Computer Science Manual for Bachelor Graduation Examination June and September 2016

Computer Science Specialization

General topics:

Part 1. Algorithms and Programming

1. Search (sequential and binary), sorting (selection sort, bubble sort, quicksort). The backtracking method.
2. OOP concepts in programming languages (Python, C++, Java, C#): class and object, members of a class and access modifiers, constructors and destructors.
3. Relationships between classes. Derived classes and the inheritance relationship. Method overriding. Polymorphism. Dynamic binding. Abstract classes and interfaces.
4. Class diagrams and UML interactions among objects: Packages, classes and interfaces. Relations between classes and interfaces. Objects. Messages
5. Lists, Maps. Specification of typical operations (without implementations)
6. Identify data structures and data types suitable (efficient) for solving problems (only the data structures specified at 5.). The use of existing libraries for these structures (Python, Java, C++, C#).

Part 2. Databases

1. Relational databases. First three normal forms of a relation**.**
2. Querying databases using relational algebra operators**.**
3. Querying relational databases using SQL (Select)**.**

Part 3. Operating systems

1. The structure of UNIX file systems.
2. Unix processes: creation and the fork, exec, wait system calls. Pipe and FIFO communication.
3. Unix Shell programming and basic Unix commands: cat, cp, cut, echo, expr, file, find, grep, less, ls, mkdir, mv, ps, pwd, read, rm, sort, test, wc, who.

Content

1. Algorithmics and programming

1.1. Searching and sorting

1.1.1 Searching

The data are available in the internal memory, as a *sequence of records*. We will search a record having a certain value for one of its fields, called *search key*. If the search is successful, we will have the position of the record in the given sequence.

We denote by $k_1, k_2, ..., k_n$ the record keys and by a the key value to be found. Our problem is, thus, to find the position p characterized by $a = k_p$.

It is a usual practice to store the keys in increasing sequence. Consequently, we will assume that

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

Sometimes, when the keys are already sorted, we may not only be interested to find the record having the requested key, but, if such a record is not available, we may need to know the insertion place of a new record with this key, such that the sort order is preserved.

We thus have the following specification for the *searching problem*:

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

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

Results p;

Postcondition: $(p=1$ *and* $a \leq k_1)$ *or* $(p=n+1$ *and* $a > k_n)$ *or* $(1<p\leq n)$ *and* $(k_{p-1} < a \leq k_p)$.

1.1.1.1 Sequential search

The first method is the *sequential search,* where the keys are successively examined. We distinguish three cases: $a \leq k_1$, $a > k_n$, and respectively, $k_1 < a \leq k_n$, the last case leading to the actual search.

```
Subalgorithm SearchSeq (a,n,K,p) is:          {n∈N, n≥1 and k1 < k2 < .... < kn}
                                        { Search p such that: (p=1 and a ≤ k1) or}
                                  { (p=n+1 and a>kn) or (1<p≤n) and (kp-1 < a ≤ kp) }
  Let p:=0;                                              {Case "not yet found"}
  If a≤k1 then p:=1 else
    If a>kn then p:=n+1 else
       For i:=2; n do
           If (p=0) and (a≤ki) then p:=i endif
       endfor
    endif
  endif
```

```
sfsub
```

We remark that this method leads to *n*-1 comparisons in the worst case, because the counter *i* will take all the values from 2 to *n*. The *n* keys divide the real axis in *n*+1 intervals. When *a* is between k_1 and k_n, the number of comparisons is still *n*-1, and when *a* is outside the interval [k_1, k_n], there are at most two comparisons. So, the average complexity has the same order of magnitude at the worst-case complexity.

There are many situations when this algorithm does useless computations. When the key has already been identified, it is useless to continue the loop for the remaining values of *i*. In other words, it is desirable to replace the **for** loop with a **while** loop. We get the second subalgorithm, described as follows.

```
Subalgorithm SearchSucc(a,n,K,p) is:                {n∈N, n≥1 and k1 < k2 < .... < kn}
                                                    {Se caută p astfel ca: p=1 and a ≤ k1) or }
                                                    {(p=n+1 and a>kn) or (1<p≤n) and (kp-1 < a ≤ kp)}
   Let p:=1;
   If a>k1 then
      While p≤n and a>kp do p:=p+1 endwh
   endif
sfsub
```

The algorithm *SearchSucc* does *n* comparisons in the worst case. But, on the average, the number of comparisons is reduced to half, and, as such, the average running-time complexity order of *SearchSucc* is the same as with the *SearchSeq* subalgorithm.

1.1.1.2 Binary search

Another method, called *binary search*, more efficient than the previous two methods, uses the "divide and conquer" technique with respect to working with the data. We start by considering the relation of the search key to the key of the element in the middle of the collection. Based on this check we will continue our search in one of the two halves of the collection. We can thus successively halve the collection portion we use for our search. Since we modify the size of the collection, we need to consider the ends of the current collection as parameters for the search. The binary search may effectively be realized with the call *BinarySearch(a, K,* 1*,n)*. This function is described hereby.

```
Subalgorithm SearchBin (a,n,K,p) is:                {n∈N, n≥1 and k1 < k2 < .... < kn}
                                                    {Search p such that: (p=1 and a ≤ k1) or}
                                                    {(p=n+1 and a>kn) or (1<p≤n) and (kp-1 < a ≤ kp)}
  If a≤K1
    then p:=1
    else
      If a>Kn
        then p:=n+1
```

```
          else p:=BinarySearch(a,K,1,n)
        endif
    endif
  sfsub

  Function BinarySearch (a, K, Left, Right) is:
    If Left≥Right-1
      then BinarySearch:=Rigt
      else m:=(Left+Right) Div 2;
         If a≤K_m
           then BinarySearch:=BinarySearch(a,K,Left,m)
           else BinarySearch:=BinarySearch(a,K,m,Right)
         endif
    endif
  sffunc
```

The variables *Left* and *Right* in the *BinarySearch* function described above represent the ends of the search interval, and *m* represents the middle of the interval. Using this method, in a collection with *n* elements, the search result may be provided after at most $log_2 n$ comparisons. Thus, the worst case time complexity is proportional to $log_2 n$.

We remark that the function *BinarySearch* is a recursive function. We can easily remove the recursion, as we see in the following function:

```
Function BinarySearchN (a,K,Left,Right) is:
  While Right-Left>1 do
         m:=(Left+Right) Div 2;
    If a≤K_m
        then Right:=m
        else Left:=m
    endif
  endwh
  BinarySearchN:=Right
endfunc
```

1.1.2 Internal sorting

Internal sorting is the operation to reorganize the elements in a collection already available in the internal memory, in such a way that the record keys are sorted in increasing (or decreasing, if necessary) order.

From an algorithms complexity point of view, our problem is reduced to keys sorting. So, the specification of the **internal sorting** problem is the following:

Data n,K; $\{K=(k_1,k_2,...,k_n)\}$

Precondition: $k_i \in R, i=1,n$

Results K';

Postcondition: K' is a permutation of K, having sorted elements, i.e.

$$k'_1 \le k'_2 \le ... \le k'_n.$$

1.1.2.1 Selection sort

The first technique, called *Selection Sort*, works by determining the element having the minimal (or maximal) key, and swapping it with the first element. Now, forget about the first element and resume the procedure for the remaining elements, until all elements have been considered.

```
Subalgorithm SelectionSort(n,K) is:                    {Do a permutation of the}
                                                       {n components of vector K such}
                                                       {that k1 ≤ k2 ≤ .... ≤ kn }
 For i:=1; n-1 do
      Let ind:=i;
      For j:=i+1; n do
           If kj < kind then ind:=j endif
      endfor
     If i<ind then t:=ki; ki:=kind; kind:=t endif
 endfor
sfsub
```

We remark that the total number of comparisons is

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

independently of the input data. So, the average computational complexity, as well as the worst-case computational complexity, is $O(n^2)$.

1.1.2.2 Bubble sort

Another method, called *BubbleSort*, compares two consecutive elements, which, if not in the expected relationship, will be swapped. The comparison process will end when all pairs of consecutive elements are in the expected order relationship.

```
Subalgorithm BubbleSort (n,K) is:
  Repeat
    Let kod:=0;                                        {Hypothesis "is sorted"}
    For i:=2; n do
      If ki-1 > ki then
        t := ki-1;
        ki-1 := ki;
        ki:=t;
        kod:=1                                         {Not sorted yet!}
      endif
    endfor
  until kod=0 endrep                                   {Sorted}
sfsub
```

This algorithms performs (n-1)+(n-2)+ ... +2+1 = n(n-1)/2 comparisons in the worst case, so the time complexity is $O(n^2)$.

An optimized variant of *BubbleSort* is:

```
Subalgorithm BubbleSort (n,K) is:
  Let s:=0;
  Repeat
    Let kod:=0;                                          {Hypothesis "is sorted"}
    For i:=2; n-s do
      If k_{i-1} > k_i then
        t := k_{i-1};
        k_{i-1} := k_i;
        k_i:=t;
        kod:=1                                           {Not sorted yet!}
      endif
    endfor
    Let s:=s+1;
  until kod=0 endrep                                     {Sorted}
sfsub
```

1.1.2.3 Quicksort

Another, more efficient sorting method is described hereby. The method, called *QuickSort*, is based on the "divide and conquer" technique. The subsequence to be sorted is given through two input parameters, the inferior and superior limits of the substring elements indices. The procedure call to sort the whole sequence is: `QuickSortRec(K,1,n)`, where `n` is the number of records of the given collection. So,

```
Subalgorithm QuickSort (n,K) is:
  Call QuickSortRec(K,1,n)
sfsub
```

The procedure `QuickSortRec(K,Left,Right)` will sort the subsequence $k_{Left}, k_{Left+1}, ..., k_{Right}$. Before performing the actual sort, the substring will be partitioned in such a way that the element k_{Left} (called *pivot*) occupies the final position in the subsequence. If i is this position, the substring will be rearranged such that the following condition is fulfilled:

$$k_j \le k_i \le k_l \text{ , for } Left \le j < i < l \le Right \qquad (*)$$

As soon as the partitioning is achieved, we will only need to sort the subsequence $k_{St}, k_{St+1}, ..., k_{i-1}$ using a recursive call to `QuickSortRec(K,Left,i-1)` and then the subsequence $k_{i+1}, ..., k_{Dr}$ using a recursive call to `QuickSortRec(K,i+1,Right)`. Of course, we will need to sort these subsequences only if they have at least two elements. Otherwise, a subsequence of one element is, actually, sorted.

