obiecte cu aceleași elemente structurale (proprietăți) și comportamentale (operații). Clasele UML sunt tipuri de date și corespund claselor din limbajele Java, C++ și C#. O clasă poate fi declarată **abstractă** și în acest caz nu poate fi instanțiată la fel ca și în Java, C++ și C#.

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

O clasă UML poate **realiza/implementa** mai multe interfețe la fel ca și în Java sau C#. Corespondența între modelele ce conțin clase ce implementează mai multe interfețe și C++ este realizată via clase C++ pur abstracte și moștenire multiplă.

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

Principiul substituției este aplicabil pentru instanțele de tipul unor clase și interfețe, la fel ca și în Java, C++ și C#. Adică, instanțele din program pot fi înlocuite cu instanțe ale tipurilor derivate fără să alterăm semantic programul.

C. Interfețe

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

SaleRepository din **Figura 1.7** este o interfață. Atunci când evidențierea metodelor interfeței nu este relevantă, notația grafică pentru interfețe este cea din **Figura 1.9**.



Figura 1.9 Interfață, enumerare și tipuri structurate

D. Enumerări și obiecte valorice

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

Tipurile structurate [29, 30] se modelează folosind stereotipul *datatype* și corespund structurilor din C++/C# și tipurilor primitive din Java - a se vedea **Eroare! Fără sursă de referință.** Instanțele acestor tipuri sunt identificate doar prin valoarea lor. Ele sunt folosite pentru a descrie proprietățile claselor și corespund obiectelor valorice (șablonul *value object*¹⁰), cu deosebirea că nu pot avea identitate.

E. Generalizări și realizări de interfețe

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

Realizarea unei interfețe în UML [29, 30] reprezintă o relație între o clasă și o interfață prin care se indică faptul că clasa este conformă contractului specificat de interfață. Aceste realizări corespund implementărilor interfețelor din Java și C#, respectiv moștenirii în C++. A se vedea notațiile grafice dintre *AbstractSaleRepository* și *SaleRepository* în **Figura 1.7** și **Figura 1.9**.

F. Proprietăți

Proprietățile [29, 30] reprezintă aspecte structurale ale unui tip de date. Proprietățile unei clase sunt introduse prin atribute și 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 și **vizibilitatea** care poate fi publică (+), privată (-), protejată (#) sau la nivel de pachet (fără specificator). Vizibilitatea UML corespunde specificatorilor de acces cu același nume din Java, având aceeași semantică. Vizibilitatea la nivel de pachet nu se regăsește în C++, iar în C# are o corespondență prin specificatorul intern din C# dar care are și conotații de distribuire a elementelor software (elementele declarate intern în C# fiind accesibile doar în distribuția binară dll sau exe din care acestea fac parte).

Celelalte elemente folosite la declararea unei proprietăți sunt opționale. **Tipul** proprietății poate fi oricare: clasă, interfață, enumerare, tip structurat sau tip primitiv. **Tipurile primitive în UML** sunt tipuri valorice [29]. UML definește următoarele tipuri primitive: String, Integer, Boolean și UnlimitedNatural. Primele trei tipuri primitive sunt în corespondență cu tipurile cu același nume din limbajele Java, C++ și C#, dar cu observațiile:

- Tipul String este clasă în Java și C#, instanțele de tip String fiind nemodificabile, spre deosebire de C++ unde șirurile de caractere sunt modificabile. Codificarea caracterelor nu este precizată în UML, în timp ce în Java și 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* și *n* valori, unde *m* și *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 mulțime), în caz contrar precizăm explicit prin **nonunique** (adică container cu valori posibil duplicate).
- Valorile pot fi referite prin indici sau nu - **implicit nu** (deci colecție), în caz contrar precizăm explicit **ordered** (deci listă).

Exemple de proprietăți:

multime : *Integer[0..*]* - mulțime de valori întregi (unice)
lista : *Integer[0..*] {ordered}* - listă cu valori întregi și distincte (unice)
lista : *Integer[0..*] {ordered, nonunique}* - listă de întregi
colectie : *Integer[0..*] {nonunique}* - colecție de întregi

Proprietăților din UML le corespund câmpuri sau variabile de tip obiect în Java și C++, respectiv proprietăți în C#. Dificultăți de interpretare se ridică în ceea ce privește proprietățile cu multiplicitate *m..**. Pentru exemplele de mai sus putem considera următoarele corespondențe cu Java (în mod similar și cu C++/C#):

- mulțimi de întregi:
 - *int[] multime* sau *Integer[] multime*, urmând să asigurăm prin operații că *multime* va conține valori distincte, sau cel mai potrivit
 - *java.util.Set multime*
- liste cu valori întregi și distincte:
 - *int[] lista*, *Integer[] lista* sau *java.util.List lista*, urmând să asigurăm prin operații că *lista* va conține valori distincte
- liste de întregi:
 - *int[] lista*, *Integer[] lista*, sau *java.util.List lista*
- colecții 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 instanțe ale unor tipuri de date. În acest sens, o asociere este un tip de date care leagă proprietăți ale altor tipuri de date.

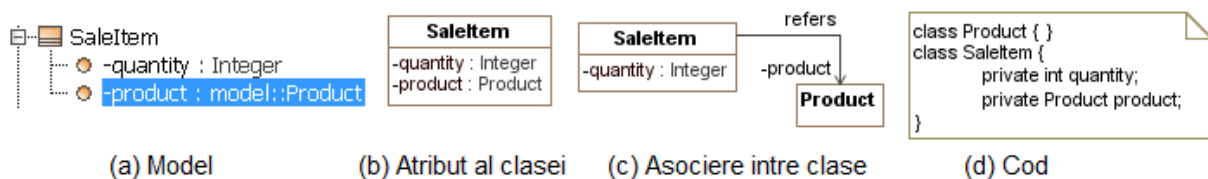


Figura 1.10 Asocieri unidirecționale

Figura 1.10 (a) prezintă modelul rezultat după adăugarea atributelor *quantity* și *product* în clasa *SaleItem* reprezentată grafic în diagrama (b). Codul (d) scris în Java/C# corespunde acestei situații. Dacă considerăm că e mai potrivită o reprezentare grafică pentru relația dintre clasele *SaleItem* și *Product*, atunci în loc să adăugăm *product* ca și atribut, folosim o asociere unidirecțională de la *SaleItem* spre *Product*. Atunci când se adaugă asocierea unidirecțională, în model se creează o asociere și o proprietate în clasa *SaleItem*, având numele rolului, adică *product*. Astfel, codul (d) corespunde reprezentării grafice (c) a modelului (a) care mai conține o asociere nearătată în figură. **Asocierile unidirecționale** introduc proprietăți în clasa sursă, de tipul clasei destinație. Numele proprietății coincide cu numele rolului asocierii, iar forma generală de definiție a proprietăților (prezentată la începutul acestei subsecțiuni) se aplică și în acest caz.

Decizia folosirii asocierilor în locul atributelor este luată în funcție de context. De exemplu, atunci când modelăm entitățile unui aplicații folosim asocieri pentru a indica relațiile dintre entități și folosim atribute atunci când descriem entitățile folosind obiecte valorice/descriptive. În general, folosim asocieri când dorim să evidențiem importanța tipurilor și a legăturilor dintre ele.

Asocierile bidirecționale leagă două proprietăți din două clase diferite sau din aceeași clasă. **Figura 1.11** prezintă o asociere bidirecțională între *SaleItem* și *Product*, precum și codul Java/C# corespunzător acestei situații.

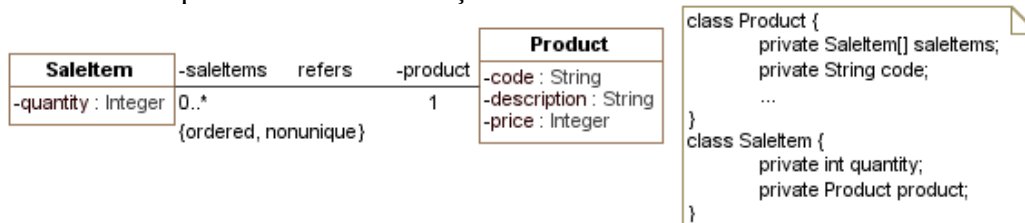


Figura 1.11 Asocieri bidirecționale

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

Relațiile întreg-parte sunt modelate în UML folosind agregări și conțineri. 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* și *Sale* că primul obiect memorează toate obiectele de tip *Sale* (vânzările sunt parte a acestui depozit). O **conținere** este o agregare prin care se indică în plus că obiectele conținute pot fi părți 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 și stabilirea relațiilor de **agregare** și **conținere**.

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

G. Dependente

Între două elemente software, *client* și *furnizor*, există o **dependență** [29, 30] dacă schimbarea definiției *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 definiției mesajului în *F* va implica schimbări în *C* privind modul de transmitere. Ca regulă generală, ar trebui să minimizăm dependențele în model, în timp ce păstrăm coezive aceste elemente.

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

H. Operații

Operațiile în UML [29, 30] definesc comportamentul obiectelor și corespund metodelor din limbajele de programare orientate obiect. De fapt, operațiile specifică comportamentul (reprezintă antetul), iar corpul/implementarea este definită de elemente comportamentale ca și interacțiuni, mașini cu stări și activități - implementările sunt numite **metode** în UML. Sintaxa specificării operațiilor este:

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

unde *vizibilitatea*, *tipul-returnat* și *proprietățile* sunt definite ca și în cazul proprietăților claselor. În lista *proprietăților* operației se poate preciza dacă este doar o operație de interogare *{query}*, adică o operație ce nu modifică starea obiectului apelant - implicit, operațiile sunt considerate comenzi, adică modifică starea obiectelor. Parametrii în *lista-parametrilor* sunt separați prin virgulă, un parametru fiind de forma:

direcție nume: tip = valoare-implicită,

direcția putând fi: *in*, *out* și *in-out*, implicit fiind *in*.

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

1.4.2. Diagrame de interacțiune

Interacțiunile UML [29, 30] descriu comportamentul sistemului, indicând cum colaborează mai mulți participanți într-un anumit scenariu. Există mai multe tipuri de interacțiuni, dar în această secțiune vom discuta numai despre **diagrame de secvențe**, un tip de interacțiuni care descriu mesajele transmise între participanți.

În general diagramele de secvență descriu **un singur scenariu**. De exemplu, diagrama din **Figura** pune față în față casierul și sistemul *POS*, descriind fluxul normal de desfășurare pentru singurul caz de utilizare discutat aici, înregistrarea unei vânzări. O astfel de diagramă ajută la identificarea **interfeței publice a sistemului**. Pornind de la descrierea cazurilor de utilizare, acțiunile utilizatorilor sunt modelate ca și mesaje la care sistemul trebuie să răspundă. Mesajele 1, 2, 4 și 6 din **Figura** 1.12 indică faptul că sistemul va trebui să facă anumite calcule și 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 interacțiune. **Mesajele** au nume, sunt în general numerotate și pot indica un **răspuns** returnat. Toate mesajele din figurile acestei secțiuni sunt sincrone (cele asincrone nu intră în scopul acestui document). Interacțiunile pot conține **fragmente: cicluri și alternative**.

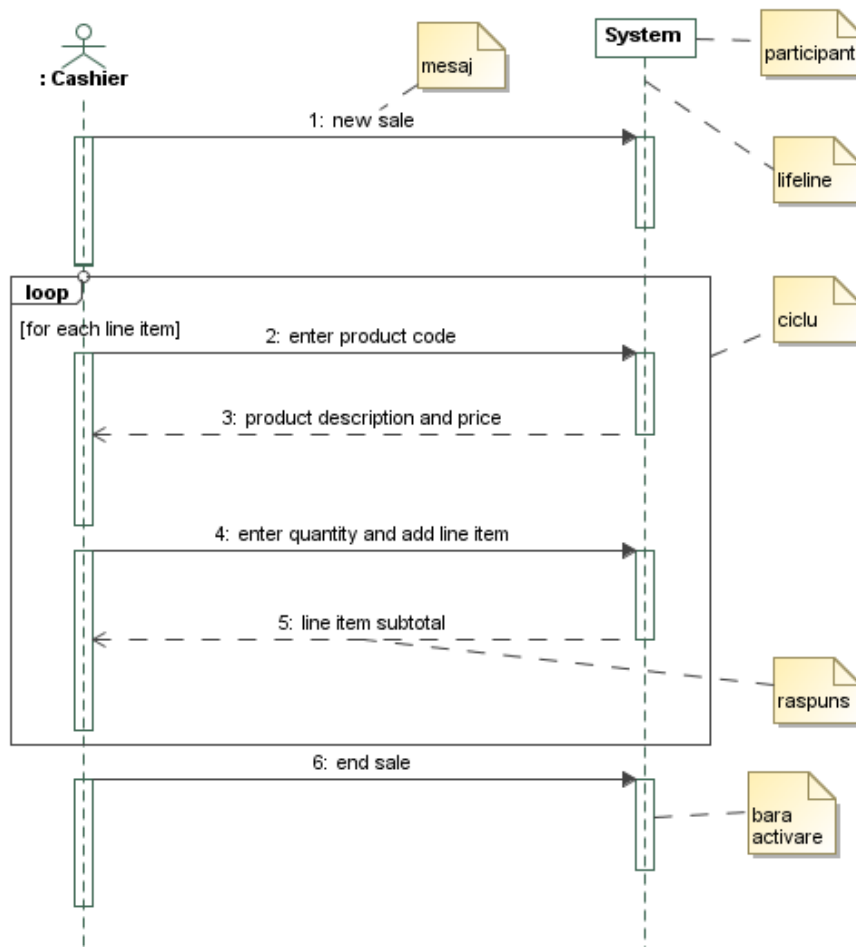


Figura 1.12 Interfața sistemului

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 **ce trebuie să facă sistemul**, se folosesc diagrame de secvență pentru a detalia **cum vor colabora obiectele din sistem**, conform responsabilităților precizate prin arhitectura stabilită. **Figura 1.13** prezintă detaliat colaborarea în cazul sistemului *POS*.