The procedure *QuickSort* is described hereby:

```
Subalgorithm QuickSort (K, Left, Right) este:
   Let i := Left; j := Right; a := k_i;
   Repeat
     While k_j ≥ a and (i < j) do j := j - 1 endwh
     k_i := k_j;
     While k_i ≤ a and (i < j) do i := i + 1 endwh
     k_j := k_i ;
   until i = j endrep
  Let k_i := a;
  If St < i-1 then Call QuickSort(K, St, i - 1) endif
  If i+1 < Dr then Call QuickSort(K, i + 1, Dr) endif
endsub
```

The time complexity of the described algorithm is $O(n^2)$ in the worst case, but the average time complexity is $O(n\log_2 n)$.

1.1.3 The backtracking method

The backtracking method is applicable to search problems with more solutions. Its main disadvantage is that it has an exponential running time. We are first considering two examples and then will give a few general algorithms for this method.

Problem 1. (Permutations generation) Let *n* be a natural number. Print all permutations of numbers 1, 2, ..., *n*.

A solution for the permutation generation problem in the particular case $n = 3$, is:

```
Subalgorithm Permutations1 is:
    For i1 := 1;3 execute
        For i2 := 1;3 execute
            For i3 := 1;3 execute
                Let possible := (i1, i2, i3)
                If components of the possible array are distinct
                    then Print possible
                endif
            endfor
        endfor
    endfor
endsub
```

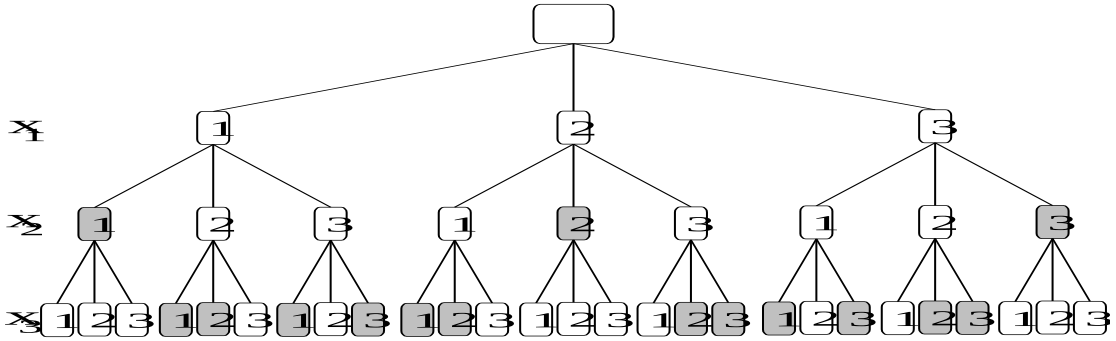


Figure 1.1. Graphical representation of the Cartesian product $\{1, 2, 3\}^3$

Let us discuss a few remarks on the subalgorithm *Permutations1*:

- It is not general: for the general case we cannot describe an algorithm with *n* imbricated **for** loops.
- The total number of checked arrays is 3^3, and in the general case n^n. The checked *possible* arrays are graphically represented in Figure 1.1: each array is a path from the tree root to the leaves.
- The algorithm *first* assigns values to all components of the array *possible*, and *afterwards* checks whether the array is a permutation.

One way to improve the efficiency of this algorithm is to check a few conditions during the construction of the array, avoiding in this way the construction of a complete array in the case we are certain it does not lead to a correct solution. For example, if the first component of the array is 1, then it is useless to assign the second component the value 1; if the second component has been assigned the value 2, it is useless to assign the third component the values 1 or 2. In this way, for a large *n* we avoid generating many arrays of the type (1, ...). For example, (1, 3, ...) is a *potential array* (potentially leading to a solution), while the array (1, 1, ...) is definitely not. The array (1, 3, ...) satisfies certain *conditions to continue* (set to lead to a solution): it has distinct components. The gray nodes in Figure 1.1 represent values that do not lead to a solution.

We will describe a general algorithm for the backtracking method, and then we will see how this algorithm may be used to solve the two particular problems we studied in this section. To start, we will state a few remarks and notations concerning the backtracking algorithm, applied to a problem where the solutions are represented on arrays of length not necessarily constant:

1. the solutions search space: $S = S_1 \times S_2 \times ... \times S_n$;
2. *possible* is the array to represent the solutions;
3. *possible*$[1..k] \in S_1 \times S_2 \times ... \times S_k$ is the subarray of solution candidates; it may or may not lead to a solution, i.e. it may or may not be extended to form a complete solution; the index *k* is the number of already constructed solution elements;

4. *possible*[1..*k*] is promising if it satisfies the conditions that may lead to a solution (Figure 1.2).;
5. *solution* (*n, k, possible*) is a function to check whether the potential array *possible*[1..*k*] is a solution of the problem.

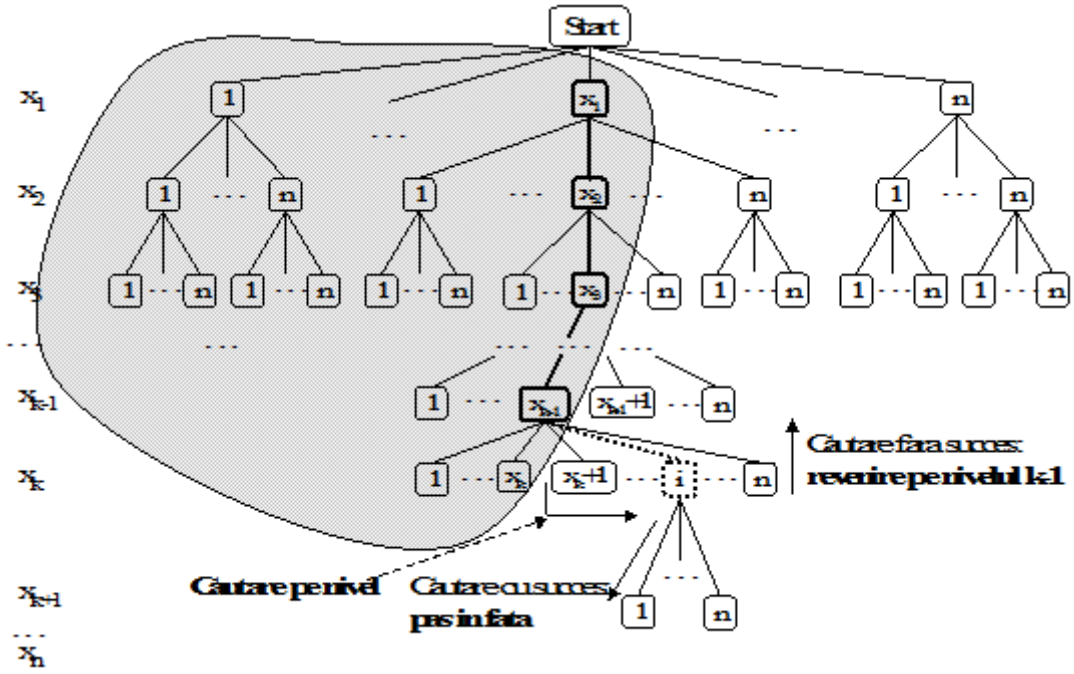


Figure 1.2. The search space for the permutations problem

The search process may be seen in the following subalgorithm:

```
Subalgorithm Backtracking(n) is:                          {draft version }
Let k = 1;
@Initialise the search for index k (= 1)
While k > 0 do
    {possible[1..k-1] is a solution candidate }
    @Sequentially search for index k a value v to extend the subarray
       possible[1..k-1] such that possible[1..k] is still a solution
       candidate
    If the search is successful
       then Let possible[k] := v;                         {possible[1..k] is a solution candidate}
             If solution (n, k, possible)                 {found a solution; we are still on level k}
             then Print possible[1..k]
                else @ Initialize the search for index k+1        {a potential array }
                Let k = k + 1                                     {step forward on level k+1}
             endif
          else k = k - 1                                  {step backward (backtrack to level k-1)}
    endif
endwh
endSub
```

In order to write the final version of this algorithm we need to specify the non-standard elements. We thus need the Boolean function

```
condToContinue  (k, possible, v)
```

that checks whether the subarray with the solution candidate *possible*[1..*k*-1], extended with the value *v*, is still a solution candidate.

Then, to initialize the search at level *j* we need a way to select a fictive element for the set S_j, with the purpose of indicating that no element has been selected from the set S_j. The function

```
init (j)
```

returns this fictive element.

In order to search a value on the level *j*, in the hypothesis that the current value is not good, we need the Boolean function

```
next (j, v, new)
```

which returns *true* if it may select a new value in S_j that follows after the value *v*, value denoted by *new* and *false* when no other values in S_j exist, so no new choice may be made. With these notations, the subalgorithm becomes:

```
Subalgorithm Backtracking(n) is:                                    {final version }
Let k = 1 ;
possible[1] := init(1);
While k > 0 execute                                          {possible[1..k-1] is potential}
   Let found := false; v := possible[k] ;
   While next (k, v, new) and not found do
     Let v := new;
     If condToContinue (k, possible, v) then found := true endif
   endwh
   If found then Let possible[k] := v                          {possible[1..k] is potential}
                 if solution (n, k, possible)     {found a solution; we are still on level k}
                          then Print possible[1..k]
                          else Let k = k + 1;                          {a potential array }
                               possible[k] := init(k);      {step forward on level k+1}
              endif
                  else k = k - 1                    {step backward (backtrack to level k-1)}
   endif
endwh
endSub
```

The process of searching a value on level *k* and the functions *condToContinue* and *solution* are problem-dependent. For example, for the permutation generation problem, these functions are:

```
function init(k) is:
   init:= 0
endfunc

function next (k, v, new) is:
   if v < n
      then next := true;
           new:= v + 1
      else next := false
   endif
endfunc

function condToContinue (k, possible, v) is:
   kod:=True; i:=1;
   while kod and (i<k)  do
      if possible[i] = v then kod := false endif
```

```
      endwh
      condToContinue := kod
   endfunc

   function solution (n, k, possible) is:
      solution := (k = n)
   endfunc
```

To conclude, we are providing here the recursive version of the backtracking algorithm, described using the same helping functions:

```
Subalgorithm backtracking(n, k, possible)  is:            {possible[1..k] is potential}
   if solution(n, k, possible)                    {a solution; terminate the recursive call}
      then print possible[1..k]                              {else, stay on same level}
      else for each value  v possible for possible[k+1] do
         if condToContinue(k+1, possible, v)
            then possible[k+1] := v
                 backtracking (n, k+1, possible)                    {step forward}
         endif
      endfor
   endif                                        {terminate backtracking(n, k, possible) call}
endsub                                                              {so, step backward}
```

The problem is solved by the call *backtracking(n, 0, possible).*

1.2. OOP concepts in programming languages

1.2.1 Classes

1.2.1.1 Data protection in modular programming

In procedural programming, developing programs means using functions and procedures for writing these programs. In the C programming language instead of functions and procedures we have functions that return a value and functions that do not return a value. But in case of large applications it is desirable to have some kind of data protection. This means that only some functions have access to problem data, specifically functions referring to that data. In modular programming, data protection may be achieved by using static memory allocation. If in a file a datum outside any function is declared static then it can be used form where it was declared to the end of the file, but not outside it.

Let us consider the following example dealing with integer vector processing. Write a module for integer vector processing that contains functions corresponding to vector initialization, disposing occupied memory, raising to the power two and printing vector elements. A possible implementation of this module is presented in the file **vector1.cpp**:

```
#include <iostream>
```

```
using namespace std;

static int* e;              //vector elements
static int d;               //vector size

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

void destroy()                //disposing occupied memory
{
  delete [] e;
}

void squared()             // raising to the power two
{
  for(int i = 0; i < d; i++)
    e[i] *= e[i];
}

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

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

The module is individually compiled and an object file is produced. A main program example is presented in the file **vector2.cpp**:

```
extern void init( int*, int);  //extern may be omitted
extern void destroy();
extern void squared();
extern void print();
//extern int* e;
int main() {
  int x[5] = {1, 2, 3, 4, 5};
  init(x, 5);
  squared();
  print();
  destroy();
  int y[] = {1, 2, 3, 4, 5, 6};
  init(y, 6);
  //e[1]=10;                   error, data are protected
  squared();
  print();
  destroy();
  return 0;
}
```

Note that even though the main program uses two vectors, we cannot use them together, so for example the module **vector1.cpp** cannot be extended to implement vector addition. In order to overcome this drawback, abstract data types have been introduced.

1.2.1.2 Abstract data types

Abstract data types enable a tighter bound between the problem data and operations (functions) referring to these data. An abstract data type declaration is similar to a struct declaration, which apart of the data also declares or defines functions referring to these data.

For example in the integer vector case we can declare the abstract data type:

```
struct vect {
  int* e;
  int d;
  void init(int* e1, int d1);
  void destroy() { delete [] e; }
  void squared();
  void print();
};
```

The functions declared or defined within the struct will be called *methods* and the data will be called *attributes*. If a method is defined within the struct (like the *destroy* method from the previous example) then it is considered an *inline* method. If a method is defined outside the struct then the function name will be replaced by the abstract data type name followed by the scope resolution operator (::) and the method name. Thus the *init*, *squared* and *print* methods will be defined within the module as follows:

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

Even though by the above approach a tighter bound between problem data and functions referring to these data has been accomplished, data are not protected, so they can be accessed by any user defined function, not only by the methods. This drawback may be overcome by using classes.

1.2.1.3 Class declaration

A class abstract data type is declared like a struct, but the keyword struct is replaced with *class*. Like in the struct case, in order to refer to a class data type one uses the name following the keyword class (the class name). Data protection is achieved with the access modifiers: *private, protected* and *public.* The access modifier is followed by the character ':'. The *private* and *protected* access modifiers represent protected data while the *public* access modifier represent unprotected data. An access modifier is valid until the next access modifier occurs within a class, the default access modifier being *private*. Note that structs also allow the use of access modifiers, but in this case the default access modifier is *public*.

For example the vector class may be declared as follows:

```
class vector {
  int* e;  //vector elements
  int d;   //vector size
public:
  vector(int* e1, int d1);
  ~vector() { delete [] e; }
  void squared();
  void print();
};
```

Note that the attributes *e* and *d* have been declared *private* (restricted access), while methods have been declared public (unrestricted access). Of course that some attributes may be declared public and some methods may be declared private if the problem specifics requires so. In general, private attributes can only be accessed by the methods from that class and by *friend functions.*

Another important remark regarding the above example is that attribute initialization and occupied memory disposal was done via some special methods.

Data declared as some class data type are called the classes' *objects* or simply *objects.* They are declared as follows:

```
class_name list_of_objects;
```

For example, a vector object is declared as follows:

```
vector v;
```

Object initialization is done with a special method called *constructor*. Objects are disposed by an automatic call of another special method called *destructor*. In the above example

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

is a constructor and

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

is a destructor.

Abstract data types of type *struct* may also be seen as classes where all elements have unrestricted access. The above constructor is declared inside the class, but it is not defined,

while the destructor is defined inside the class. So the destructor is an inline function. In order to define methods outside a class, the scope resolution operator is used (like in the *struct* case).

1.2.1.4 Class members. The this pointer

In order to refer to class attributes or methods the dot (.) or arrow (→) operator is used (like in the *struct* case). For example, if the following declarations are considered:

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

then printing the vector v and the vector referred by the p pointer is done as follows:

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

However inside methods, in order to refer to attributes or (other) methods only their name needs to be used, the dot (.) or arrow (→) operators being optional. In fact, the compiler automatically generates a special pointer, the *this* pointer, at each method call and it uses the generated pointer to identify attributes and methods.

The *this* pointer will be declared automatically as a pointer to the current object. In the example from above the *this* pointer is the address of the vector v and the address referred by the p pointer respectively.

For example, if inside the *print* method an attribute *d* is used then it is interpreted as *this->d*.

The *this* pointer may also be used explicitly by the programmer.

1.2.1.5. The constructor

Object initialization is done with a special method called constructor. The constructor name has to be the same with the class name. The class may have multiple constructors. In this case these methods will have the same name and this is possible due to function overloading. Of course that the number and/or formal parameter types has to be different otherwise the compiler cannot choose the correct constructor.

Constructors do not return any value. In this situation the use of the keyword *void* is forbidden.

In the following we show an example of a class having as attributes a person's last name and first name and a method for printing the person's whole name.

File **person.h**:

```
class person {
  char* lname;
  char* fname;
public:
  person();                        //default constructor
```

```
    person(char* ln, char* fn);      //constructor
    person(const person& p1);    //copy constructor
    ~person();                     //destructor
    void print();
  };
```

File **person.cpp**:

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

using namespace std;

person::person()
{
  lname = new char[1];
  *lname = 0;
  fname = new char[1];
  *fname = 0;
  cout << "Calling default constructor." << endl;
}

person::person(char* ln, char* fn)
{
  lname = new char[strlen(ln)+1];
  fname = new char[strlen(fn)+1];
  strcpy(lname, ln);
  strcpy(fname, fn);
  cout << "Calling constructor (lname, fname).\n";
}

person::person(const person& p1)
{
  lname = new char[strlen(p1.lname)+1];
  strcpy(lname, p1.name);
  fname = new char[strlen(p1.fname)+1];
  strcpy(fname, p1.fname);
  cout << "Calling copy constructor." << endl;
}

person::~person()
{
  delete[] lname;
  delete[] fname;
}

void person::print()
{
  cout << fname << ' ' << lname << endl;
}
```

File **personTest.cpp**:

```
#include "person.h"

int main() {
  person A;             //calling default constructor
  A.print();
```

```
  person B("Stroustrup", "Bjarne");
  B.print();
  person *C = new person("Kernighan","Brian");
  C->print();
  delete C;
  person D(B);          //equivalent to person D = B;
                          //calling copy constructor
  D.print();
  return 0;
}
```

We may notice the presence of two special types of constructors: the *default constructor* and the *copy constructor*. If a class has a constructor without any parameters then this is called *default constructor*. The *copy constructor* is used for object initialization given an object of the same type (in the above example a person having the same last and first name). The copy constructor is declared as follows:

```
class_name(const class_name& object);
```

The *const* keyword expresses the fact that the copy constructor's argument is not changed.

A class may contain attributes of other class type. Declaring the class as:

```
class class_name {
  class_name_1 ob_1;
  class_name_2 ob_2;
  ...
  class_name_n ob_n;
  ...
};
```

its constructor's header will have the following form:

```
class_name(argument_list):
      ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)
```

where argument_list and l_arg_i respectively represent the list of formal parameters from the *class_name*'s constructor and object *ob_i* respectively.

From the list `ob_1(l_arg_1), ob_2(l_arg_2), ..., ob_n(l_arg_n)` one my choose not to include the objects that do not have user defined constructors, or objects that are initialized by the default constructor, or by a constructor having only implicit parameters.

If a class contains attributes of another class type then first these attributes' constructors are called followed by the statements from this classes' constructor.