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

Participanții 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 interacțiunilor din **Figura 1.13**.

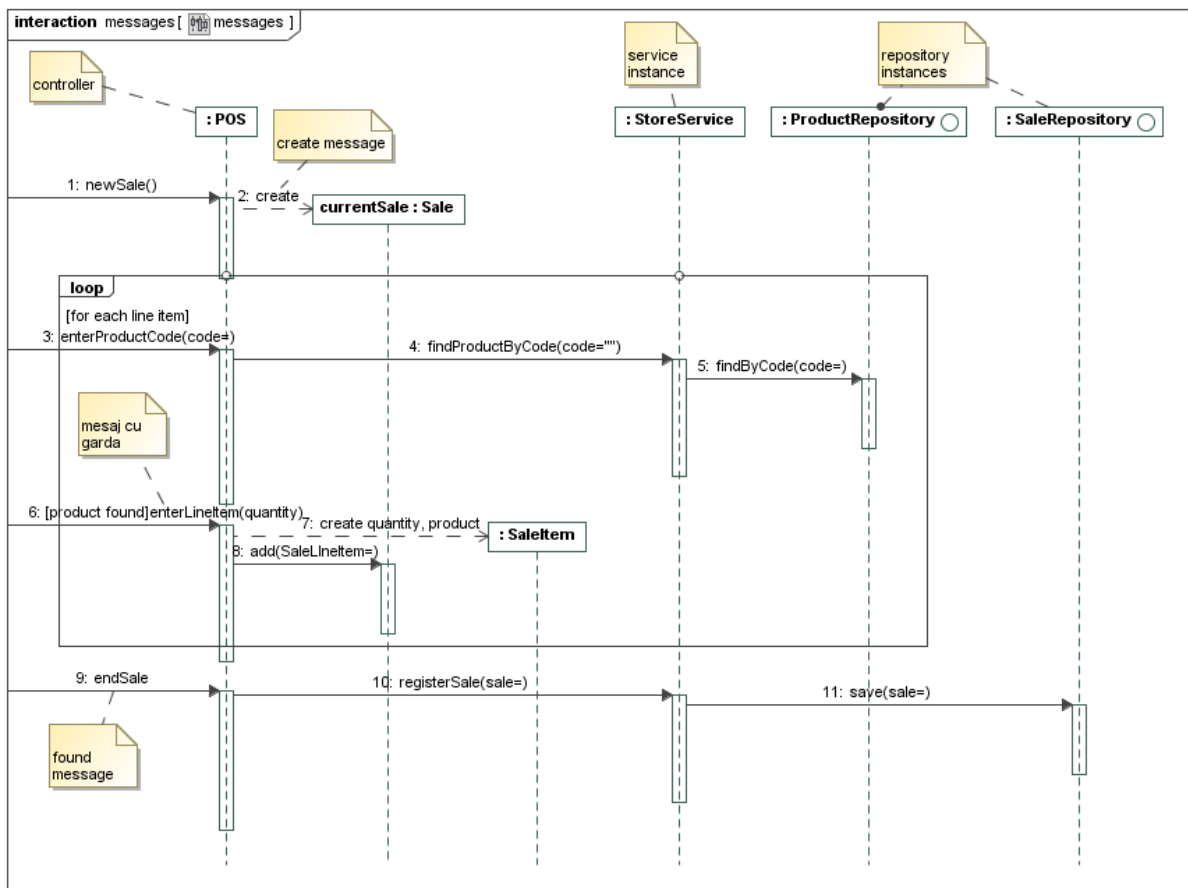


Figura 1.13 Proiectare detaliată - interacțiuni între obiecte

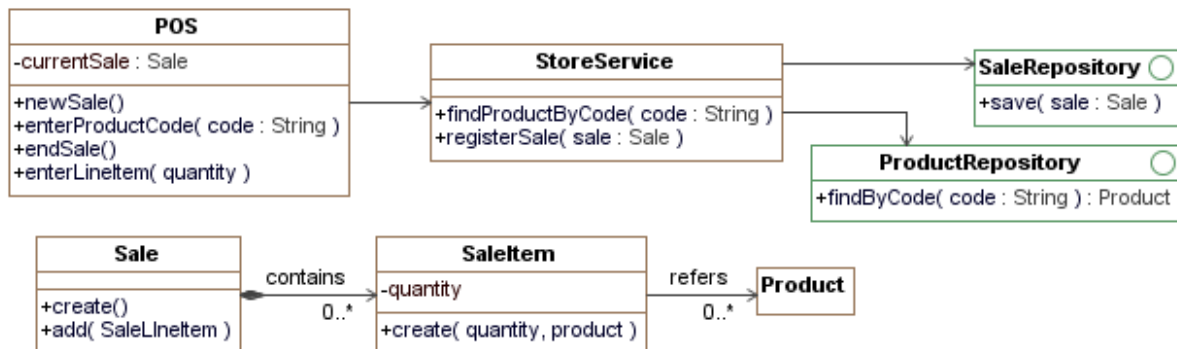


Figura 1.14 Proiectare detaliată - diagramă de clase

1.5. Liste și dicționare

În cele ce urmează vom prezenta două dintre containerele des folosite în programare și anume *listele* și *dicționarele*. Vom specifica tipurile abstracte de date corespunzătoare, indicând și specificând operațiile caracteristice. Pentru fiecare operație din interfața unui tip de date, vom da specificarea operației în limbaj natural, indicând datele și condițiile operației (**pre**), precum și rezultatele și postcondițiile operației (**post**).

1.5.1. Liste

În limbajul uzual cuvântul “listă” referă o “înșirare, 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 prețuri, listă de studenți, etc. Ordinea în listă poate fi interpretată ca un fel de „legătură” între elementele listei (după prima cumpăratură urmează a doua cumpăratură, după a doua cumpăratură urmează a treia cumpăratură, etc) sau poate fi văzută ca fiind dată de numărul de ordine al elementului în listă (1-a cumpăratură, a 2-a cumpăratură, etc). Tipul de date `Listă` care va fi definit în continuare permite implementarea în aplicații a acestor situații din lumea reală.

Ca urmare, o **listă** o putem vedea ca pe o secvență de elemente $\langle l_1, l_2, \dots, l_n \rangle$ de un același tip (`TElement`) aflate într-o anumită ordine, fiecare element având o *poziție* bine determinată în cadrul listei. Ca urmare, poziția elementelor în cadrul listei este esențială, astfel accesul, ștergerea și adăugarea se pot face pe orice poziție în listă. Lista poate fi văzută ca o colecție dinamică de elemente în care este esențială ordinea elementelor. Numărul n de elemente din listă se numește *lungimea* listei. O listă de lungime 0 se va numi lista *vidă*. Caracterul de dinamicitate al listei este dat de faptul că lista își poate modifica în timp lungimea prin adăugări și ștergeri de elemente în/din listă.

În 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 excepția ultimului element) are un singur succesori;
- fiecare element din listă (cu excepția primului element) are un singur predecesor.

Ca urmare, într-o listă liniară se pot insera elemente, șterge elemente, se poate determina succesori (predecesori) unui element, se poate accesa un element pe baza *poziției* sale în listă.

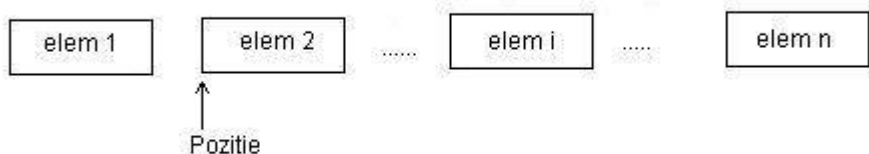
O listă liniară se numește **circulară** dacă se consideră predecesori primului nod a fi ultimul nod, iar succesori ultimului nod a fi primul nod.

Conform definiției anterioare, fiecare element al unei listei liniare are o poziție bine determinată în listă. De asemenea, este importantă prima poziție în cadrul listei, iar dacă se cunoaște poziția unui element din listă atunci pe baza acestei poziții se poate identifica elementul din listă, poziția elementului predecesor și poziția elementului succesori în listă (dacă acestea există). Ca urmare, într-o listă se poate stabili o ordine între pozițiile elementelor în cadrul listei.

Poziția 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, poziția 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/șterge elemente pe orice poziție în listă.
2. ca fiind dată de o **referință** la locația unde se stochează elementul listei (ex: pointer spre locația unde se memorează elementul).

Pentru a asigura generalitatea, vom abstractiza noțiunea de **poziție** a unui element în listă și vom presupune că elementele listei sunt accesate prin intermediul unei **poziții** generice.



Vom spune că o poziție p într-o listă este **validă** dacă este poziția unui element al listei. Spre exemplu, dacă p ar fi un pointer spre locația 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 depășește numărul de elemente din listă.

Ca urmare, dacă ne gândim la o listă liniară în care operațiile de acces/inserare/ștergere să se facă pe baza unei **poziții** 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

$$\mathcal{L} = \{l \mid l \text{ este o listă cu elemente de tip TEElement}\}$$

operații (interfața 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 sfârșitul listei

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

post: $l' \in \mathcal{L}, l'$ este l în care a fost adăugat e la sfârșit

adaugăInceput(l, e)

descriere: se adaugă un element la începutul listei

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

post: $l' \in \mathcal{L}, l'$ este l în care a fost adăugat e la început

valid(l, p)

descriere: funcție care verifică dacă o poziție în listă este validă

pre: $l \in \mathcal{L}, p$ e o poziție în l

post: $\text{valid} = \begin{matrix} \text{adevărat} & \text{dacă } p \text{ este o poziție validă în } l \\ \text{fals} & \text{în caz contrar} \end{matrix}$

adaugăÎnainte(l, p, e)

descriere: se adaugă un element înaintea unei anumite poziții în listă

pre: $l \in \mathcal{L}, e \in \text{TElement}, p$ e o poziție în l , $\text{valid}(l, p)$

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

adaugăDupă(l, p, e)

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

pre: $l \in \mathcal{L}, e \in \text{TElement}, p$ e o poziție în l , $\text{valid}(l, p)$

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

șterge(l, p, e)

descriere: se șterge elementul din listă situat pe o anumită poziție

pre: $l \in \mathcal{L}, e \in \text{TElement}, p$ e o poziție în l , $\text{valid}(l, p)$

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

element(l, p, e)

descriere: accesarea elementului din listă de pe o anumită poziție

pre: $l \in \mathcal{L}, e \in \text{TElement}, p$ e o poziție în l , $\text{valid}(l, p)$

post: $e \in \text{TElement}$, e este elementul de pe poziția p din l

modifica(l, p, e)

descriere: modificarea elementului din listă de pe o anumită poziție

pre: $l \in \mathcal{L}, e \in \text{TElement}, p$ e o poziție în l , $\text{valid}(l, p)$

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

prim(l)

descriere: funcție care returnează poziția primului element în listă

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

post: prim = poziția primului element din l sau o poziție care nu e validă dacă l e vidă

ultim(l)

descriere: funcție care returnează poziția ultimului element în listă

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

post: ultim = poziția ultimului element din l sau o poziție care nu e validă dacă l e vidă

următor(l, p)

descriere: funcție care returnează poziția în listă următoare unei poziții date

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

post: urmator = poziția din l care urmează poziției p sau o poziție care nu e validă dacă p e poziția ultimului element din listă

precedent(l, p)

descriere: funcție care returnează poziția în listă precedentă unei poziții date

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

post: precedent = poziția din l care precede poziția p sau o poziție care nu e validă dacă p e poziția primului element din listă

caută(l, e)

descriere: funcție care caută un element în listă

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

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

apare(l, e)

descriere: funcție care verifică apartenența unui element în listă

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

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

vidă(l)

descriere: funcție 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: funcție care returnează numărul de elemente din listă

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

post: *dim*=numărul de elemente din listă

iterator(l, i)

descriere: se construiește 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 operației *iterator* din interfața listei.

Subalgoritmul tipărire(l) este:

{pre: l este o listă}

{post: se tipăresc elementele listei}

 iterator(l, i) {se obține un iterator pe lista l }

 Cât timp valid(i) execută {cât timp iteratorul e valid}

 element(i, e) { e este elementul curent referit de iterator}

 @ tipărește e {se tipărește elementul curent}

 următor(i) {iteratorul referă următorul element}

 sf-cât

sf-tipărire

Observație

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 sfârșitul listei

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

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 sfârșitul listei

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

post: $l \in \mathcal{L}, l$ este modificată prin adăugarea lui e la sfârșit și păstrarea celorlate elemente pe pozițiile lor

1.5.2. Dicționare

Dicționarele reprezintă containere conținând elemente sunt forma unor perechi (cheie, valoare). Dicționarele păstrează elemente în așa fel încât ele să poată fi ușor localizate folosind chei. Operațiile de bază pe dicționare sunt căutare, adăugare și ștergere elemente. Într-un dicționar cheile sunt unice și în general, o cheie are o unică valoare asociată.