File **`pair.cpp`**:

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

using namespace std;
```

```
class pair {
  person husband;
  person wife;
public:
  pair()               //implicit constructor definition
  {                       //the implicit constructors
  }                       //for objects husband and wife are called
  pair(person& ahusband, person& awife);
  pair(char* lname_husband, char* fname_husband,
    char* lname_wife, char* fname_wife):
    husband(lname_husband, fname_husband),
    wife(lname_wife, fname_wife)
  {
  }
  void print();
};

inline pair::pair(person& ahusband, person& awife):
  husband(ahusband), wife(awife)
{
}

void pair::print()
{
  cout << "husband: ";
  husband.print();
  cout << "wife: ";
  wife.print();
}

int main() {
  person A("Pop", "Ion");
  person B("Popa", "Ioana");
  pair AB(A, B);
  AB.print();
  pair CD("C","C","D","D");
  CD.print();
  pair EF;
  EF.print();
  return 0;
}
```

Note that in the second constructor, the formal parameters *husband* and *wife* have been declared as references to type *person.* If they had been declared as formal parameters of type *person*, then in the following situation:

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

the copy constructor would have been called four times. In situations like this, temporary objects are first created using the copy constructor (two calls in this case), and then the constructors of the attributes having a class type are executed (other two calls).

1.2.1.6 The destructor

The destructor is the method called in case of object disposal. Global object destructor is called automatically at the end of the *main* function as part of the *exit* function. So using the

exit function in a destructor is not recommended as it leads to an infinite loop. Local objects destructor is executed automatically when the bloc in which these objects were defined is finished. In case of dynamically allocated objects, the destructor is usually called indirectly via the *delete* operator (provided that the object has been previously created using the *new* operator). There is also an explicit way of calling the destructor and in this case the destructor name needs to be preceded by the class name and the scope resolution operator.

The destructor name starts with the ~ character followed by the class name. Like in the constructor case, the destructor does not return any value and using he *void* keyword is forbidden. The destructor call is various situations in shown in the following example:

File **`destruct.cpp`**:

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

using namespace std;

class write {  //write on stdout what it does.
  char* name;
public:
  write(char* n);
  ~write();
};

write::write(char* n)
{
  name = new char[strlen(n)+1];
  strcpy(name, n);
  cout << "Created object: " << name << '\n';
}

write::~write()
{
  cout << "Destroyed object: " << name << '\n';
  delete name;
}

void function()
{
  cout << "Call function" << '\n';
  write local("Local");
}

write global("Global");

int main() {
  write* dynamic = new write("Dynamic");
  function();
  cout << "In main" << '\n';
  delete dynamic;
  return 0;
}
```

1.3. Relationships between classes

1.3.1 Theoretical basis

The use of abstract data types creates an ensamble for managing data and operations on this data. By means of the abstract type class data protection is also achieved, so usually the protected elements can only be accessed by the methods of the given class. This property of objects is called *encapsulation.*

But in everyday life we do not see separate objects only, but also different relations among these objects, and among the classes these objects belong to. In this way a class hierarchy is formed. The result is a second property of objects: *inheritance*. This means that all attributes and methods of the base class are inherited by the derived class, but new members (both attributes and methods) can be added to it. If a derived class has more than one base class, we talk about *multiple inheritance.*

Another important property of objects belonging to the derived class is that methods can be overridden. This means that an operation related to objects belonging to the hierarchy has a single signiture, but the methods that describe this operation can be different. So, the name and the list of formal parameters of the method is the same in both the base and the derived class, but the implementation of the method can be different. Thus, in the derived class methods can be specific to that class, although the operation is identified through the same name. This property is called *polymorphism.*

1.3.2. Declaration of derived classes

A derived class is declared in the following way:

```
class name_of_derived_class : list_of_base_classes {
  //new attributes and methods
};
```

where `list_of_base_classes` is of the form:

```
elem_1, elem_2, ..., elem_n
```

and `elem_i` for each $1 \leq i \leq n$ can be

```
public base_class_i
```

or

```
protected base_class_i
```

or

```
private base_class_i
```

The *public*, *protected* and *private* keywords are called *inheritance access modifiers* in this situation too. They can be missing, and in this case the default modifier is *private*. Access to elements from the derived class is presented on Table 1.

Access to elements from the base class	Inheritance access modifier	Access to elements from the derived class
public	public	public
protected	public	protected
private	public	inaccesible
public	protected	protected
protected	protected	protected
private	protected	inaccesible
public	private	private
protected	private	private
private	private	inaccesible

Tabel 1: access to elements from the derived class

We can observe that *private* members of the base class are inaccesible in the derived class. *Protected* and *public* members become *protected* and *private*, respectively, if the inheritance access modifier is *protected* and *private*, respectively, and remain unchanged if the inheritance access modifier is *public*. This is why, generally, attributes and methods are declared *protected* and the inheritance access modifier is *public*. Thus, they can be accessed, but are protected in the derived class, too.

1.3.3. Virtual functions

Polymorphism leads naturally to the problem of determining the method that will be called for a given object. Let us consider the following example. We declare a base class, called *base*, and a class derived from this class, called *derived*. The base class has two methods: *method_1* and *method_2* and *method_2* calls *method_1*. In the derived class *method_1* is overridden, but *method_2* is not. In the main program an object of the derived class is declared and *method_2*, inherited from the base class, is called. In the C++ language, this example is written in the following way:

File `virtual1.cpp`:

```
#include <iostream>

using namespace std;

class base {
  public:
    void method_1();
    void method_2();
};
```

```
class derived : public base {
  public:
    void method_1();
};

void base::method_1()
{
  cout << "Method method_1 of the"
       << " base class is called" << endl;
}

void base::method_2()
{
  cout << "Method method_2 of the"
       << " base class is called" << endl;
  method_1();
}

void derived::method_1()
{
  cout << "Method method_1 of the"
       << " derived class is called" << endl;
}

int main() {
  derived D;
  D.method_2();
}
```

Executing the code, we will have the following result:

```
Method method_2 of the base class is called
Method method_1 of the base class is called
```

But this is not the desired result, because in the *main* function method *method_2*, inherited from the base class, was called, but method *method_1* called by *method_2* was determined at compile-time. Consequently, although *method_1* was overridden in the derived class, the method from the base class was called, not the overridden one.

This shortcoming can be overcome by introducing the notion of *virtual* methods. If a method is virtual, than for every call of it, the implementation corresponding to the class hierarchy will not be determined at compile-time, but at execution, depending on the type of the object on which the call was made. This property is called *dynamic binding*, and if a method is determined at compile-time, we talk about *static binding.*

We have seen that if the `virtual1.cpp` program is executed, methods *method_1* and *method_2* from the base class are called. But *method_1* being overridden in the derived class, we wanted the overridden method to be called instead of the one from the base class.

This can be realised by declaring *method_1* as a virtual method. Thus, for each call of *method_1*, the implementation of the method that will be called is determined at execution-time and not at compile-time. So, the method *method_1* is determined through *dynamic binding.*

In the C++ language a method is declared virtual in the following way: in the declaration of the class, the header of the method will start with the keyword *virtual*.

If a method is declared virtual in the base class, then the methods overriding it will be considered virtual in all derived classes of the hierarchy.

For the above example the declaration of the base class is modified in the following way:

```
class base {
  public:
    virtual void method_1();
    void method_2();
};
```

The result of the execution becomes:

```
Method method_2 of the base class is called
Method method_1 of the derived class is called
```

So, *method_1* from the derived class is called indeed.

Further we will present another example, where the neccessity of introducing virtual methods appears. Let us define the class *fraction* referring to rational numbers, having as attributes the numerator and the denominator of the fraction. The class has to have a constructor, the default value for the numerator being 0 and for the denominator being 1, and two methods: *product*, for computing the product of two fractions and *multiply*, for multiplying the current object with a fraction given as parameter. Also, the *fraction* class has to have a method for displaying a rational number. Using class *fraction* as base class, we will define the derived class *fraction_write*, in which the *product* function will be overridden, so that besides executing the multiplication, the operation is displayed on *stdout*. The *multiply* method will not be overridden, but the performed operation has to be displayed on the standard output in this case, too. File **`fvirt1.cpp`**:

```
#include <iostream>

using namespace std;

class fraction {
protected:
     int numerator;
     int denominator;
public:
     fraction(int numerator1 = 0, int denominator1 = 1);
     fraction product(fraction& r);    //computes the product of two
                                       //fractions, but does not simplify
     fraction& multiply(fraction& r);
     void display();
};

fraction::fraction(int numerator1, int denominator1)
{
     numerator     = numerator1;
     denominator   = denominator1;
}

fraction fraction::product(fraction& r)
{
     return fraction(numerator * r.numerator, denominator * r.denominator);
}
```

```
fraction& fraction::multiply(fraction& q)
{
     *this = this->product(q);
     return *this;
}

void fraction::display()
{
     if ( denominator )
          cout << numerator << " / " << denominator;
     else
          cerr << "Incorrect fraction";
}

class fraction_write: public fraction{
public:
     fraction_write( int numerator1 = 0, int denominator = 1 );
     fraction product( fraction& r);
};

inline fraction_write::fraction_write(int numerator1, int denominator1) :
fraction(numerator1, denominator1)
{
}

fraction fractie_write::product(fraction& q)
{
     fraction r = fraction(*this).product(q);
     cout << "(";
     this->display();
     cout << ") * (";
     q.display();
     cout << ") = ";
     r.display();
     cout << endl;
     return r;
}

int main()
{
     fraction p(3,4), q(5,2), r;
     r = p.multiply(q);
     p.display();
     cout << endl;
     r.display();
     cout << endl;
     fraction_write p1(3,4), q1(5,2);
     fraction r1, r2;
     r1 = p1.product(q1);
     r2 = p1.multiply(q1);
     p1.display();
     cout << endl;
     r1.display();
     cout << endl;
     r2.display();
     cout << endl;
     return 0;
}
```

Executing the code we will get:

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

We can observe that the result is not the desired one, since the multiplication operation was displayed only once, namely for the expression `r1 = p1.product(q1)`. In case of the expression `r2 = p1.multiply(q1)` the multiplication was not displayed. This is caused by the fact that the *multiply* method was not overriden in the derived class, so the method inherited from class *fraction* was called. Inside *multiply* the method *product* is called, but since this method was determined at compile-time, the one referring to class *fraction* was called and not the one from the derived class *fraction_write.* So, the operation was displayed only once.

The solution is, like for the previous example, to declare a virtual method, namely to declare method *product* virtual. So, the declaration of the base class is modified in the following way:

```
class fraction {
protected:
     int numerator;
     int denominator;
public:
     fraction(int numerator1 = 0, int denominator = 1);
     virtual fraction product(fraction& r);  //computes the product of two
                                    //fractions, but does not simplify
     fraction& multiply(fraction& r);
     void display();
};
```

After making these modifications, the result of the execution will be:

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

It can be observed that the operation was displayed twice, once for each expression. Virtual methods, just like other methods, do not neccessarily have to be overridden in the derived classes. If they are not overridden, the method from a superior level is inherited.

The corresponding implementation of virtual methods is determined based on some automatically built and managed tables. Objects of classes with virtual methods contain a pointer to this table. Because of this, managing virtual methods requires more memory and a longer execution time.

1.3.4. Abstract classes

In case of a complicated class hierarchy, the base class can have some properties which we know exist, but we can only define them for the derived classes. For example, let's consider the class hierarchy from Figure 1.3.

We notice that we can determine some properties that refer to the derived classes, for example: average weight, lifespan and speed. These properties will be described using different methods. Theoretically, average weight, lifespan and speed exists for the *animal* class, too, but they are too complicated to determine, and are not important for us in such a general context. Still, for a uniform treatment, it would be good, if these three methods were declared in the base class and defined in the derived classes. For this purpose the notion of *pure virtual method* was introduced.

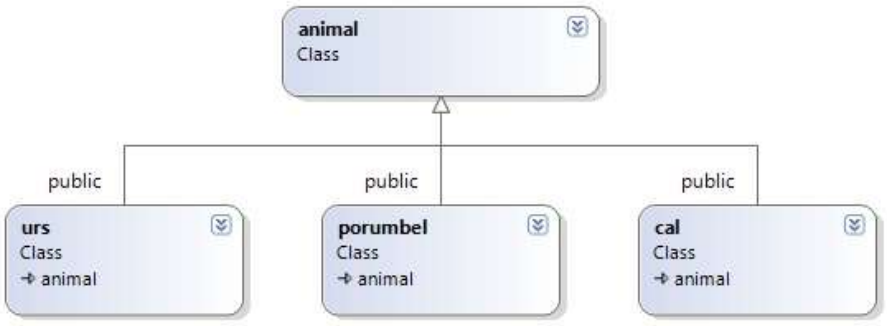


Figure 1.3. Class hierarchy of animals

A pure virtual method is a method which is declared in a given class, but is not defined in it. It *has to* be defined in a derived class. A pure virtual method is declared in the following way: the regular header of the method is preceeded by the *virtual* keyword, and the header ends with = 0. As its name and declaration show, a pure virtual method is a virtual method, so the selection of the implementation of the method from the class hierarchy will be done during the execution of the program.

Classes that contain at least one pure virtual method are called *abstract classes.*

Since abstract classes contain methods that are not defined, it is not possible to create objects that belong to an abstract class. If a pure virtual method was not defined in the derived class, than the derived class will also be abstract and it is impossible to define objects belonging to it.

Let's consider the above example and write a program that determines whether a *dove*, a *bear* or a *horse* is fat or skinny, fast or slow and old or young, respectively. The result will be displayed by a method of the *animal* class, which is not overridden in the derived classes. File **`abstract1.cpp`**:

```
#include <iostream>

using namespace std;

class animal {
protected:
  double weight;  // kg
```

```
  double age;    // years
  double speed;    // km / h
public:
  animal( double w, double a, double s);
  virtual double average_weight() = 0;
  virtual double average_lifespan() = 0;
  virtual double average_speed() = 0;
  int fat() { return weight > average_weight(); }
  int fast() { return speed > average_speed(); }
  int young()
    { return 2 * age < average_lifespan(); }
  void display();
};

animal::animal( double w, double a, double s)
{
  weight = w;
  age = a;
  speed = s;
}

void animal::display()
{
  cout << ( fat() ? "fat, " : "skinny, " );
  cout << ( young() ? "young, " : "old, " );
  cout << ( fast() ? "fast" : "slow" ) << endl;

}

class dove : public animal {
public:
  dove( double w, double a, double s):
    animal(w, a, s) {}
  double average_weight() { return 0.5; }
  double average_lifespan() { return 6; }
  double average_speed() { return 90; }
};


class bear: public animal {
public:
  bear( double w, double a, double s):
    animal(w, a, s) {}
  double average_weight() { return 450; }
  double average_lifespan() { return 43; }
  double average_speed() { return 40; }
};

class horse: public animal {
public:
  horse( double w, double a, double s):
    animal(w, a, s) {}
  double average_weight() { return 1000; }
  double average_lifespan() { return 36; }
  double average_speed() { return 60; }
};

int main() {
  dove d(0.6, 1, 80);
  bear b(500, 40, 46);
  horse h(900, 8, 70);
```

```
  d.display();
  b.display();
  h.display();
  return 0;
}
```

We notice that, although the *animal* class is abstrat, it is useful to introduce it, since there are many methods that can be defined in the base class and inherited without modifications in the three derived classes.

1.3.5. Interfaces

The C++ language has no notion of interfaces, which exist in Java or C# languages. But any abstract class that contains only pure virtual methods can be considered an interface. Obviously, in this case no attributes will be declared inside the class. The *animal* abstract class contains both attributes and nonvirtual methods, so it cannot be considered an interface.

Further we will introduce an abstract class, *Vehicle*, which contains only pure virtual methods, and two classes derived from it. File **`vehicle.cpp`**:

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

class Vehicle
{
public:
     virtual void Start() = 0;
     virtual void Stop() = 0;
     virtual void Go(int km) = 0;
     virtual void Stand (int min) = 0;
};

class Bicycle : public Vehicle
{
public:
     void Start();
     void Stop();
     void Go(int km);
     void Stand(int min);
};

void Bicycle::Start() {
     cout << "The bicycle starts." << endl;
}
void Bicycle::Stop() {
     cout << "The bicycle stops." << endl;
}
void Bicycle::Go(int km) {
     cout << "The bicycle goes " << km <<
          " kilometers." << endl;
}
void Bicycle::Stand(int min) {
     cout << "The bicycle stands " << min <<
          " minutes." << endl;
}
```

```
class Car : public Vehicle
{
public:
     void Start();
     void Stop();
     void Go(int km);
     void Stand(int min);
};

void Car::Start() {
     cout << "The car starts." << endl;
}
void Car::Stop() {
     cout << "The car stops." << endl;
}
void Car::Go(int km) {
     cout << "The car goes " << km <<
          " kilometers." << endl;
}
void Car::Stand(int min) {
     cout << "The car stands " << min <<
          " minutes."  << endl;
}

void Path(Vehicle *v)
{
     v->Start();
     v->Go(3);
     v->Stand(2);
     v->Go(2);
     v->Stop();
}