Aplicații ale dicționarelor sunt multiple. Spre exemplu:

- Informații despre conturi bancare: fiecare cont este un obiect identificat printr-un număr de cont (considerat cheia elementului) și informații adiționale (numele și adresa deținătorului contului, informații despre depozite, etc). Informațiile adiționale vor fi considerate ca fiind valoarea elementului.
- Informații despre abonați telefonici: fiecare abonat este un obiect identificat printr-un număr de telefon (considerat cheia elementului) și informații adiționale (numele și adresa abonatului, informații auxiliare, etc). Informațiile adiționale vor fi considerate ca fiind valoarea elementului.
- Informații despre studenți: fiecare student este un obiect identificat printr-un număr matricol (considerat cheia elementului) și informații adiționale (numele și adresa studentului, informații auxiliare, etc). Informațiile adiționale vor fi considerate ca fiind valoarea elementului.

Dăm în continuare specificația Tipului Abstract de Date **Dicționar**.

Tipul Abstract de Date DICȚIONAR

domeniu

$\mathcal{D} = \{d \mid d \text{ este un dicționar cu elemente } e = (c, v), c \text{ de tip } \mathbf{TCheie}, v \text{ de tip } \mathbf{TValoare}\}$

operații (interfața minimală)

creează(d)

descriere: se creează un dicționar vid

pre: true

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

adaugă(d, c, v)

descriere: se adaugă un element în dicționar

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

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

caută(d, c)

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

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

post: $\text{caută} = v \in \text{TValoare}$ dacă $(c, v) \in d$
elementul nul al **TValoare** în caz contrar

șterge(d, c)

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

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

post: $\text{șterge} = v \in \text{TValoare}$ 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: funcție care returnează numărul de elemente din listă

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

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

vid(d)

descriere: funcție care verifică dacă dicționarul este vid

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

post: $\text{vid} =$ adevărat în cazul în care d e dicționarul vid
fals în caz contrar

chei(d, m)

descriere: se determină mulțimea cheilor din dicționar

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

post: $m \in \mathcal{M}$, m este mulțimea cheilor din dicționarul d

valori(d, c)

descriere: se determină colecția valorilor din dicționar

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

post: $c \in \text{Col}$, c este colecția valorilor din dicționarul d

perechi(d, m)

descriere: se determină mulțimea perechilor (cheie, valoare) din dicționar

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

post: $m \in \mathcal{M}$, m este mulțimea perechilor (cheie, valoare) din dicționarul d

iterator(d, i)

descriere: se creează un iterator pe dicționar

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

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

distruge(d)

descriere: distruge un dicționar

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

post: dicționarul d a fost distrus

Reamintim modul în care va putea fi tipărit un dicționar (ca orice alt container care poate fi iterat) folosind iteratorul construit pe baza operației *iterator* din interfața dicționarului.

```
Subalgoritmul tipărire(d) este:  
{pre: d este un dicționar}  
{post: se tipăresc elementele dicționarului}  
    iterator(d, i)          {se obține un iterator pe dicționarul d}  
    Cât timp valid(i) execută {cât timp iteratorul e valid}  
        element(i, e) {e este elementul curent referit de iterator}  
        @ tipărește e    {se tipărește elementul curent}  
        următor(i)      {iteratorul referă următorul element}  
    sf-cât  
sf-tipărire
```

1.6. Probleme propuse

1. Scrieți un program într-unul din limbajele de programare Python, C++, Java, C# care:
 - a. Definiște o clasă **B** având un atribut b de tip întreg și o metodă de tipărire care afișează atributul b la ieșirea standard.
 - b. Definiște o clasă **D** derivată din **B** având un atribut d de tip șir de caractere și de asemenea o metodă de tipărire pe ieșirea standard care va afișa atributul b din clasa de bază și atributul d .
 - c. Definiște o funcție care construiește o listă conținând: un obiect o_1 de tip **B** având b egal cu 8; un obiect o_2 de tip **D** având b egal cu 5 și d egal cu "D5"; un obiect o_3 de tip **B** având b egal cu -3; un obiect o_4 de tip **D** având b egal cu 9 și d egal cu "D9".
 - d. Definiște o funcție care primește o listă cu obiecte de tip **B** și returnează o listă doar cu obiectele care satisfac proprietatea: $b > 6$.
 - e. Pentru tipul de dată **listă** utilizat în program, scrieți specificațiile operațiilor folosite.

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

2. Scrieți un program într-unul din limbajele de programare Python, C++, Java, C# care:
 - a. Definiște o clasă **B** având un atribut b de tip întreg și o metodă de tipărire care afișează atributul b la ieșirea standard.
 - b. Definiște o clasă **D** derivată din **B** având un atribut d de tip șir de caractere și de asemenea o metodă de tipărire pe ieșirea standard care va afișa atributul b din clasa de bază și atributul d .
 - c. Definiște o funcție care construiește un dicționar conținând: un obiect o_1 de tip **B** având b egal cu 8; un obiect o_2 de tip **D** având b egal cu 5 și d egal cu "D5"; un obiect o_3 de tip **B** având b egal cu -3; un obiect o_4 de tip **D** având b egal cu 9 și d egal cu "D9". (*cheia* unui obiect din dicționar este valoarea b , iar *valoarea* asociată cheii este obiectul).
 - d. Definiște o funcție care primește un dicționar cu obiecte de tip **B** și verifică dacă în dicționar există un obiect care satisface proprietatea: $b > 6$.

- e. Pentru tipul de dată **dictionar** utilizat în program, scrieți specificațiile operațiilor folosite.

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

3. Subiectul va prezenta **o diagramă de clase** și o diagramă de **interacțiuni între obiecte** și 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 relaționale. Primele trei forme normale ale unei relații

2.1.1. Modelul relațional

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

Fie A_1, A_2, \dots, A_n o mulțime de atribute (coloane, constituanți, nume de date, etc.) și $D_i = \text{Dom}(A_i) \cup \{?\}$ domeniul valorilor posibile pentru atributul A_i , unde prin “?” s-a notat valoarea de “*nedefinit*” (*null*). Valoarea de *nedefinit* se folosește 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, șiruri de caractere, date calendaristice, etc.).

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

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

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

R	A₁	...	A_i	..	A_n
r₁	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 relației**, sau **tupluri**, sau **înregistrări**, care în general sunt distincte, și $a_{ij} \in D_j, j = 1, \dots, n, i = 1, \dots, m$. Deoarece modul în care se evidențiază elementele relației **R** de mai sus seamănă cu un tabel, relația se mai numește și **tabel**. Pentru a pune în evidență numele relației (tabelului) și lista atributelor vom nota această relație cu:

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

Modelul relațional al unei baze de date constă dintr-o **colecție de relații** ce variază în timp (conținutul relațiilor se poate schimba prin operații de adăugare, ștergere și actualizare).

O **bază de date relațională** constă din trei părți:

1. **Datele** (relații sau tabele, legături între tabele) și descrierea acestora;

2. **Reguli de integritate** (pentru a memora numai valori corecte în relații);
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

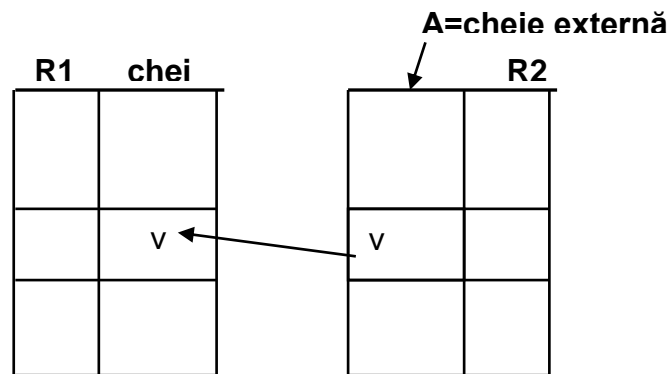
Pentru fiecare relație se poate preciza un atribut sau o colecție de atribute, din cadrul relației, numit **cheie**, cu rol de identificare a elementelor relației (cheia ia valori diferite pentru înregistrări diferite din relație, deci fiecare înregistrare se poate identifica prin valoarea cheii). Dacă 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 submulțime de atribute din cheie nu este cheie. Deoarece toate elementele relației sunt diferite, o cheie există totdeauna (în cel mai rău caz cheia este formată din toate atributele relației). Pentru exemplul 1 se poate alege NUME ca și cheie (atunci în baza de date nu pot exista doi studenți cu același nume), iar pentru exemplul 2 se poate alege grupul de atribute {AUTORI, TITLU, EDITURA, AN_APARITIE} ca și cheie, sau să se introducă un nou atribut (de exemplu COTA) pentru identificare.

Pentru anumite relații 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 existența a două elemente distincte într-o relație cu aceeași valoare pentru oricare cheie (principală sau secundară), deci precizarea unei chei constituie o **restricție** pentru baza de date.

Exemplul 3. ORAR [ZI, ORA, SALA, PROFESOR, CLASA, DISCIPLINA],
cu orarul pe o săptămână. Se pot alege ca și chei următoarele mulțimi de atribute:
{ZI, ORA, SALA}; {ZI, ORA, PROFESOR}; {ZI, ORA, CLASA}.

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

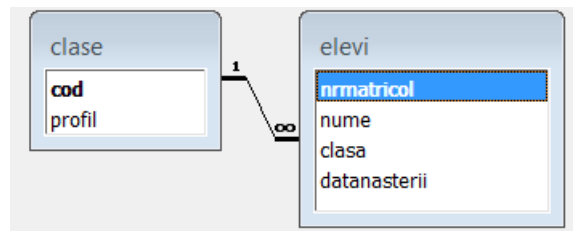
atributului **A** se caută printre valorile cheii din relația **R1**. Cele două relații **R1** și **R2** nu este obligatoriu să fie distincte.



Exemplu:

CLASE [cod, profil]

ELEVI [nrmatricol, nume, clasa, datanasterii].

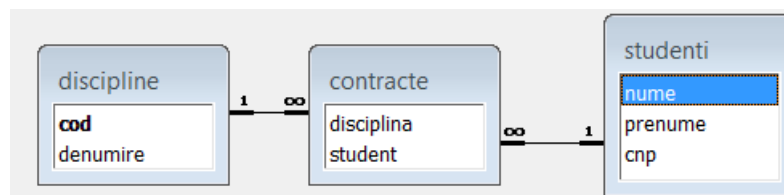


Legătura o putem stabili între relația **CLASE** (considerată ca părinte pentru legătură) și relația **ELEVI** (ca membru pentru legătură) prin egalitatea **CLASE.cod=ELEVI.clasa**. Unei anumite clase (memorată în relația **CLASE**), identificată printr-un cod, îi corespund toți elevii din clasa cu codul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

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

Cheia externă se poate folosi și pentru a memora **legături m:n** între entități.

Fie două entități: **discipline** și **studenți**. La o disciplină sunt "inscriși" mai mulți studenți, iar un student are asociate mai multe discipline. Varianta de memorare cuprinde o relație intermediară.



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

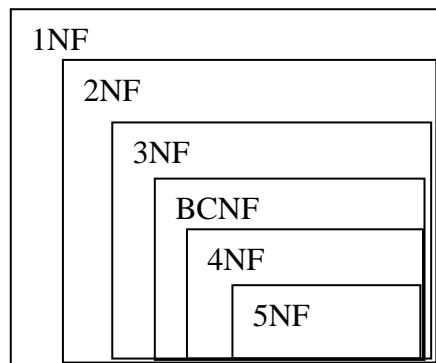
- **restricții asociate coloanei:**
 - Not Null - coloana nu poate să primească valori nedefinite
 - Primary Key - coloana curentă se definește cheia primară
 - Unique - valorile coloanei sunt unice

- Check(condiție) - se dă condiția pe care trebuie să o îndeplinească valorile coloanei (condiții 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
- **restricții asociate tabelului:**
 - Primary key(lista coloane) - definirea cheii primare pentru tabel
 - Unique(lista coloane) - valorile sunt unice pentru lista de coloane precizată
 - Check(condiție) - pentru a preciza condiția 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 definește cheia externă

2.1.2. Primele trei forme normale ale unei relații

În general anumite date se pot reprezenta în mai multe moduri prin relații (la modelul relațional). Pentru ca aceste date **să se poată prelucra cât mai simplu** (la o operație de actualizare a datelor să nu fie necesare teste suplimentare) este necesar ca relațiile în care se memorează datele să verifice anumite condiții (să aibă un anumit nivel de normalizare).

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



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

Definiție. Pentru descompunerea unei relații se folosește operatorul de **proiecție**. Fie $R[A_1, A_2, \dots, A_n]$ o relație și $\alpha = \{A_{i_1}, A_{i_2}, \dots, A_{i_p}\}$ o submulțime de atribute, $\alpha \subset \{A_1, A_2, \dots, A_n\}$. Prin **proiecția** relației **R** pe **α** se înțelege relația:

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

unde:

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

și toate elementele din R' sunt distincte.

Definiție. Pentru compunerea relațiilor se folosește operatorul de **join natural**. Fie $R[\alpha, \beta]$, $S[\beta, \gamma]$ două relații peste mulțimile de atribute α, β, γ , $\alpha \cap \gamma = \emptyset$. Prin **joinul natural** al relațiilor **R** și **S** se înțelege relația:

$$R * S[\alpha, \beta, \gamma] = \left\{ \left(\prod_{\alpha}(r), \prod_{\beta}(r), \prod_{\gamma}(s) \right) \mid r \in R, s \in S \text{ și } \prod_{\beta}(r) = \prod_{\gamma}(s) \right\}$$

O relație **R** se poate descompune în mai multe relații noi R_1, R_2, \dots, R_m . Această descompunere este **bună** dacă $R = R_1 * R_2 * \dots * R_m$, deci datele din **R** se pot obține din datele memorate în relațiile R_1, R_2, \dots, R_m și nu apar date noi prin aceste operații de compunere.

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

ContracteStudiu[Student, CadruDidactic, Disciplina],

și două noi relații obținute prin proiecția acestei relații: **SC**[Student, CadruDidactic] și **CD**[CadruDidactic, Disciplina]. Presupunem că pentru relația inițială avem următoarele valori:

R	Student	CadruDidactic	Disciplina
r ₁	s1	c1	d1
r ₂	s2	c2	d2
r ₃	s1	c2	d3

Folosind definiția proiecției se obțin următoarele valori pentru cele două relații obținute din **R** și pentru joinul natural al acestor relații:

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 relația **SC*CD** se obțin înregistrări suplimentare față de relația inițială, deci descompunerea sugerată **nu este bună**.

Observație. Prin **atribut simplu** vom înțelege un atribut oarecare din relație, iar prin **atribut compus** vom înțelege o mulțime de atribute (cel puțin două) din relație.

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

Exemplul 4. Fie relația:

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

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

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

Exemplul 5. Fie relația:

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

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

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

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

Exemplul 6. Relația STUDENT din exemplul 4 se descompune în următoarele două relații:

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

Exemplul 7. Relația CARTE din exemplul 5 se descompune în următoarele trei relații (în relația CARTE există două grupuri repetitive):

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

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

Definiție. O relație este de **prima formă normală (1NF)** dacă ea **nu conține grupuri** (de atribute) **repetitive**.

Sistemele de gestiune a bazelor de date relaționale permit descrierea numai a relațiilor ce se află în 1NF. Există și sisteme ce permit gestiunea relațiilor non-1NF (exemplu Oracle, unde o coloană poate fi un *obiect* sau o “*colecție*” de date, sau mai recent bazele de date NoSQL).

Următoarele forme normale ale unei relații utilizează o noțiune foarte importantă, și anume **dependența funcțională** dintre diverse submulțimi de atribute. Stabilirea dependențelor funcționale este o sarcină a administratorului bazei de date și depinde de semnificația (semantica) datelor ce se memorează în relație. Operațiile de actualizare a datelor din relație (înserare, ștergere, modificare) nu trebuie să modifice dependențele funcționale (dacă pentru relație există astfel de dependențe).

Definiție. Fie $R[A_1, A_2, \dots, A_n]$ o relație și $\alpha, \beta \subset \{A_1, A_2, \dots, A_n\}$ două submulțimi de atribute. Atributul (simplu sau compus) β este **dependent funcțional** de atributul α (simplu sau compus), notație: $\alpha \rightarrow \beta$, dacă și numai dacă fiecare valoare a lui α din \mathbf{R} are asociată o **valoare precisă și unică** pentru β (această asociere este valabilă “tot timpul” existenței relației \mathbf{R}). O valoare oarecare a lui α poate să apară în mai multe linii ale lui \mathbf{R} și atunci fiecare dintre aceste linii conține aceeași valoare pentru atributul β , deci:

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

Valoarea α din implicația (dependența) $\alpha \rightarrow \beta$ se numește **determinant**, iar β este **determinat**.

Observație. Dependența funcțională se poate folosi ca o **proprietate (restricție)** pe care baza de date trebuie să o îndeplinească pe perioada existenței acesteia: se adaugă, elimină, modifică elemente în relație numai dacă dependența funcțională este verificată.

Existența unei dependențe funcționale într-o relație înseamnă că anumite asocieri de valori se memorează de mai multe ori, deci apare o **redundanță**. Pentru exemplificarea unor probleme care apar vom lua relația următoare, care memorează rezultatele la examene pentru studenți:

Exemplul 8. EXAMEN [NumeStudent, Disciplina, 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 restricția (dependența) $\{Disciplina\} \rightarrow \{CadruDidactic\}$.

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	Frișan Florin	Matematică	10	Rus Teodor

Dacă păstrăm o astfel de dependență funcțională, atunci pot apare următoarele probleme:

- **Risipă de spațiu:** aceleași asocieri se memorează de mai multe ori. Legătura dintre disciplina de *Matematică* și profesorul *Rus Teodor* este memorată de trei ori, iar dintre disciplina *Istorie* și 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 ști câte astfel de asocieri există), altfel baza

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

- **Anomalii la inserare:** la adăugarea unei înregistrări trebuie să se cunoască valorile atributelor, nu se pot folosi valori nedefinite pentru atributele implicate în dependențele funcționale.
- **Anomalii la ștergere:** la ștergerea unor înregistrări se pot șterge și asocieri (între valori) ce nu se pot refăce. De exemplu, dacă se șterg înregistrările 2 și 3, atunci asocierea dintre *Disciplina* și *CadruDidactic* se pierde.

Anomaliile de mai sus apar datorită existenței unei dependențe funcționale între mulțimi de atribute. Pentru a elimina situațiile amintite trebuie ca aceste dependențe (asocieri) de valori să se păstreze într-o relație separată. Pentru aceasta este necesar ca relația inițială să se descompună, fără ca prin descompunere să se piardă date sau să se introducă date noi prin compunerea de relații (trebuie ca descompunerea "să fie bună"). O astfel de descompunere se face în momentul proiectării bazei de date, când se pot stabili dependențele funcționale.

Observații. Se pot demonstra ușor următoarele proprietăți simple pentru dependențele funcționale:

1. Dacă C este o cheie pentru $R[A_1, A_2, \dots, A_n]$, atunci $C \rightarrow \beta, \forall \beta \subset \{A_1, A_2, \dots, A_n\}$.
2. Dacă $\beta \subseteq \alpha$, atunci $\alpha \rightarrow \beta$, numită **dependența funcțională trivială** sau **reflexivitatea**.

$$\Pi_{\alpha}(r_1) = \Pi_{\alpha}(r_2) \Rightarrow_{\beta \subset \alpha} \Pi_{\beta}(r_1) = \Pi_{\beta}(r_2) \Rightarrow \alpha \rightarrow \beta$$
3. Dacă $\alpha \rightarrow \beta$, atunci $\gamma \rightarrow \beta, \forall \gamma$ cu $\alpha \subset \gamma$.

$$\Pi_{\gamma}(r_1) = \Pi_{\gamma}(r_2) \Rightarrow_{\alpha \subset \gamma} \Pi_{\alpha}(r_1) = \Pi_{\alpha}(r_2) \Rightarrow_{\alpha \rightarrow \beta} \Pi_{\beta}(r_1) = \Pi_{\beta}(r_2) \Rightarrow \gamma \rightarrow \beta$$
4. Dacă $\alpha \rightarrow \beta$ și $\beta \rightarrow \gamma$, atunci $\alpha \rightarrow \gamma$, care este proprietatea de **tranzitivitate** a dependenței funcționale.

$$\Pi_{\alpha}(r_1) = \Pi_{\alpha}(r_2) \Rightarrow_{\alpha \rightarrow \beta} \Pi_{\beta}(r_1) = \Pi_{\beta}(r_2) \Rightarrow_{\beta \rightarrow \gamma} \Pi_{\gamma}(r_1) = \Pi_{\gamma}(r_2) \Rightarrow \alpha \rightarrow \gamma$$
5. Dacă $\alpha \rightarrow \beta$ și $\gamma \subset A$, atunci $\alpha\gamma \rightarrow \beta\gamma$, unde $\alpha\gamma = \alpha \cup \gamma$.

$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left\{ \begin{array}{l} \Pi_{\alpha}(r_1) = \Pi_{\alpha}(r_2) \Rightarrow \Pi_{\beta}(r_1) = \Pi_{\beta}(r_2) \\ \Pi_{\gamma}(r_1) = \Pi_{\gamma}(r_2) \end{array} \right. \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

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

Definiție. Fie $R[A_1, A_2, \dots, A_n]$ și $\alpha, \beta \subset \{A_1, A_2, \dots, A_n\}$. Atributul β este **complet dependent funcțional** de α dacă β este dependent funcțional de α (deci $\alpha \rightarrow \beta$) și nu este dependent funcțional de nici o submulțime de atribute din α ($\forall \gamma \subset \alpha, \delta \rightarrow \beta$ nu este adevărat).

Observație. Dacă atributul β **nu** este **complet dependent funcțional** de α (deci este dependent de o submulțime a lui α), atunci α este un **atribut compus**.

Definiție. O relație este de a **două 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 funcțional de oricare cheie a relației**.

Observație. Dacă o relație este de prima formă normală (1NF) și nu este de a doua formă normală (2NF), atunci **are o cheie compusă** (dacă o relație nu este de a doua formă normală, atunci există o dependență funcțională $\alpha \rightarrow \beta$ cu α atribut inclus într-o cheie).

Pentru a preciza modul de **descompunere** pentru cazul general, fie $R[A_1, A_2, \dots, A_n]$ o relație și $C \subset A = \{A_1, A_2, \dots, A_n\}$ o cheie. Presupunem că există $\beta \subset A$, $\beta \cap C = \emptyset$ (β este un atribut necheie), β dependent funcțional de $\alpha \subset C$ (β este complet dependent funcțional de o submulțime strictă de atribute din cheie). Dependența $\alpha \rightarrow \beta$ se poate elimina dacă relația **R** se descompune în următoarele două relații:

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

Vom analiza relația din exemplul 8:

EXAMEN [NumeStudent, Disciplina, Nota, CadruDidactic],

unde cheia este $\{NumeStudent, Disciplina\}$ și există dependența funcțională (restricția) $\{Disciplina\} \rightarrow \{CadruDidactic\}$. De aici deducem că atributul *CadruDidactic* nu este complet dependent funcțional de o cheie, deci relația EXAMEN nu este de a doua formă normală. Eliminarea acestei dependențe funcționale se poate face prin descompunerea relației în următoarele relații:

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

Exemplul 9. Presupunem că pentru memorarea contractelor de studiu se folosește relația:

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

Cheia relației este $\{CNP, CodDisciplina\}$. În relație mai există două dependențe funcționale: $\{CNP\} \rightarrow \{Nume, Prenume\}$ și $\{CodDisciplina\} \rightarrow \{DenumireDisciplina\}$. Pentru eliminarea acestor dependențe se descompune relația în următoarele:

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

Pentru a treia formă normală este necesară noțiunea de **dependență tranzitivă**.

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

Definiție. O relație este de a **treia formă normală (3NF)** dacă este 2NF și **orice atribut neprim nu este tranzitiv dependent de oricare cheie a relației**.

Dacă **C** este o cheie și β un atribut tranzitiv dependent de cheie, atunci există un atribut α care verifică: $C \rightarrow \alpha$ (dependență care este verificată totdeauna) și $\alpha \rightarrow \beta$. Deoarece relația este 2NF, obținem că β este complet dependent de **C**, deci $\alpha \not\subset C$. De aici deducem că o relație ce este 2NF și nu este 3NF are o dependență $\alpha \rightarrow \beta$, iar α este atribut neprim. Această dependență se poate elimina prin descompunerea relației R în mod asemănător ca la eliminarea dependențelor de la 2NF.

Exemplul 10. Rezultatele obținute de absolvenți la lucrarea de licență sunt trecute în relația:

LUCRARI_LICENTA [NumeAbsolvent, Nota, CadruDidIndr, Departament].

Aici se memorează numele cadrului didactic îndrumător și denumirea departamentului la care se află acesta. Deoarece se introduc date despre absolvenți, câte o înregistrare pentru un absolvent, putem să stabilim că *NumeAbsolvent* este cheia relației. Din semnificația atributelor incluse în relație se observă următoarea dependență funcțională:

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

Din existența acestei dependențe funcționale se deduce că relația **nu este de 3NF**. Pentru a elimina dependența funcțională, relația se poate descompune în următoarele două relații:

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

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

ADRESE [CNP, Nume, Prenume, CodPostal, LocalitateDomiciliu, Strada, Nr].

Cheia relației este {CNP}. Deoarece la unele localități codul poștal se stabilește la nivel de stradă, sau chiar poțiuni de stradă, există dependența funcțională:

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

Deoarece există această dependență funcțională, deducem că relația ADRESE nu este de a treia formă normală, deci este necesară descompunerea ei.

Exemplul 12. Să considerăm următoarea relație care memorează o eventuală planificare a studenților pentru examene:

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

cu următoarele restricții (cerințe care trebuie respectate) și care se transpun în definiția de chei sau de dependențe funcționale:

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 și alte examene, dar la alte ore, deci există următoarea dependență funcțională:

$\{Cadru_did, Data\} \rightarrow \{Sala\}$

Toate attributele din această relație apar în cel puțin o cheie, deci nu există attribute neprime. Având în vedere definiția formelor normale precizate până acum, putem spune că relația **este în 3NF**. Pentru a elimina și dependențele funcționale de tipul celor pe care le avem în exemplul de mai sus s-a introdus o nouă formă normală:

Definiție. O relație este în **3NF Boyce-Codd**, sau **BCNF**, dacă orice determinant (pentru o dependență funcțională) este cheie, deci nu există dependențe funcționale $\alpha \rightarrow \beta$ astfel încât α să nu fie cheie.

Pentru a elimina dependența funcțională amintită mai sus trebuie să facem următoarea descompunere pentru relația **PLANIFICARE_EX**:

PLANIFICARE_EX [Data, Cadru_did, Ora, Student],

REPARTIZARE_SALI [Cadru_did, Data, Sala].

După această descompunere nu mai există dependențe funcționale, deci relațiile sunt de tipul BCNF, dar a dispărut cheia asociată restricției precizate la punctul 3 de mai sus: {Sala, Data, Ora}. Dacă se mai dorește păstrată o astfel de restricție, atunci ea trebuie verificată altfel (de exemplu, prin program).

2.2. Interogarea BD cu operatori din algebra relațională

Pentru a **explica** limbajul de interogare (cererea de date) bazat pe algebra relațiilor vom preciza la început **tipurile de condiții** ce pot apare în cadrul diferiților operatori relaționali.