int main()
{
     Vehicle *b = new Bicycle;
     Path(b);
     Vehicle *c = new Car;
     Path(c);
     delete b;
     delete c;
}
```

In the *main* function two dynamic objects of type *Bicycle* and *Car*, respectively, are declared, and in this way, calling the *Path* function we will get different results, although this function has as formal parameter only a pointer to the abstract class *Vehicle*.

1.4 Class diagrams and UML interactions among abjects: Packages, classes and interfaces. Relations between classes and interfaces. Objects. Messages

The Unified Modelling Language (UML) [29] defines a set of *modelling elements* and *graphical notations* associated to these elements. The modelling elements can be used for describing any software system. Particularly, the UML language contains elements which can be used for the object oriented systems.

This section contains several basic elements used for describing the **structure** and **behaviour** of an object oriented software system – *class diagrams* and *interactions diagrams.* These elements correspond to the selection from [30], chapters 3 and 4.

Before presenting the above mentioned elements, we start to show the context in which they are used for a software system development. The main questions we have to answer are: (A) **what types of models** we build, (B) **when should we build the models** depending on the used development processes and (C) what is the **connection between the models and the written code**.

For briefly responding to these questions, we will give some examples related to an application used by a cashier for recording the selling at a point of sale in a store. The application is called **POS** (Point of Sale) and assumes the implementation of a single use case, recording a sale.

A. Types of models

Regarding the types of models we build, it is appropriate to use the concepts introduced by model driven architecture approaches, mainly **Model-Driven Architecture - MDA** [31]. In the following we describe the models proposed by the MDA guide.

CIM - Computation Independent Models. These models describe **what** the system **does** and not **how this behaviour is supplied.** They are also known **domain models** (or **business models**) and describe the problem domain. From a structural perspective, the class diagrams are used to define the **domain concepts.** The interaction diagrams are rarely used for **CIM**. In order to express the behaviour desired for the system, other elements are used, such as use cases, business processes, etc – but these are not discussed in this section.

The first model from Figure 1.4 presents an extract from the conceptual model for the POS application. The model is built for illustrating the concepts used by the users. These concepts are used for expressing the desired behaviour, using other modelling elements.

PIM - Platform Independent Models. These models describe **how the system is working,** independent from the possible concrete platforms in which it will be implemented. At this level, the **architectural elements** will be introduced, and the class and interaction diagrams between the objects represent two important tools for a detailed description of the system (**detailed design**). Certainly, other modelling elements are also used, structural and behavioural, but these are not discussed here – for example collaborations, state machines, activities, etc.

The second model from Figure 1.4 presents an extract of the PIM model for *POS*. Both the CIM and PIM models contain only UML constructions (ex. Data types defined in the UML specification) and possibly extensions of this language (platform independent).

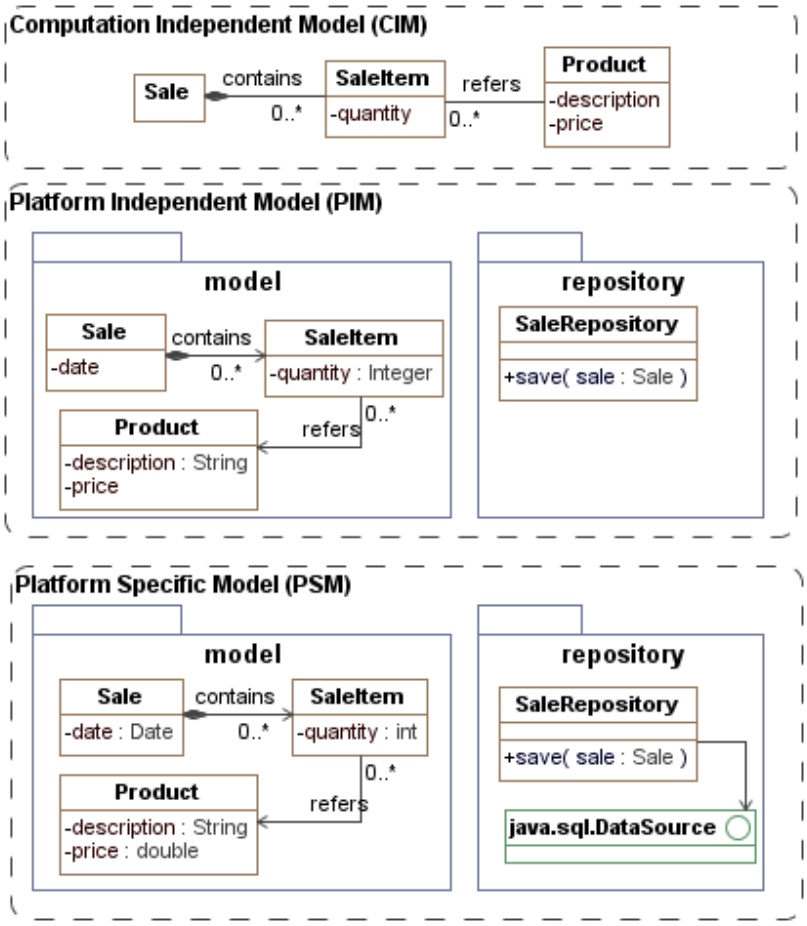


Figure 1.4 Types of models

PSM - Platform Specific Models. These models are a transformation of the *PIM* models toward different chosen platforms. The architect can decide to build such a model in order to express different used elements of the chosen platforms, for example specific data types. The class diagrams and interaction diagrams between the objects are also used for these models.

The last model shown in Figure 1.4 represents a transformation of the PIM model when implementing the system in Java. The data types from this model are Java types (ex. String, double), and the model includes a data type from the java.sql package.

According to the MDA guide, we define the models such that to finally **generate code to some chosen platforms.** The code may be generated starting from the PIM or PSM models. In the generation process, **correspondences between the model's elements and elements from the chosen platform** are used.

B. Development processes and CASE tools

Different development processes indicate the use of different type of models at different development stages.

The model driven development processes generally subscribe to the guide [31] and create PIM models, optionally derived from the CIM models. Then, from the PIM models, they generate code, using optionally an intermediate PSM model. These development processes require the use of design tools (CASE - Computer Aided Software Engineering) which support this infrastructure for model transformation. Examples are the model driven

processes for service oriented applications that use platform independent languages/extensions of UML, such as SoaML[1].

There exist **model driven processes which are not using PIM models, but directly the PSM models**. These are based on specifications made for different platforms, such as component based architectures which are service providers, SCA[2].

The more sophisticated development processes such as RUP[3] (for large systems), recommend the use of all **CIM, PIM** and **PSM** models, in the context of using CASE tools.

The agile development processes , such as test driven development[4] or model driven agile development[5], do not generally recommend the use of CASE tools but recommend the use of models before start writing the code. **The models are in fact sketches** (written on a paper or blackboard) and are used for communicating ideas about the system design.

Independent of the used development process, most of the modern development tools allow **direct synchronization between the written code and the corresponding models**. This synchronization is actually between the code and the PSM models. For example, a CASE tool which synchronizes the models with the code written in Java, does it between the code and the PSM models according to the Java platform.

A last and recent category of development processes which proposes the use of **PIM** models and the **direct and complete code generation** is the category of processes based on **executable models**. Nowadays, the adoption of the standard for executable UML models (fUML - Foundational UML)[6] is being to be finalized. According to these processes, we expect a **new development style** in which will be build only **models** and the **code** will be written **in a textual language**[7] **defined on the elements from these models**. This way, the PSM models and the code written in languages like Java, C++ or C# will be automatically generated by the CASE tools.

C. The correspondence between the models and the code

The correspondences between the models and the code are important, as shown by (A) and (B). If we generate code from the PIM models, or if we are using a CASE tool which synchronizes the PSM model with the code, it is important to know what kind of elements will be generated from the built models. Even if we are working agile and we are using sketches (without using CASE tools), the same problem arises.

Considering the executable models mentioned above (which are at the PIM level), in the following sections we will discuss only the **correspondences between the PIM models** and **C++, Java and C# languages**. The PSM models contain in addition to the PIM models data types specific to some languages, thus the correspondences between the PSM models and the code are the same, but there are present in models and UML extensions according to the specific types.

---

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

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/

1.4.1 Class diagrams

A diagram is a graphical representations (usually 2D) of the elements from a model. **The class diagrams** represent the types of objects used in a system, as well as the relationships between them. The structural elements selected in this section are (a) **types of objects**: classes, interfaces, enumerations; (b) **grouping of elements** using packages and (c) **relationships between theses elements**: associations, generalizations, realizations and dependencies.

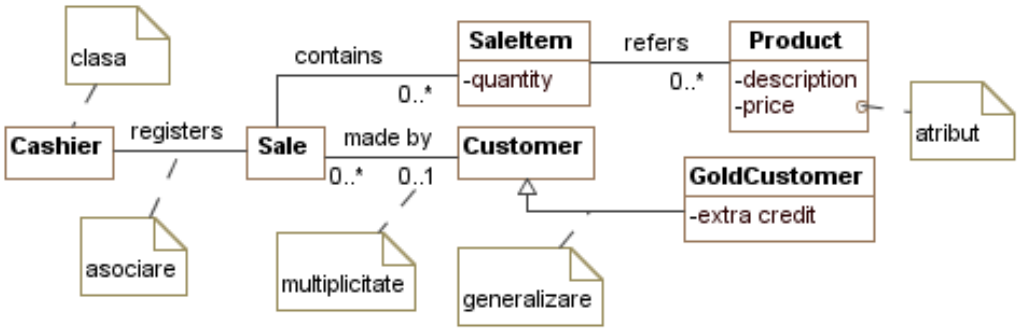


Figure 1.5 Conceptual model

Figure 1.5 presents an initial conceptual model for *POS*. The classes are used for identifying the concepts from a domain. If it is not relevant, the compartment for the class attributes is hidden. The classes properties are defined through attributes and associations and the data types for the attributes are not given.