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

nume atribut *operator_relațional* **valoare**

2. O relație cu o singură coloană poate fi considerată ca o mulțime. Următoarea condiție testează dacă o anumită valoare aparține sau nu unei mulțimi:

$$\text{nume_atribut} \left\{ \begin{array}{l} IS\ IN \\ IS\ NOT\ IN \end{array} \right\} \text{relație_cu_o_coloană}$$

3. Două relații (considerate ca mulțimi de înregistrări) se pot compara prin operațiile de egalitate, diferit, incluziune, neincluziune. Intre două relații cu același număr de coloane și cu aceleași tipuri de date pentru coloane (deci între două mulțimi comparabile) putem avea condiții de tipul următor:

$$\text{relație} \left\{ \begin{array}{l} IS\ IN \\ IS\ NOT\ IN \\ = \\ \diamond \end{array} \right\} \text{relație}$$

4. Tot *condiție* este și oricare din construcțiile următoare:

(*condiție*)
NOT *condiție*
condiție1 AND *condiție2*
condiție1 OR *condiție2*

unde *condiție*, *condiție1*, *condiție2* sunt condiții de tipurile 1-4.

În primul tip de condiție apare construcția '*valoare*', care poate fi una din tipurile următoare. Pentru fiecare construcție se ia în *valoare* o anumită relație 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 relații curente care conțin câte un atribut cu acest nume), atunci se va face o calificare a atributului cu numele relației sub forma: **relație.atribut**.

- **expresie** - dacă se evaluează expresia, iar dacă apar și denumiri de atribute, atunci acestea se iau dintr-o înregistrare curentă.
- **COUNT(*) FROM relație** - precizează numărul de înregistrări din relația specificată.

- $\left. \begin{array}{l} COUNT \\ SUM \\ AVG \\ MAX \\ MIN \end{array} \right\} ([DISTINCT] \text{nume_atribut})$ - care determină o valoare plecând de la toate

înregistrările din relația curentă. La determinarea acestei valori se iau toate valorile atributului precizat ca argument (din toate înregistrările), sau numai valorile distincte, după cum lipsește 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 relaționale**.

- **Selectia** (sau proiecția orizontală) a unei relații **R** - determină o nouă relație ce are aceeași schemă cu a relației **R**. Din relația **R** se iau numai înregistrările care îndeplinesc o condiție **c**. Notăția pentru acest operator este: $\sigma_c(R)$.
- **Proiecția** (sau proiecția verticală) - determină o relație nouă ce are atributele precizate printr-o mulțime α de atribute. Din fiecare înregistrare a unei relații **R** se determină numai valorile atributelor incluse în mulțimea α . Mulțimea α de atribute se poate extinde la o **mulțime de expresii** (în loc de o mulțime de atribute), care precizează coloanele relației care se construiește. Notăția pentru acest operator este: $\prod_{\alpha}(R)$.
- **Produsul cartezian** a două relații: **R₁ × R₂** - care determină o relație nouă ce are ca atribute concatenarea atributelor din cele două relații, iar fiecare înregistrare din **R₁** se concatenează cu fiecare înregistrare din **R₂**.
- **Reuniunea, diferența și intersecția** a două relații: **R₁ ∪ R₂, R₁ - R₂, R₁ ∩ R₂**. Cele două relații trebuie să aibă *aceeași schemă*.
- Există mai mulți operatori **join**.

Joinul condițional sau **theta join**, notat prin **R₁ ⊗_θ R₂** - care determină acele înregistrări din produsul cartezian al celor două relații care îndeplinesc o anumită condiție. Din definiție se observă că avem: **R₁ ⊗_θ R₂ = σ_θ(R₁ × R₂)**.

Joinul natural, notat prin **R₁ * R₂**, care determină o relație nouă ce are ca atribute reuniunea atributelor din cele două relații, iar înregistrările se obțin din toate perechile de înregistrări ale celor două relații care au aceleași valori pentru atributele comune. Dacă cele două relații au schemele **R₁[α], R₂[β]**, și $\alpha \cap \beta = \{A_1, A_2, \dots, A_n\}$, atunci joinul natural se poate calcula prin construcția următoare:

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

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

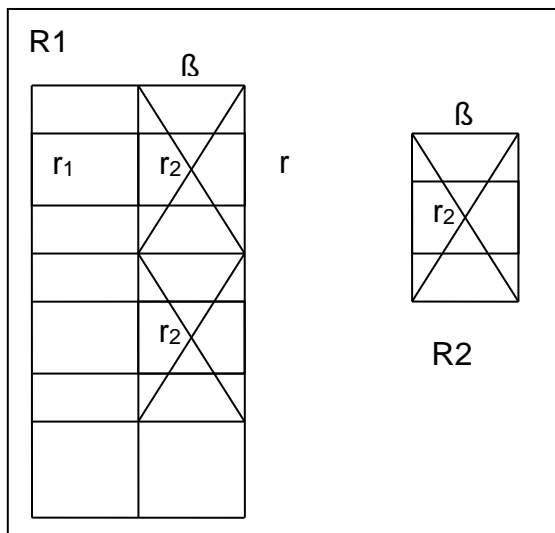
Joinul extern dreapta, notat prin $R_1 \triangleleft_c R_2$, se obține asemănător ca joinul extern stânga, dar la înregistrările din $R_1 \otimes_c R_2$ se adaugă înregistrările din R_2 care nu s-au folosit la acest join condițional combinate cu valoarea *null* pentru toate atributele corespunzătoare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

- **Câtul** pleacă de la două relații $R_1[\alpha]$, $R_2[\beta]$,

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

1. $\prod_{\alpha - \beta}(r_1) = r$;
2. $\prod_{\beta}(r_1) = r_2$.

Semnificația relației cât se vede și din figura alăturată. O înregistrare r_1 aparține câtului dacă în relația R_1 apar toate concatenările dintre această înregistrare și fiecare înregistrare din R_2 .



O problemă importantă legată de operatorii descriși mai sus constă în determinarea unei **submulțimi independente** de operatori. O mulțime M de operatori este **independentă** dacă eliminând un operator oarecare op din M se diminuează puterea mulțimii, adică va exista o relație obținută cu operatori din M și care nu se poate obține cu operatori din mulțimea $M - \{op\}$.

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

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

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

- Fie $R_1[\alpha]$, $R_2[\beta]$, și $R_3[\beta] = (\text{null}, \dots, \text{null})$, $R_4[\alpha] = (\text{null}, \dots, \text{null})$.

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

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

- Dacă $R_1[\alpha], R_2[\beta]$, 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 îndeplinește condițiile: $\prod_{\alpha-\beta}(r_1) = r$ și $\prod_{\beta}(r_1) = r_2$.

De aici deducem că r este din $\prod_{\alpha-\beta}(R_1)$. În $(\prod_{\alpha-\beta}(R_1)) \times R_2$ sunt toate elementele ce au o parte în $\prod_{\alpha-\beta}(R_1)$ și a doua parte în R_2 . Din relația astfel obținută vom elimina pe R_1 și rămân acele elemente ce au o parte în $\prod_{\alpha-\beta}(R_1)$ și nu au cealaltă parte în $\prod_{\beta}(R_1)$. De aici obținem:

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

La lista de operatori relaționali amintiți mai sus se pot aminti câteva instrucțiuni utile la rezolvarea unor probleme:

- **Atribuirea:** unei variabile (relații) R îi vom atribui o relație dată printr-o expresie construită cu operatorii de mai sus. În instrucțiune se poate preciza, pentru R , și denumirea coloanelor.

R[lista] := expresie

- **Eliminarea duplicărilor** unei relații: $\delta(R)$
- **Sortarea** înregistrărilor dintr-o relație: $s_{\{lista\}}(R)$
- **Gruparea:** $\gamma_{\{lista1\} \text{ group by } \{lista2\}}(R)$, care este o extensie pentru proiecție. Înregistrările din R sunt grupate după coloanele din $lista2$, iar pentru un grup de înregistrări cu aceleași valori pentru $lista2$ se evaluează $lista1$ (unde pot apare *funcții de grupare*).

2.3. Interogarea bazelor de date relaționale cu SQL

Pentru gestiunea bazelor de date relaționale 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 relațional
- 1974 - la IBM (din San Jose) se definește limbajul SEQUEL (Structured English Query Language)
- 1975 - se definește limbajul SQUARE (Specifying Queries as Relational Expressions).
- 1976 - la IBM se definește 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 și se obține SQL2 sau SQL92

- 1999 - se completează SQL cu posibilități de gestiune orientate obiect, rezultând SQL3 (sau SQL1999)
- 2003 - se adaugă noi tipuri de date și noi funcții, rezultând SQL2003.

Comanda SELECT este folosită pentru interogarea bazelor de date (obținerea de informații). Această comandă este cea mai complexă din cadrul sistemelor ce conțin comenzi SQL. Comanda permite obținerea de date din mai multe surse de date. Ea are, printre altele, funcțiile de selecție, proiectie, produs cartezian, join și reuniune, intersecție și diferență din limbajul de interogare a bazelor de date relaționale bazat pe algebra relațiilor. Sintaxa comenzii este dată în continuare.

$$\text{SELECT } \left\{ \begin{array}{l} \text{ALL} \\ \text{DISTINCT} \\ \text{TOP } n[\text{PERCENT}] \end{array} \right\} \left\{ \begin{array}{l} * \\ \text{exp} [\text{AS } \text{câmp}] [\text{exp} [\text{AS } \text{câmp}]] \dots \end{array} \right\}$$

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

[**WHERE** condiție]

[**GROUP BY** lista_câmpuri [**HAVING** condiție]]

$$\left[\left\{ \begin{array}{l} \text{UNION} [\text{ALL}] \\ \text{INTERSECT} \\ \text{EXCEPT} \end{array} \right\} \text{comanda_SELECT} \right]$$

$$\left[\text{ORDER BY } \left\{ \begin{array}{l} \text{câmp} \\ \text{nrcâmp} \end{array} \right\} \left[\left\{ \begin{array}{l} \text{ASC} \\ \text{DESC} \end{array} \right\} \right] \right], \text{ORDER BY } \left\{ \begin{array}{l} \text{câmp} \\ \text{nrcâmp} \end{array} \right\} \left[\left\{ \begin{array}{l} \text{ASC} \\ \text{DESC} \end{array} \right\} \right] \dots \right]$$

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 definește un "**alias**", atunci calificarea se face numai cu el (nu se va mai face cu numele tabelului).

O construcție numită **sursa** poate fi:

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

O **condiție_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 comparației trebuie să aparțină la tabele diferite.

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

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

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

$$\text{Sursa1} \left\{ \begin{array}{l} \text{INNER} \\ \text{LEFT [OUTER]} \\ \text{RIGHT[OUTER]} \\ \text{FULL[OUTER]} \end{array} \right\} \text{JOIN Sursa2 ON condiție}$$

Joinul condițional, din algebra relațională, notat prin $\text{Sursa1} \otimes_c \text{Sursa2}$, este precizat prin **Sursa1 INNER JOIN sursa2 ON condiție**, și determină acele înregistrări din produsul cartezian al celor două surse care îndeplinesc condiția din **ON**.

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

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

Joinul extern total, precizat prin **Sursa1 FULL [OUTER] JOIN sursa2 ON condiție**, se obține prin reuniunea rezultatelor obținute de joinul extern stânga și 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**, și celelalte tabele este indicat să existe anumite **legături** (stabilite prin **condiții**). Plecând de la fiecare înregistrare a tabelului principal se determină înregistrările din celelalte tabele asociate prin astfel de legături (deci înregistrările ce verifică o condiție). Dacă legătura (condiția) nu se stabilește, atunci se consideră că ea asociază toate înregistrările din celelalte tabele pentru fiecare înregistrare a tabelului principal (se consideră că valoarea condiției este **true** atunci când ea lipsește). Această condiție de legătură dintre sursele de date se precizează prin:

FROM sursa1[, sursa2] ... WHERE condiție_legătură

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

În **tabel_rezultat** se pot păstra toate înregistrările obținute din sursele de date, sau se poate face o **filtrare** prin utilizarea unei condiții de filtrare. Aceasta condiție de filtrare va fi trecută în clauza **WHERE** în continuarea condiției de legătură. Cu o condiție de filtrare condiția din **WHERE** este de forma:

WHERE condiție_filtrare

WHERE condiție_legătură AND condiție_filtrare

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

Condițiile 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* și *valmax*) sau nu este cuprinsă între aceste valori (apare **NOT**)

- **câmp (NOT) LIKE șablon**

După **LIKE** apare un *șablon* (ca un șir de caractere) ce precizează o mulțime de valori. În funcție de sistemul de gestiune folosit, există un caracter în șablon ce precizează locul unui singur caracter necunoscut în câmp, sau un caracter în șablon ce precizează un șir neprecizat de caractere în câmp.

- **expresie[NOT] IN** $\left\{ \begin{array}{l} \text{valoare [valoare] ...} \\ \text{(subselectie)} \end{array} \right\}$

Se verifică dacă valoarea expresiei apare (sau nu - cu **NOT**) într-o listă de valori sau într-o subselectie. O **subselectie** este o sursă de date generată cu comanda **SELECT** și care are numai un singur câmp - cu valori de același tip cu valorile expresiei. Această condiție corespunde testului de "**apartenență**" al unui element la o mulțime.

- **câmp operator_relational** $\left\{ \begin{array}{l} \text{ALL} \\ \text{ANY} \\ \text{SOME} \end{array} \right\}$ (subselectie)

Valorile câmpului din stânga operatorului relațional și valorile singurului câmp din subselectie trebuie să fie de același tip. Se obține valoarea *adevărat* pentru condiție dacă valoarea din partea stângă este în relația dată de operator pentru:

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

Condiții echivalente:

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

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

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

Cu **EXISTS** se obține valoarea *adevărat* dacă în subselectie există cel puțin o înregistrare, și *fals* dacă subselectia este vidă. Prezența lui **NOT** inversează valorile de adevăr.

- O **condiție de filtrare** poate fi:
 - condiție elementară
 - (condiție)
 - **not** condiție
 - condiție1 **and** condiție2
 - condiție1 **or** condiție2

O condiție elementară poate avea una din valorile: **true**, **false**, **null**. Valoarea null se obține dacă unul din operandii utilizați 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 aceeași denumire în tabele diferite se pot califica prin numele sau alias-ul tabelului sursă. Numele câmpului sau expresiei din tabelul rezultat este stabilit automat de sistem (în funcție de expresia ce-l generează), sau se poate preciza prin clauza **AS** ce urmează expresiei (sau câmpului). În acest fel putem construi valori pentru o nouă înregistrare într-un **tabel final**.

Expresiile se precizează cu ajutorul operandilor (câmpuri, rezultatul unor funcții) și a operatorilor corespunzători tipurilor de operandi.

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

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

Înregistrările 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 poziția lor (numărul câmpului) în lista de câmpuri din comanda **SELECT** (precizarea prin poziție este obligatorie atunci când se dorește 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 înlocuiește 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) crescător după valorile câmpurilor din **GROUP BY**. Înregistrările consecutive din fișierul astfel sortat, ce au aceeași valoare pentru toate câmpurile din **GROUP BY**, se înlocuiesc cu o singură înregistrare. Prezența unei astfel de înregistrări poate fi condiționată de valoarea *adevărat* pentru o condiție ce se trece în clauza **HAVING**.

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

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

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

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

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

Această funcție 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**).

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

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.

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

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

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

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

Două tabele cu același număr de câmpuri (coloane) și cu același tip pentru valorile câmpurilor aflate pe aceleași poziții se pot **reuni** într-un singur tabel obținut cu ajutorul operatorului **UNION**. Din tabelul rezultat obținut se pot păstra toate înregistrările (apare **ALL**) sau numai cele distincte (nu apare **ALL**). Clauza **ORDER BY** poate apare numai pentru ultima selecție. Nu se pot combina subselecții prin clauza **UNION**.

Între două rezultate (mulțimi de înregistrări) obținute cu instrucțiuni **SELECT** se poate folosi operatorul **INTERSEC** sau **EXCEPT** (sau **MINUS**).

Clauzele din instrucțiunea **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**. Definierea este:

CREATE VIEW nume_view AS comanda_SELECT

2.4. Probleme propuse

I.

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

- **activități:** cod activitate, descriere, tip activitate;
- **angajați:** cod angajat, nume, listă activități, 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 activități, iar la o activitate pot să participe mai mulți angajați;

Justificați că tabelele obținute sunt în 3NF.

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

b1. Numele angajaților care lucrează la cel puțin o activitate de tipul "*Proiectare*" și nu lucrează la nici o activitate de tipul "*Testare*".

b2. Numele angajaților care sunt liderii unei echipe cu cel puțin 10 angajați.

II.

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

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

Justificați că tabelele sunt în 3NF.

b. Pentru baza de date de la punctul a, folosind algebra relațională și instrucțiuni SELECT-SQL, se cer studenții (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 relațională, cu tabele în 3NF, ce gestionează următoarele informații despre absolvenții înscriși pentru examnul de licență: Nr.matricol, Cod și denumire secție absolvită, Titlul lucrării, Cod și nume cadru didactic îndrumător, Cod și denumire departament de care aparține cadrul didactic îndrumător, Lista de resurse soft necesare pentru susținerea lucrării (ex. C#.Net, C++, etc.), Lista de resurse hard necesare pentru susținerea lucrării (Ram 8Gb, DVD Reader, etc.). Justificați că tabelele sunt în 3NF.

b. Pentru baza de date de la punctul a, folosind algebra relațională și instrucțiuni SELECT-SQL cel puțin o dată fiecare, se cer următoarele liste (explicați modul de obținere a listelor):

- i. Absolvenții (nume, titlu lucrare, nume cadru didactic) pentru care cadrul didactic îndrumător aparține de un departament dat prin denumire.
- ii. Pentru departament se cere numărul de absolvenți care au cadrul didactic îndrumător de la acest departament.
- iii. Cadrele didactice care nu au îndrumat absolvenți la lucrarea de licență.
- iv. Numele absolvenților care au nevoie de următoarele două resurse soft: Oracle și C#.

IV.

Dați exemplul de instanță a unei relații $R(ABCD)$ unde dependența funcțională $ABC \rightarrow D$ nu este respectată.

V.

Se dau următoarele instanțe a două relații R și S :

	A	B	C
R:	1	2	2
	3	4	5

	D	E
S:	1	2
	3	6
	3	5

Care este rezultatul execuției următoarei interogări:

$$S - \pi_{A,C}(R)?$$

VI.

Fie relația *Persoane*(Cod, Nume, DataNastere, Oras, Profesie), unde Cod este cheie primară, și interogarea. Descrieți în cuvinte rezultatul obținut în urma execuției interogării:

$$\pi_{Oras} (\sigma_{Profesie='Programator'} (Persoane)).$$

și scrieți în SQL o interogare ce returnează același rezultat.

VII.

Rezultatul interogărilor Q_1 și Q_2 va fi dat de colecția de înregistrări returnată de execuția comenzii `SELECT * FROM R`. Vom presupune că structura relației R este $R(a, b)$.

Q_1 : `UPDATE R SET b = 10 WHERE a = 20;`
`SELECT * FROM R;`

Q_2 : `DELETE FROM R WHERE a = 20;`
`INSERT INTO R VALUES (20, 10);`
`SELECT * FROM R;`

Determinați care dintre următoarele variante este adevărată oricare ar fi conținutul tabelului R și explicați de ce.

- Q_1 și Q_2 produc același răspuns
- Răspunsul lui Q_1 este întotdeauna conținut în răspunsul lui Q_2
- Răspunsul lui Q_2 este întotdeauna conținut în răspunsul lui Q_1
- Q_1 și Q_2 produc răspunsuri diferite.

VIII.

Se dă mai jos instanța unei relații cu schema $S[FK_1, FK_2, A, B, C, D, E]$ și cheia $\{FK_1, FK_2\}$. Răspundeți întrebărilor următoare:

FK1	FK2	A	B	C	D	E
1	1	a1	b1	c1	7	2
1	2	a_	b3	c1	5	2
1	3	a2	b1	c2	null	2
2	1	a3	b3	c2	null	100
2	2	a3	b3	c3	null	100

A. Câte înregistrări va furniza interogarea:

```
SELECT *
FROM S
WHERE A LIKE 'a_'
```

- a. 5
- b. 4
- c. 0
- d. 1
- e. nicio variantă de mai sus nu este corectă

B. Cât este diferența între cardinalitatea rezultatului primei interogări și cardinalitatea rezultatului celei de a doua interogări:

```
SELECT FK2, FK1, COUNT(DISTINCT B)
FROM S
GROUP BY FK2, FK1
HAVING FK1 = 1
```

```
SELECT FK2, FK1, COUNT(C)
FROM S
GROUP BY FK2, FK1
HAVING FK1 = 2
```

- a. 1
- b. 0
- c. -1
- d. -2
- e. nicio variantă de mai sus nu este corectă

C. Care dintre propozițiile de mai jos este adevărată:

- a. cel puțin una din dependențele $\{A\} \rightarrow \{B\}$, $\{FK_1, FK_2\} \rightarrow \{A, B\}$, $\{FK_1\} \rightarrow \{A\}$ nu este satisfăcută de datele din relație;
- b. pe baza datelor din relație, putem afirma cu certitudine că cel puțin una din dependențele $\{A\} \rightarrow \{B\}$, $\{FK_1\} \rightarrow \{A, B\}$, $\{FK_1\} \rightarrow \{A\}$ este specificată la nivelul schemei S;
- c. cel puțin două din dependențele $\{FK_2\} \rightarrow \{A, B\}$, $\{A\} \rightarrow \{E\}$, $\{A, B\} \rightarrow \{E\}$, $\{B\} \rightarrow \{C, E\}$ nu sunt satisfăcute de datele din relație;
- d. pe baza datelor din relație, putem afirma cu certitudine că cel puțin două din dependențele $\{FK_2\} \rightarrow \{A, B\}$, $\{A\} \rightarrow \{E\}$, $\{A, B\} \rightarrow \{E\}$, $\{B\} \rightarrow \{C, E\}$ sunt specificate la nivelul schemei S;
- e. nicio variantă de mai sus nu este corectă

D. Câte înregistrări va furniza interogarea::

```
SELECT *
FROM S
WHERE B = 'b1' OR D = 5
```

- a. 2
- b. 3
- c. 1
- d. 5
- e. nicio variantă de mai sus nu este corectă

3. Sisteme de operare

3.1. Structura sistemelor de fişiere Unix

3.1.1. Structura Internă a Discului UNIX

3.1.1.1. Partiții și Blocuri

Un sistem de fişiere Unix este găzduit fie pe un periferic oarecare (hard-disc, CD, dischetă etc.), fie pe o partiție a unui hard-disc. Partiționarea unui hard-disc este o operație (relativ) independentă de sistemul de operare ce va fi găzduit în partiția respectivă. De aceea, atât partițiilor, cât și suporturilor fizice reale le vom spune generic, discuri Unix.

Blocul 0	- bloc de boot
Blocul 1	- Superbloc
Blocul 2	- inod
- - - - -	- - - - -
Blocul n	- inod
Blocul n+1	zona fişier
- - - - -	- - - - -
Blocul n+m	zona fişier

Structura unui disc UNIX

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

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

Blocul 1 este numit și superbloc. În el sunt trecute o serie de informații prin care se definește sistemul de fişiere de pe disc. Printre aceste informații amintim:

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

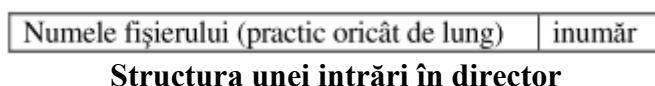
Blocurile 2 la n, unde n este o constantă a formatării discului. Un inod (sau i-nod) este numele, în terminologia Unix, a descriptorului unui fişier. Inodurile sunt memorate pe disc sub forma unei liste (numită-listă). Numărul de ordine al unui inod în cadrul i-listei se

reprezintă pe doi octeți și se numește i-număr. Acest i-număr constituie legătura dintre fișier și programele utilizator.

Partea cea mai mare a discului este rezervată zonei fișierelor. Alocarea spațiului pentru fișiere se face printr-o variantă elegantă de indexare. Informațiile de plecare pentru alocare sunt fixate în inoduri.

3.1.1.2. Directori și I-noduri

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



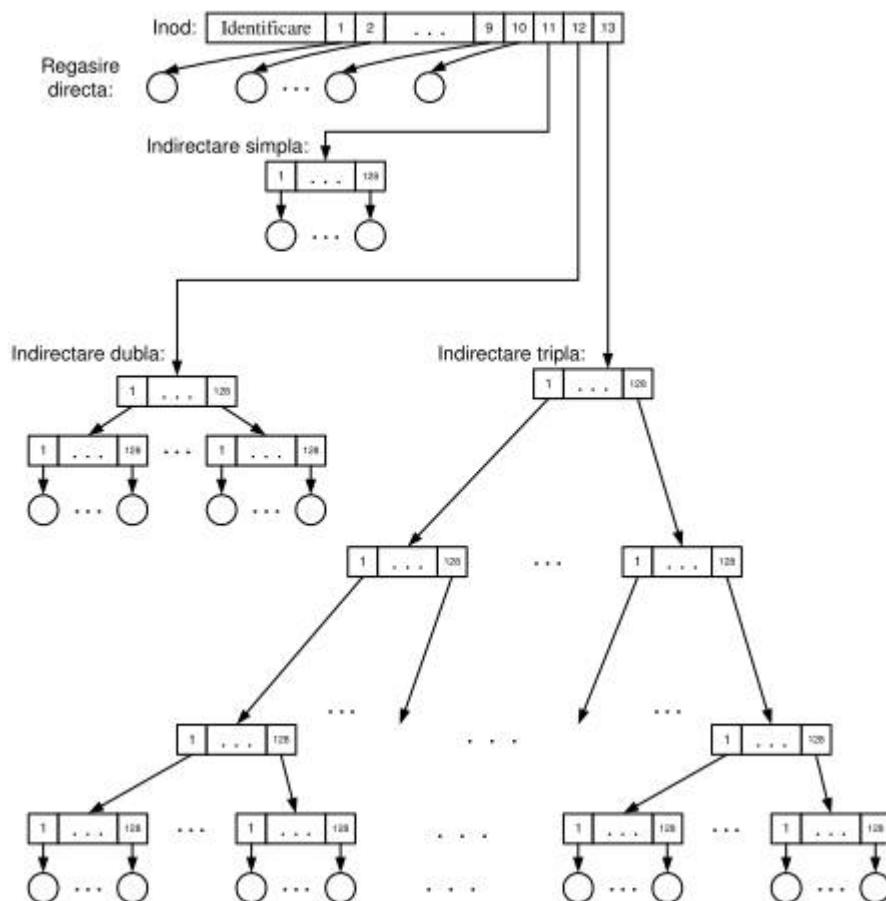
Deci, în director se află numele fișierului și referința spre inodul descriptor al fișierului. Un inod are, de regulă, între 64 și 128 de octeți și el conține informațiile din tabelul următor:

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

3.1.1.3. Schema de alocare a blocurilor disc pentru un fișier

Fiecare sistem de fișiere 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 și câte referințe se trec în lista de referințe indirecte. Indiferent de valorile acestor constante, principiile de înregistrare / regăsire sunt aceleași.