The conceptual models are of CIM type and are used for generating PIM models. As the CIM models, they can not contain details regarding the attributes representation. If the development process used does not assume the use of a CIM model (but PIM or PSM), then the model from Figure 1.5 is a PIM or an incomplete PSM model.

In the PIM context, the architecture of the system from the structural perspective is described using packages (hierarchically organized) and dependencies between them – see Figure 1.6 for *POS*. The packages are defined with cohesive responsibilities, such as the interaction with the user (*UI*), facade over the domain (*service*), entities (*model*) and deposits of objects or data access objects (*repository*, *inmemory repository*).

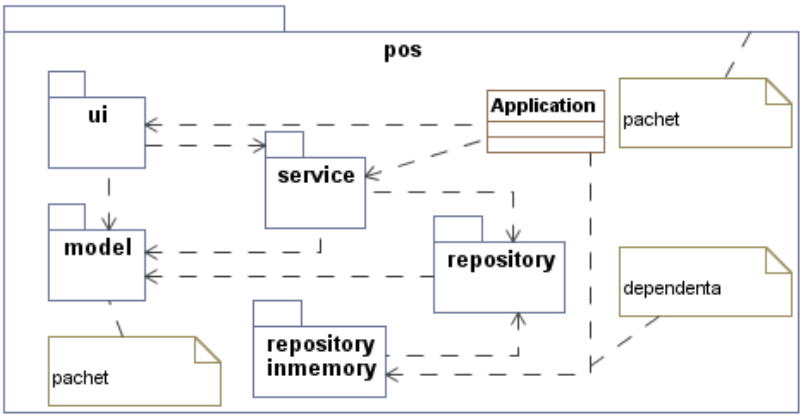


Figure 1.6 Stratified architecture

The main concern when deciding the system's architecture is to follow the SOLID[8,9] object oriented principles. The packages from Figure 1.6 are designed conform to the single responsibility principle, according to which the objects should have a single responsibility, and the objects with related responsibilities need to be logically grouped.

The dependency between two software elements (*A* depends on *B*) indicate that when an element will be modified (*B*), it is possible that the dependant element (*A*) have to be also modified. The dependencies between the packages from Figure 1.6 respect the layered architectures recommendation, i.e. the elements from the higher levels are dependant on the lower levels, for example *UI* depends on the *service* and *model*, *service* depends on *model* and *repository*, but *service* does not depend on the concrete implementation for the repository, namely *repository inmemory*. Inverting the latest dependency follows another SOLID principle, namely dependency inversion. Figure 1.7 shows the details regarding the inversion of the dependencies between *service* and *repository inmemory*. Instead of *StoreService* being dependant on the concrete implementation *InmemorySaleRepository*, the *SaleRepository* interface was introduced in order to decouple these two elements. Actually, *SaleRepository* abstracts the access to the *Sale* objects, allowing this way to replace the *repository inmemory* system with another implementation, without affecting the others packages from the system.

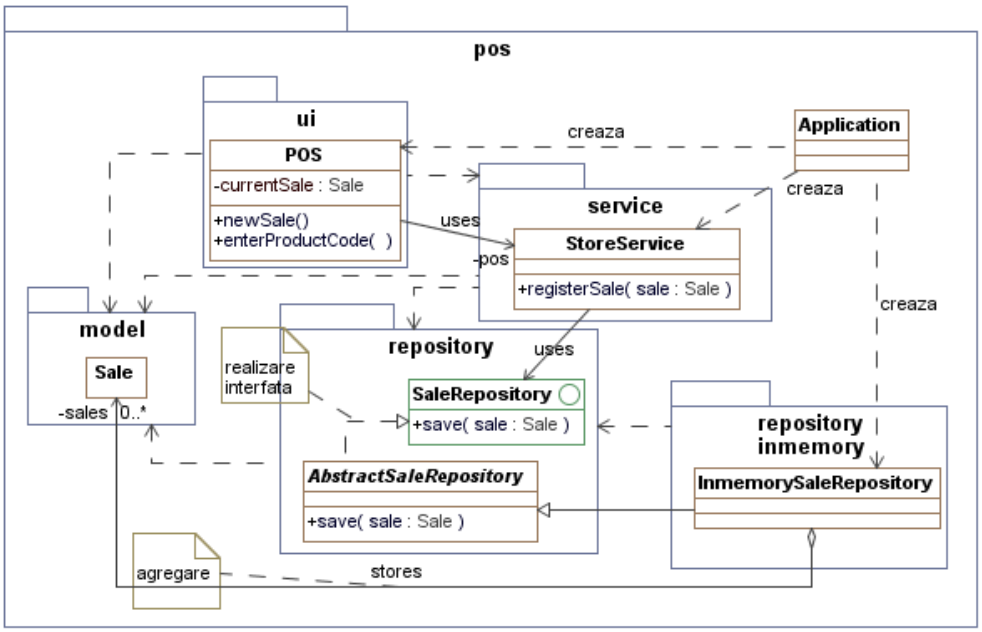


Figure 1.7 Layered architecture – justification of the dependencies

At the PIM or PSM levels, the class diagrams are used to refine the entities and the relationships between them – see Figure 1.8. These elements will be discussed in the following sections.

[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)

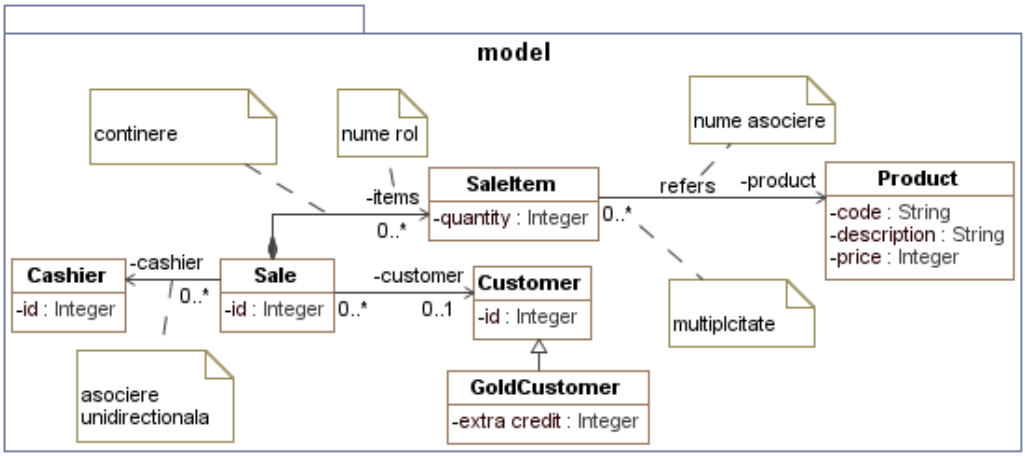


Figure 1.8 POS entities

A. Packages

The UML packages [29, 30] group elements from the model and offer a namespace for the grouped elements. The packages may contain data types and other packages. A data type or a package may be part of a single package.

In terms of programming languages, the UML packages correspond to Java packages and C++ and C# namespaces. The UML packages are referred using the resolution operator ::, as well as in C++ and C#. For example, the complete name of the *UI* package from Figures 1.6 or 1.7 is *pos*::*ui*.

The class diagrams which indicate the packages of a system are used for describing its architecture – see Figure 1.6 for the *POS* system. The dependencies between the packages indicate a summary of the dependencies between the contained elements and the elements from other packages. From an architectural perspective, a good management of these dependencies is very important in the process of building and maintaining the system.

B. Classes

An UML class [29, 30] represent a set of objects with the same structural elements (properties) and behavioural elements (operations). The UML classes are data types and correspond to application classes from Java, C++ and C# languages. A class may be declared **abstract** and in this case it can not be instantiated as in Java, C++ and C#.

An UML class may be **derived** from many other classes, as in C++. The use of multiple inheritance in the model does not lead to a direct correspondence between the model and the code in case of Java or C# languages.

An UML class can **implement** many interfaces as in Java or C#. The correspondence between the models which contain classes that implement multiple interfaces and C++ is made through purely abstract C++ classes and multiple inheritance.

All the classes from Figure 1.8 are concrete classes, and *AbstractSaleRepository* from Figure 1.7 is abstract class (the name is written italic).

The substitution principle is applicable for all the instances having a class or interface type, as in Java, C++ and C#. This means that an instances of a class may be replaced with instances of the derived types, without semantically altering the program.

C. Interfaces

An UML interface [29, 30] is a data type which declares a set of operations, i.e. a contract. A class that implement an interface need to provide the declared operations (fulfill the contract). This concept correspond to the same Java/C# concept and to the pure abstract classes from C++.

SaleRepository from Figure 1.7 is an interface When the highlighting of the interface's methods is not relevant, the graphical notation for interfaces is the one from Figure 1.9.

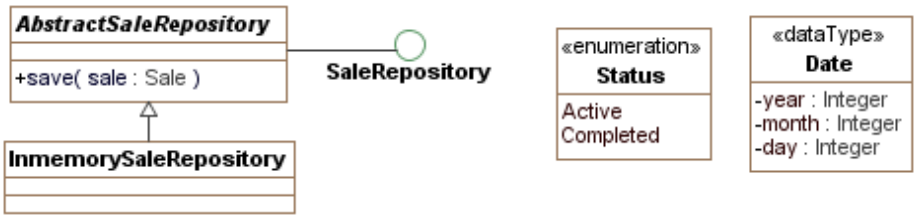


Figure 1.9 Interface, enumeration and structured types

D. Enumerations and value objects

UML enumerations [29, 30] describe a set of symbols which have no associated values, as the same concepts may be found in C++, Java and C#.

The structured types [29, 30] are modelled using the *datatype* stereotype and correspond to C++/C# structures and primitives types from Java. The instances of these data types are identified by their values. They are used to describe the classes properties and correspond to value objects (the value object[10] pattern), but they can not have identity.

E. Generalization and interface realization

The generalization [29, 30] is a relation between a more general data type (basic) and one more specialized (derived). This relation can be applied between two classes or two interfaces, corresponding to the inheritance relations between classes from Java and C++/C# and interfaces (pure abstract classes in C++), respectively

The realization of an interface in UML [29, 30] represents a relation between class and an interface and indicates that the class is conform to the contract specified by the interface. These realizations correspond to interfaces implementations from Java and C# and to inheritance in C++, respectively. See the graphical notations between *AbstractSaleRepository* and *SaleRepository* in **Figure 1.7** and **Figure 1.9**.

F. Properties

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

Properties [29, 30] represent structural aspects of a data type. The properties of a class are introduced through attributes and associations. An **attribute** describes a property of the class in its second compartment, as:

visibility name: *type multiplicity = value* {*properties*}

The **name** is compulsory, as well as the **visibility** which can be public (+), private (-), protected (#) or at the package level (without specifier). The UML visibility correspond to the access specifiers (having the same names) from Java, with the same semantics. The visibility at the package level is not found in C++, and in C# has a correspondence through the internal specifier from C# but this has also distribution connotations for the software elements (the elements declared internal in C# are accessible only in the binary distribution dll or exe which contains them).

The other elements used for declaring a property are optional. **The type** of a property can be any of: class, interface, enumeration, structured type or primitive type. **The primitive types in UML** are value types [29]. UML defines the following primitive types: String, Integer, Boolean and UnlimitedNatural. The first three primitive types are in correspondence with the types having the same name in Java, C++ and C# languages, but:

- The String type is a class in Java and C#, the instances of String type can not be modified, in contrast to C++ where the strings can be modified The character encoding is not given in UML, while in Java and C# it is Unicode, and in C++ ASCII.
- The Integer type in UML is in unlimited precision, while in Java, C++ and C# languages it has a limited domain of values..

The **multiplicity** can be 1 (implicit value, when the multiplicity is not given), , 0..1 (optional), 0..* (zero or more values), 1..* (one or more values), m..n (between *m* and *n* values, where *m* and *n* are constants, *n* can be *). For a property having the multiplicity ***m..**** we can additionally specify if:

- The values can be repetitive or not – **implicitly the values are unique** (i.e. set), otherwise we explicitly specify by **non-unique** (i.e. container with possibly duplicate values).