Pentru fixarea ideilor, vom alege aceste constante cu valorile întâlnite mai frecvent la sistemele de fișiere deja consacrate. Astfel, vom presupune că un bloc are lungimea de 512 octeți. O adresă disc se reprezintă pe 4 octeți, deci într-un bloc se pot înregistra 128 astfel de adrese. În 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 atașate unui fișier Unix.



Structura unui inod și accesul la blocurile unui fișier

În inodul fișierului se află o listă cu 13 intrări, care desemnează blocurile fizice aparținând fișierului.

- Primele 10 intrări conțin adresele primelor 10 blocuri de câte 512 octeți care aparțin fișierului.
- Intrarea nr. 11 conține adresa unui bloc, numit bloc de indirectare simplă. El conține adresele următoarelor 128 blocuri de câte 512 octeți, care aparțin fișierului.
- Intrarea nr. 12 conține adresa unui bloc, numit bloc de indirectare dublă. El conține adresele a 128 blocuri de indirectare simplă, care la rândul lor conțin, fiecare, adresele a câte 128 blocuri, de 512 octeți fiecare, cu informații aparținând fișierului.
- Intrarea nr. 13 conține adresa unui bloc, numit bloc de indirectare triplă. În acest bloc sunt conținute adresele a 128 blocuri de indirectare dublă, fiecare dintre acestea conținând adresele a câte 128 blocuri de indirectare simplă, iar fiecare dintre acestea conține adresele a câte 128 blocuri, de câte 512 octeți, cu informații ale fișierului.

În figura de mai sus am ilustrat prin cercuri blocurile de informație care aparțin fișierului, iar prin dreptunghiuri blocurile de referințe, în interiorul acestora marcând referințele. Numărul de accese necesare pentru a obține direct un octet oarecare este cel mult 4. Pentru fișiere mici acest număr este și mai mic. Atât timp cât fișierul este deschis, inodul lui este prezent în memoria internă. Tabelul următor dă numărul maxim de accese la disc pentru a obține, în acces direct orice octet dintr-un fișier, în funcție de lungimea fișierului.

Lungime maximă (blocuri)	Lungime maximă (octeți)	Accese indirecte	Accese la informație	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 octeți care poate înregistra 1024 adrese, iar în inod se înregistrează direct adresele primelor 12 blocuri. În aceste condiții, tabelul de mai sus se transformă în:

Lungime maximă (blocuri)	Lungime maximă (octeți)	Accese indirecte	Accese la informație	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 fișiere și sisteme de fișiere

În cadrul unui sistem de fișiere, apelurile sistem Unix gestionează opt tipuri de fișiere și anume:

1. Normale (obișnuite)
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 entități, pe care apelurile sistem le văd, din punct de vedere sintactic, tot ca și fișiere. Aceste entități sunt gestionate de nucleul Unix, au suportul fizic tot în nucleu și folosite la comunicări între procese. Aceste entități sunt:

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

Fișierele obișnuite sunt privite ca șiruri de octeți, accesul la un octet putându-se face fie secvențial, fie direct prin numărul de ordine al octetului.

Fișierele directori. Un fișier director se deosebește de un fișier obișnuit numai prin informația conținută în el. Un director conține lista de nume și adrese pentru fișierele subordonate lui. Uzual, fiecare utilizator are un director propriu care punctează la fișierele lui obișnuite, sau la alți subdirectori definiți de el.

Fișierele speciale. În această categorie putem include, pentru moment, ultimele 6 tipuri de fișiere. În particular, Unix privește fiecare dispozitiv de I/O ca și un fișier de tip special. Din punct de vedere al utilizatorului, nu există nici o deosebire între lucrul cu un fișier disc normal și lucrul cu un fișier special.

Fiecare director are două intrări cu nume speciale și anume:

- " . " (punct) denumește generic (punctează spre) însuși directorul respectiv;
- " . . " (două puncte succesive), denumește generic (punctează spre) directorul părinte.

Fiecare sistem de fișiere conține un director principal numit **root** sau **/**.

În mod obișnuit, fiecare utilizator folosește un *director curent*, atașat utilizatorului la intrarea în sistem. Utilizatorul poate să-și schimbe acest director (`cd`), poate crea un nou director subordonat celui curent, (`mkdir`), să șteargă un director (`rmdir`), să afișeze *calea* de acces de la `root` la un director sau fișier (`pwd`) etc.

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

- Solaris utilizează sistemul de fișiere `ufs`;
- Linux utilizează cu precădere sistemul de fișiere `ext2` și mai nou, `ext3`;
- IRIX utilizează `xf`
- etc.

Actualele distribuții de Unix permit utilizarea unor sisteme de fișiere proprii altor sisteme de operare. Printre cele mai importante amintim:

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

Din fericire, aceste extinderi sunt transparente pentru utilizatorii obișnuiți. Totuși, se recomandă prudență atunci când se efectuează altfel de operații decât citirea din fișierele create sub alte sisteme de operare decât sistemul curent. De exemplu, modificarea sub Unix a unui octet într-un fișier de tip `doc` creat cu Word sub Windows poate ușor să compromită fișierul așa încât el să nu mai poată fi exploatat sub Windows!

Administratorii sistemelor Unix trebuie să țină cont de sistemele de fișiere pe care le instalează și de drepturile pe care le conferă acestora vis-a-vis de userii obișnuiți.

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

3.1.2.1. Legături hard și legături simbolice

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

Legăturile hard sunt identice cu legăturile naturale și 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 fișiere 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 fișier primește efectiv două nume. Din această cauză, la parcurgerea unei structuri arborescente, fișierele punctate prin legături hard apar duplicate. Fiecare duplicat apare cu numărul de legături către el.

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

```
#ln vechi linknou
```

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

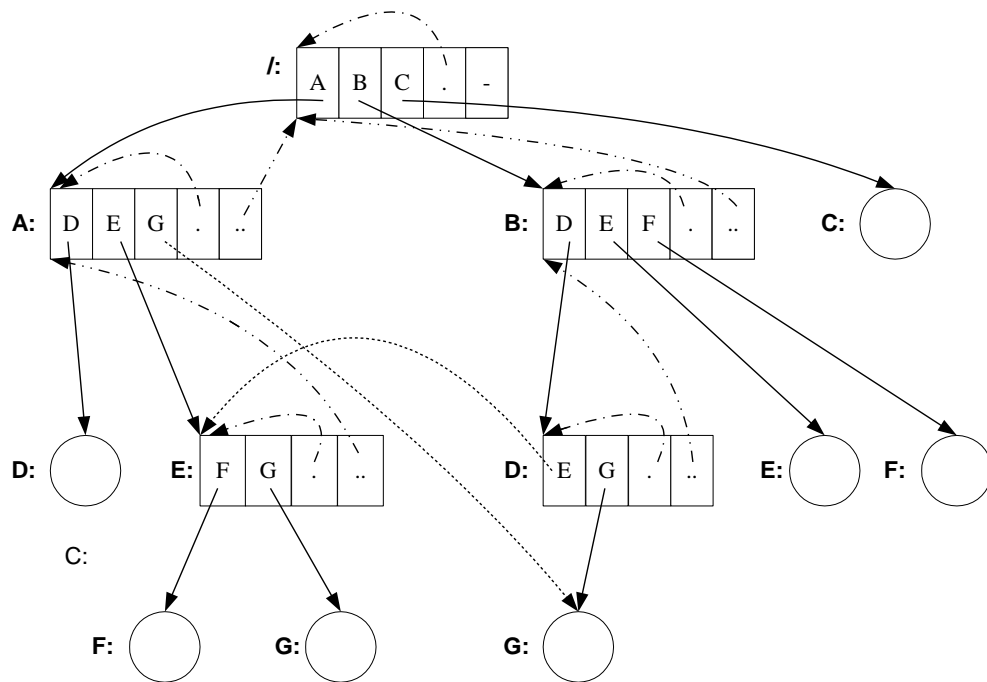
Legăturile hard pot fi făcute numai în interiorul aceluiași sistem de fișiere (detalii puțin mai târziu).

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

În 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 și de către utilizatorii obișnuiți. De asemenea, ele pot puncta și înafara sistemului de fișiere.



Structura arborescentă împreună cu legăturile simbolice sau hard conferă sistemului de fișiere Unix o structură de graf aciclic. În exemplul din figura de mai sus este prezentat un exemplu simplu de structură de fișiere. Prin literele mari A, B, C, D, E, F, G am indicat nume de fișiere obișnuite, nume de directori și nume de legături. Este evident posibil ca același nume să apară de mai multe ori în structura de directori, grație structurii de directori care elimină ambiguitățile. Fișierele obișnuite sunt reprezentate prin cercuri, iar fișierele directori prin dreptunghiuri.

Legăturile sunt reprezentate prin săgeți de trei tipuri:

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

În exemplul de mai sus există 12 noduri - fișiere obișnuite sau directori. Privit ca un arbore, deci considerând numai legăturile naturale, el are 7 ramuri și 4 nivele.

Să presupunem că cele două legături (desenate cu linie punctată) 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 subordonații lui de la stânga spre dreapta. Următoarele 12 linii indică toate cele 12 noduri din structură. Pe aceeași linie apar, atunci când este posibil, mai multe specificări ale aceluiași 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	<u>D/E</u>	<u>./D/E</u>		
/A/E/F	../A/E/F	<u>D/E/F</u>	<u>./D/E/F</u>		#
/A/E/G	../A/E/G	<u>D/E/G</u>	<u>./D/E/G</u>		#
/B	.				
/B/D	D	./D			
/B/D/G	D/G	./D/G	<u>/A/G</u>	<u>../A/G</u>	#
/B/E	E	./E			#
/B/F	F	./F			#
/C	../C				#

Ce se întâmplă cu ștergerea 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 fișierul trebuie să rămână activ dacă este șters numai de către una dintre specificări.

Pentru aceasta, în descriptorul fișierului respectiv există un câmp numit *contor de legare*. Acesta are valoarea 1 la crearea fișierului și crește cu 1 la fiecare nouă legătură. La ștergere, se radiază legătura din directorul părinte care a cerut ștergerea, iar contorul de legare scade cu 1. Abia dacă acest contor a ajuns la zero, fișierul va fi efectiv șters de pe disc și blocurile ocupate de el vor fi eliberate.

3.2. Procese Unix

Procese Unix: creare, funcțiile `fork`, `exec`, `exit`, `wait`; comunicare prin pipe și FIFO

3.2.1. Principalele apeluri system de gestiune a proceselor

În secțiunile din acest subcapitol vom prezenta cele mai importante apeluri sistem pentru lucrul cu procese: `fork`, `exit`, `wait` și `exec*`. Începem cu `fork()`, apelul sistem pentru crearea unui proces.

3.2.1.1. Crearea proceselor Unix. Apelul `fork`

În sistemul de operare Unix un proces se creează prin apelul funcției sistem `fork()`. La o funcționare 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 inițial*. Cele două procese își continuă execuția în paralel cu instrucțiunea care urmează apelului `fork`.

Procesul nou creat poartă numele de *proces fiu*, iar procesul care a apelat funcția `fork()` se numește *proces părinte*. Exceptând faptul că au spații de adrese diferite, procesul fiu diferă de procesul părinte doar prin identificatorul său de proces PID, prin identificatorul procesului părinte PPID și 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.*

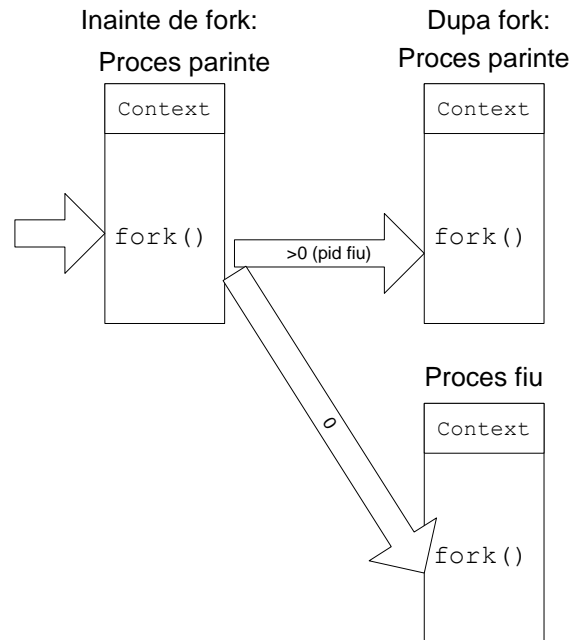


Figura 3.1 Mecanismul fork

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

In caz de eșec, `fork` întoarce valoarea `-1` și desigur, setează corespunzător variabila `errno`. Eșecul apelului `fork` poate să apară dacă:

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

Acest comportament al lui `fork` permite descrierea ușoară a două secvențe de instrucțiuni care să se deruleze în paralel, sub forma:

```
if ( fork() == 0 )
{ - - - instrucțiuni ale procesului fiu - - - }
else
{ - - - instrucțiuni 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");
}
}
}

```

În mod intenționat, am făcut astfel încât procesul fiu să aștepte mai mult decât părintele (în cazul calculului complexe apar adesea situații în care operațiile unuia dintre procese durează mai mult în timp). Ca urmare, părintele va termina mai repede execuția. Rezultatele obținute 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. Execuția unui program extern; apelurile exec

Aproape toate sistemele de operare și 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 aceluiași proces. Apelului `exec i` se furnizează numele unui fișier executabil, iar conținutul acestuia se suprapune peste programul procesului existent, așa cum se vede în Figura 3.2.

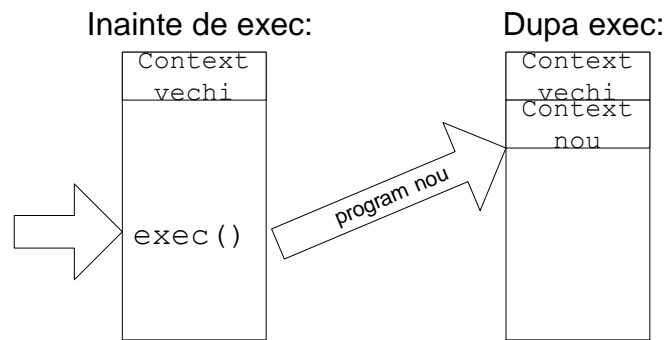


Figure 3.2 Mecanismul exec

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

Unix oferă, în funcție de trei criterii, șase 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 moștenit 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 și mediu nou. Prototipurile celor șase apeluri `exec*()` sunt:

```
int execl (char *fisier, char *argv[], char *argn,
NULL);
int execlp(char *fisier, char *arg0, ..., char *argn,
char *envp[]);
int execvp(char *fisier, char *argv[]);
int execlp(char* fisier, char *arg0, ..., char *argn,
NULL);
```

Semnificația parametrilor `exec` este următoarea:

- `fisier` - numele fișierului 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 conține argumentele liniei de comandă pentru noul program lansat în execuție.
- `arg0, arg1, ... , argn, NULL` conține argumentele liniei de comandă scrise explicit ca și stringuri; această secvență trebuie terminată cu `NULL`.
- `envp` este tabloul de pointeri, terminat cu un pointer `NULL`, care conține stringurile corespunzătoare noilor variabile de mediu sub forma "nume=valoare".

3.2.1.3. Apelurile `exit` și `wait`

Apelul sistem:

```
exit(int n)
```

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

Așteptarea 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 execuția 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

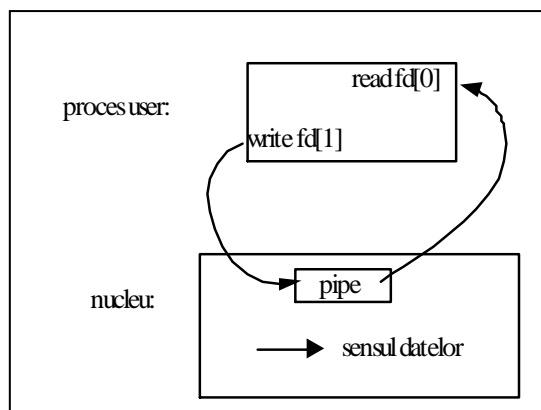
3.2.2.1. Conceptul de pipe

Conceptul a apărut prima dată sub Unix, pentru a permite unui proces fiu să comunice cu părintele său. De obicei procesul părinte redirecționează ieșirea sa standard, `stdout`, către un *pipe*, iar procesul fiu își redirecționează intrarea standard, `stdin`, din același *pipe*. În majoritatea sistemelor de operare se folosește operatorul “|” pentru specificarea acestui gen de conexiuni între comenzi ale sistemului de operare.

Un *pipe* Unix este un flux unidirecțional de date, gestionat de către nucleul sistemului. De fapt, în nucleu se rezervă un buffer de minimum 4096 octeți în care octeții sunt gestionați așa cum am descris mai sus. Crearea unui *pipe* se face prin apelul sistem:

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

Întregul `fd[0]` se comportă ca un întreg care identifică descriptorul pentru citirea din "fișierul" *pipe*, iar `fd[1]` ca un întreg care indică descriptorul pentru scriere în *pipe*. În 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 esențială funcționarea 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.

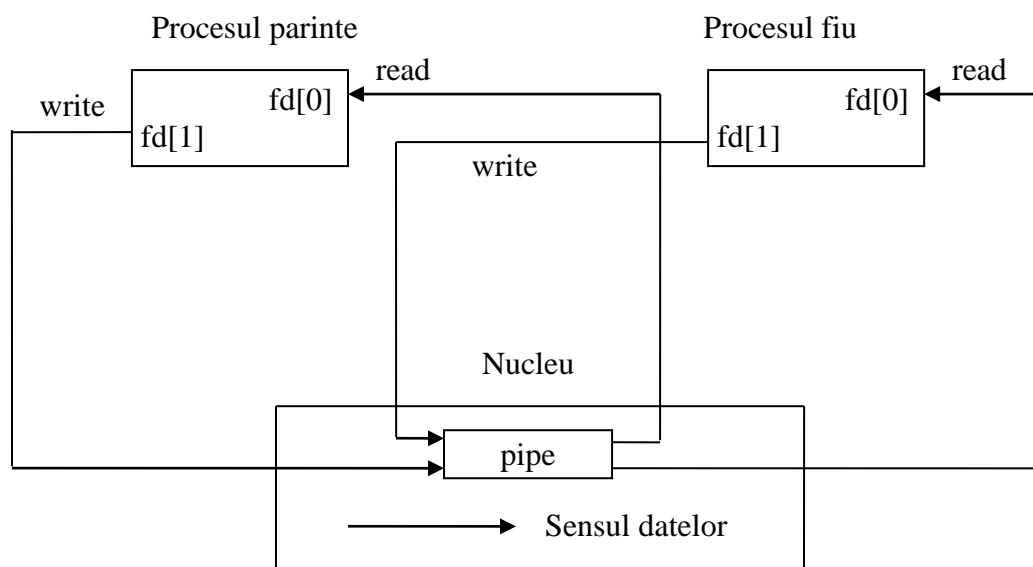


Figura 3.3 Un `pipe` leagă două procese înrudite

Asigurarea unidirecționalității 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 dorește ordinea inversă, atunci se vor executa operațiile `close (fd[1])` în procesul părinte și `close (fd[0])` în procesul fiu.

3.2.2.2. Exemplu: implementarea `who | sort` prin `pipe` și `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 înlocuiește procesul shell) generează doi fii, iar aceștia își redirectează corespunzător intrările / ieșirile. Primul dintre ele execută `who`, celălalt `sort`, iar părintele așteaptă 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 înțelege 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 descendenți din procesul creator al lui *pipe*. Aceasta deoarece întregii descriptori de citire/scriere din/în *pipe* sunt unici și sunt transmiși proceselor fiu ca urmare a apelului `fork()`.

În jurul anului 1985 (Unix System V), a apărut conceptul *FIFO* (*pipe cu nume*). Acesta este un flux de date unidirecțional, accesat prin intermediul unui fișier rezident în sistemul de fișiere. Începând cu Unix System V, există fișiere de tip *FIFO*. Spre deosebire de *pipe*, fișierele *FIFO* au nume și ocupă un loc în sistemul de fișiere. Din această cauză, un *FIFO*

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

Conceptual, canalele *pipe* și *FIFO* sunt similare. Deosebirile esențiale dintre ele sunt următoarele două:

- suportul pentru *pipe* este o porțiune 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 relație 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 "fișierului" de tip *FIFO*.
- Argumentul `mod`, în cazul apelurilor sistem, reprezintă drepturile de acces la acest fișier. În cazul apelului `mknod`, `mod` trebuie să specifice flagul `S_IFIFO`, pe lângă drepturile de acces la fișierul *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`.

Menționăm că cele două apeluri de mai sus, deși sunt specificate de POSIX, nu sunt amândouă apeluri sistem pe toate implementările de Unix. Astfel, pe FreeBSD sunt prezente ambele apeluri sistem `mknod()` și `mkfifo()`, dar pe Linux și Solaris există numai apelul sistem `mknod()`, funcția 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` și `mkfifo` sunt permise numai super-user-ului. Începând cu Unix System V 4.3 ele sunt disponibile și utilizatorului obișnuit.

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

Condiții	normal	setat O_NDELAY
deschide <i>FIFO</i> read-only, dar nu există proces de scriere în <i>FIFO</i>	așteaptă până când apare un proces care deschide <i>FIFO</i> pentru scriere	revine imediat fără a semnala eroare
deschide <i>FIFO</i> write-only, dar nu există proces de citire din <i>FIFO</i>	așteaptă până apare un proces pentru citire	revine imediat cu semnalarea de eroare: variabila <code>errno</code> va deveni <code>ENXIO</code>
citire din <i>FIFO</i> sau din pipe, dar nu există date de citit	așteaptă până când apar date în pipe sau <i>FIFO</i> , 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 <i>FIFO</i> sau pipe, dar acesta este plin	așteaptă până când se face spațiu disponibil, apoi scrie atâtea date, cât îi permite spațiul disponibil	revine imediat, cu întoarcerea valorii 0

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

3.2.3.2. Exemplu: aplicare FIFO la comunicare client / server

Modelul de aplicație client / server este clasic în programare. În cele ce urmează vom ilustra o schemă de aplicații client / server bazate pe comunicații prin *FIFO*. Pentru a se asigura comunicarea bidirecțională se folosesc două *FIFO*-uri. Pentru partea specifică aplicației, se folosesc metodele `client(int in, int out)` și `server(int in, int out)`. Fiecare dintre ele primește ca și parametri descriptorii de fișiere, presupuse deschise, prin care comunică cu partenerul.

În cele două programe care urmează este schițată schema serverului și a clientului. Cele două programe presupun că cele două canale *FIFO* sunt create, respectiv șterse, 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 fișierelor de comenzi

3.3.1. Funcționarea unui interpretor de comenzi shell

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

1. *limbaj de comandă* care asigură interfața dintre calculator și utilizator. În momentul în care un utilizator își deschide o sesiune de lucru, în mod implicit, un *shell* se instalează ca interpretor de comenzi. *Shell*-ul afișează la ieșirea standard (asociată de obicei unui terminal) un prompter, invitând astfel utilizatorul să introducă comenzi sau să lanseze în execuție fișiere de comenzi, eventual parametrizate.
2. *limbaj de programare*, ce are ca element de bază (element primitiv) *comanda* Unix (similară semantic cu *instrucțiunea de atribuire* din limbajele de programare). Ca și 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 *condiția*). *Shell*-urile dispun de conceptele de variabilă, constantă, expresie, structuri de control și subprogram. Spre deosebire de alte limbaje de programare, expresiile cu care lucrează *shell*-urile sunt preponderent șiruri de caractere. În ceea ce privește cerințele sintactice, acestea au fost reduse la minim prin eliminarea parantezelor de delimitare a parametrilor, a diferitelor caractere de separare și terminare, a declarațiilor de variabile, etc.

un shell lansat în execuție 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)
  Afișează prompter;
  Citește linia de comandă;
  Dacă ( linia se termină cu '&' ) atunci
    Crează un proces și-i dă spre execuție comanda
    Nu așteaptă ca execuția să se termine
  Altfel
    Crează un proces și-i dă spre execuție comanda
    Așteaptă să se termine execuția comenzii
  SfDacă
SfCâtTimp
```

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

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

comandă. Comada care se dorește 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 și numai una în foreground. Iată, spre exemplu, trei astfel de comenzi, două lansate în background - o copiere de fișier (comanda `cp`) și o compilare (comanda `gcc`) și una în foreground - editare de fișier (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

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

Vom considera următoarele convenții, 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, construcția dintre paranteze va apărea cel mult odată.
- []+ Semnifică faptul că, construcția dintre paranteze va apărea cel puțin odată.
- []* Semnifică faptul că, construcția dintre paranteze poate să apară de 0 sau mai multe ori.

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

Semnificația unora dintre elementele sintactice din fig. 2.2 este:

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

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

Se poate observa că, în conformitate cu gramatica de mai sus, **sh** acceptă și construcții fără semantică! De exemplu, *comandă* poate fi o *comandăElementară*, care să conțină un singur *element*, format din **>&-;**. O astfel de linie este acceptată de **sh**, fiind corectă din punct de vedere sintactic, deși 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** și **case** sunt închise de construcțiile **fi**, respectiv **esac**, obținute prin oglindirea cuvintelor de start. În cazul ciclurilor repetitive, sfârșitul acestora este indicat prin folosirea cuvântului rezervat **done**. Nu s-a folosit construcția similară corespunzătoare lui **do**, deoarece **od** este numele unui comenzi clasice Unix.

Incheiem acest subcapitol cu prezentarea sintaxei unor **construcții rezervate**, precum și a unor **caractere cu semnificație specială** în shell-ul **sh**.

a) Construcții sintactice:

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

b) Machete și specificări generice:

*	înlocuiește orice șir de caractere
?	înlocuiește orice caracter
[. . .]	înlocuiește cu orice caracter din ...

Observație: aceste machete și specificări generice nu trebuie confundate cu convenția 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 cuvânt in [ cuvânt [ | cuvânt ]* ) listaCom ;; ]+ esac
for nume do listaCom done
for nume in [ cuvânt ]+ 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.

- Descrieți pe scurt funcționarea apelului sistem `fork` și valorile pe care le poate returna.
- Ce tipărește pe ecran secvența de program de mai jos, considerând că apelul sistem `fork` se execută cu succes? Justificați 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;
}
```

- Ce tipărește pe ecran fragmentul de script shell de mai jos? Explicați funcționarea 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. Indicați liniile care se vor tipări pe ecran în ordinea în care vor apărea, considerând că apelul sistem fork se execută cu succes? Justificați 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. Explicați funcționarea fragmentului de script shell de mai jos. Ce se întâmplă, dacă fișierul *raport.txt* lipsește inițial. Adăugați rândul de cod care lipsește pentru generarea fișierului *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 în C++*. Programare generică și modele de proiectare aplicabile, 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., Dragoș 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 și algoritmi. O perspectivă orientată obiect.*, Ed. Casa Cărții de Știință, 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 și Informatică, Centrul de Formare Continuă și Invățământ la Distanță, 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>