- The values can be referred or not through indices - **implicitly not** (i.e. collection), otherwise we have to explicitly specify **ordered** (i.e. list).

Examples of properties:

set *: Integer[0..*]* – set of integer values (unique)

list *: Integer[0..*] {ordered}* – list with integer and distinct values (unique)
list *: Integer[0..*] {ordered, non-unique}* – list of integers
collection *: Integer[0..*] {non-unique}* – collection of integers

The UML properties correspond to fields or object type variables in Java and C++, respectively properties in C#. Interpretation difficulties are related to the properties having an *m*..* multiplicity. For the above examples we can consider the following Java correspondences (and similarly with C++/C#):

– set of integers:
  1. *int[] set* or *Integer[] set*, and we will ensure through the operations that *set* will contain distinct values, or more appropriate
  2. *java.util.Set set*
– list with integer and distinct values:
  1. *int[] list*, *Integer[] list* or *java.util.List list*, and we will ensure through the operations that *list* will contain distinct values
– list of integers:

1. *int[] list, Integer[] list,* or *java.util.List list*

– collection of integers:

1. *int[] collection, Integer[] collection,* or *java.util.Collection collection*

UML associations [29, 30] represent a set of tuples each tuple linking two instances of some data types. An association is a data type which links properties of other data types.

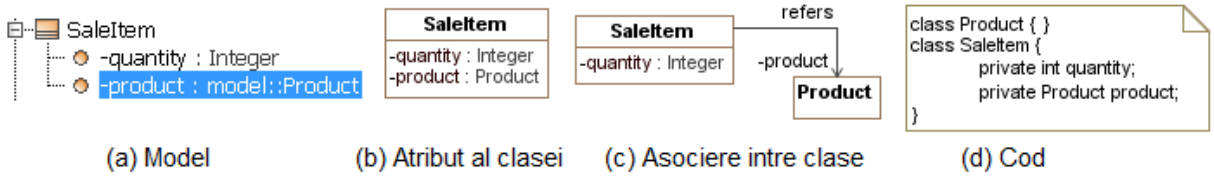


Figure 1.10 Unidirectional associations

Figure 1.10 (a) presents the model resulted after adding the attributes *quantity* and *product* in the class *SaleItem* represented in diagram (b). The code (d) written in Java/C# correspond to this situation. If we consider that it is more appropriate a graphical representation for the relation between the classes *SaleItem* and *Product*, then instead of adding *product* as an attribute, we use an unidirectional association from *SaleItem* to *Product*. When the unidirectional association is added, an association is created in the model and a property is created in the *SaleItem* class, having the role name, i.e. *product*. This way, the code (d) corresponds to the graphical representation (c) of the model (a) which also contains an association not shown in the figure. **The unidirectional association** introduce properties in the source class, having the type of the destination class. The name of the property coincides with the name of the association role, and the general form for defining the properties (presented at the beginning of this section) is also applicable in this case.

The decision of using associations instead of attributes depends on the context. For example, when we are modelling the entities from an application we use associations for indicating relationships between entities and we are using attributes when we describe the entities using value/descriptive objects. Usually, association are used when we want to emphasize the importance of the types and of the relationships between them.

The bidirectional associations link two properties from two different classes or from the same class. **Figure 1.11** presents a bidirectional association between *SaleItem* and *Product*, as well as the Java/C# code corresponding to this situation.

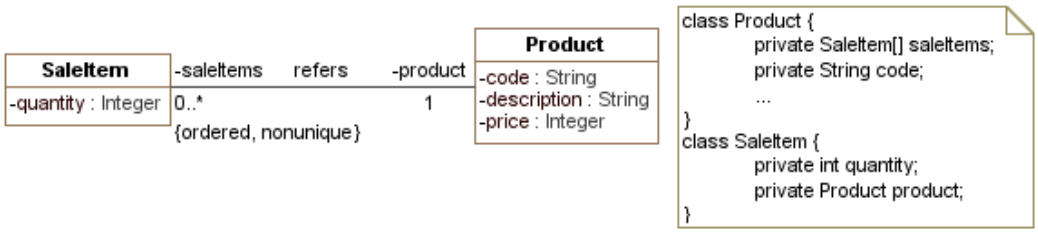


Figure 1.11 Bidirectional association

The conceptual model contain bidirectional associations. Storing the bidirectional associations in the PIM/PSM models can lead to an inefficient execution due to the objects representation. A compulsory step that has to be made during the detailed analysis is the **association refinement**, firstly **the transformation of the bidirectional associations into**

unidirectional ones. **Figure 1.8** presents the result of refining the associations from **Figure 1.5** .

The whole-part relationships are modelled in UML using aggregations and containments. An **aggregation** is an association which indicates that an object is part of another object. For example, Figure 1.7 indicates through the aggregation between *InmemorySaleRepository* and *Sale* that the first object stores all the objects of *Sale* type (the sales are part of this deposit). A **containment** is an aggregation which additionally indicates that the contained objects can be part of a single whole, for example an element of the sale (*SaleItem*) from **Figure 1.8** can be part of a single sale (*Sale*). The **associations' refinement** also includes the setting of the **aggregation** and **containment** relationships.

As in Java, C++ and C#, we can define **static or class type properties**, in diagrams these being represented through underlining.

G. Dependencies

Between two software elements, *client* and *supplier*, a **dependency** [29, 30] exists if changing definition of the *supplier* leads to the change of the *client*. For example, if a class C sends a *message* to another class *F*, then *C* is dependent on *F* because changing the definition of the message in *F* will imply changes in *C* regarding the way of transmission. As a general rule, we should minimize the dependencies in the model, while keeping these elements cohesive.

One can explicitly indicate in UML the dependencies between any elements from the model. But their explicit presentation can make the model hard to read. That is why, the dependencies are presented selectively, emphasizing important dependencies and architectural elements – see Figures 1.6 and 1.7.

H. Operations

Operations in UML [29, 30] define the behaviour of the objects and correspond to the methods in object oriented programming languages. Actually, the operations specify the behaviour (represent the signature) and the body/implementation is defined by behavioural elements such as interactions, state machines and activities – the implementations are known as **methods** in UML. The syntax for specifying an operation is:

visibility name (*list-of-parameters*) : *returned-type* {*properties*}

where *visibility*, *returned-type* and *properties* are defined as for the classes properties. In the operation list of *properties* one can specify if it is only a query operation *{query}*, i.e. an operation which does not modify the state of the object – implicitly, the operations are considered commands, i.e. they modify the state of the objects. The parameters from the *list-of-parameters* are separated by commas, a parameter having the following form:

direction name: *type* = *implicit-value*,

direction can be: *in*, *out* and *in-out*, implicitly being *in*.

1.4.2 Interaction diagrams

The UML interactions [29, 30] describe the behaviour of the system, indicating how multiple participants involved in a scenario are collaborating There are more interaction types, but in this section we will discuss only about **sequence diagrams**, a type of interactions which describe the messages sent between participants.

Usually the sequence diagrams describe a **single scenario.** For example, the diagram from Figure 1.12 shows the cashier and the *POS* system, describing the normal flow for the single use case discussed here, recording a sale. Such a diagram helps to identify the **public interface of the system**. Starting from the use cases description, the users actions are modelled as messages to which the system has to respond. Messages 1, 2, 4 and 6 from **Figure 1.12** indicate that system should make some computations and to respond to the user.

The vertical bar associated to a participant represent the **temporal axis**. On this axis we place the **activation bar** for indicating when those participant is involved in the interaction. The **messages** have a name, are generally numbered and can indicate a returned **response**. All the messages from the figures of this section are synchronous The interactions may contain **fragments**: **cycles** and **alternatives**.

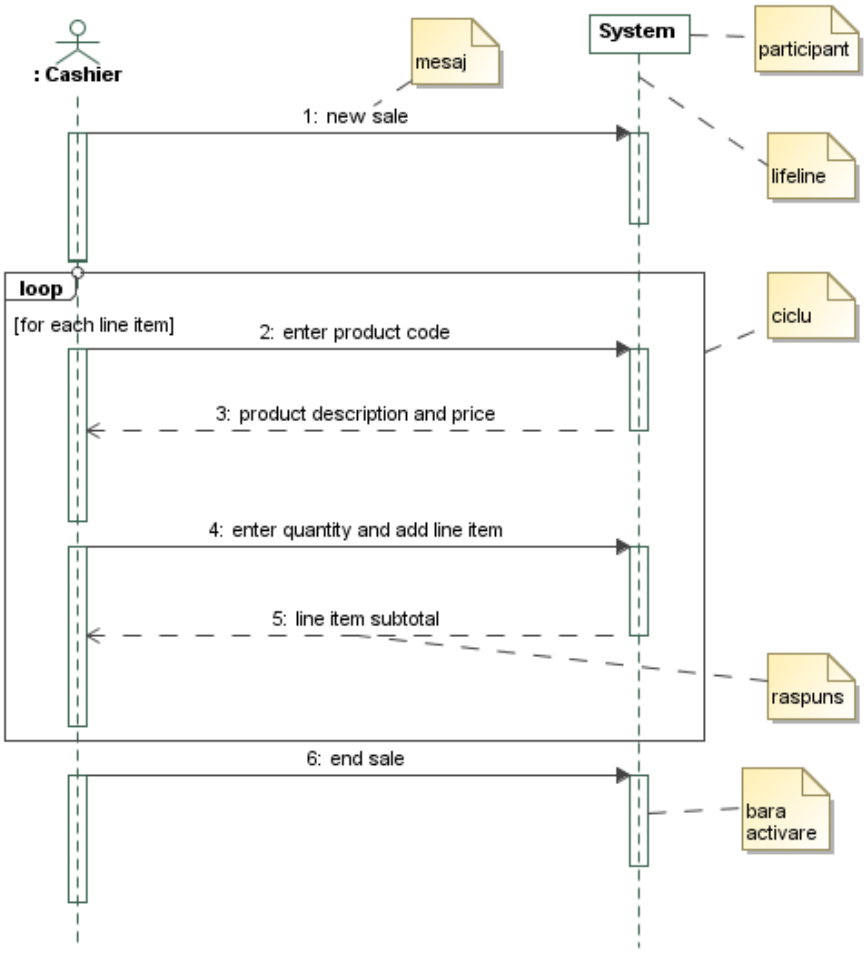


Figure 1.12 The interface of the system

The diagrams like the one presented in Figure 1.12 may be defined in the CIM context, before establishing an architecture. At the PIM level, once we have identified **what the system should do**, we use sequence diagrams for detailing **how the objects from the system**

will collaborate, according to the responsibilities specified through the selected architecture. Figure 1.13 presents in detail the collaboration in the case of the *POS* system.

The participants from **Figure 1.12** do not indicate instances of some types from the model. This time, the main participants from Figure 1.13 are objects of type *controller*, *service* and *repository*, according to the *POS* architecture. The diagram presents the messages addressed to controller (1, 3, 6 and 9) because in the diagram it is not presented the user interface element.

As the participants are objects, the sent messages will be methods calls for the objects to which the messages are sent. So, the diagram lead us to the identification of the objects' methods. Figure 1.14 present the methods identified based on the interactions from Figure 1.13.

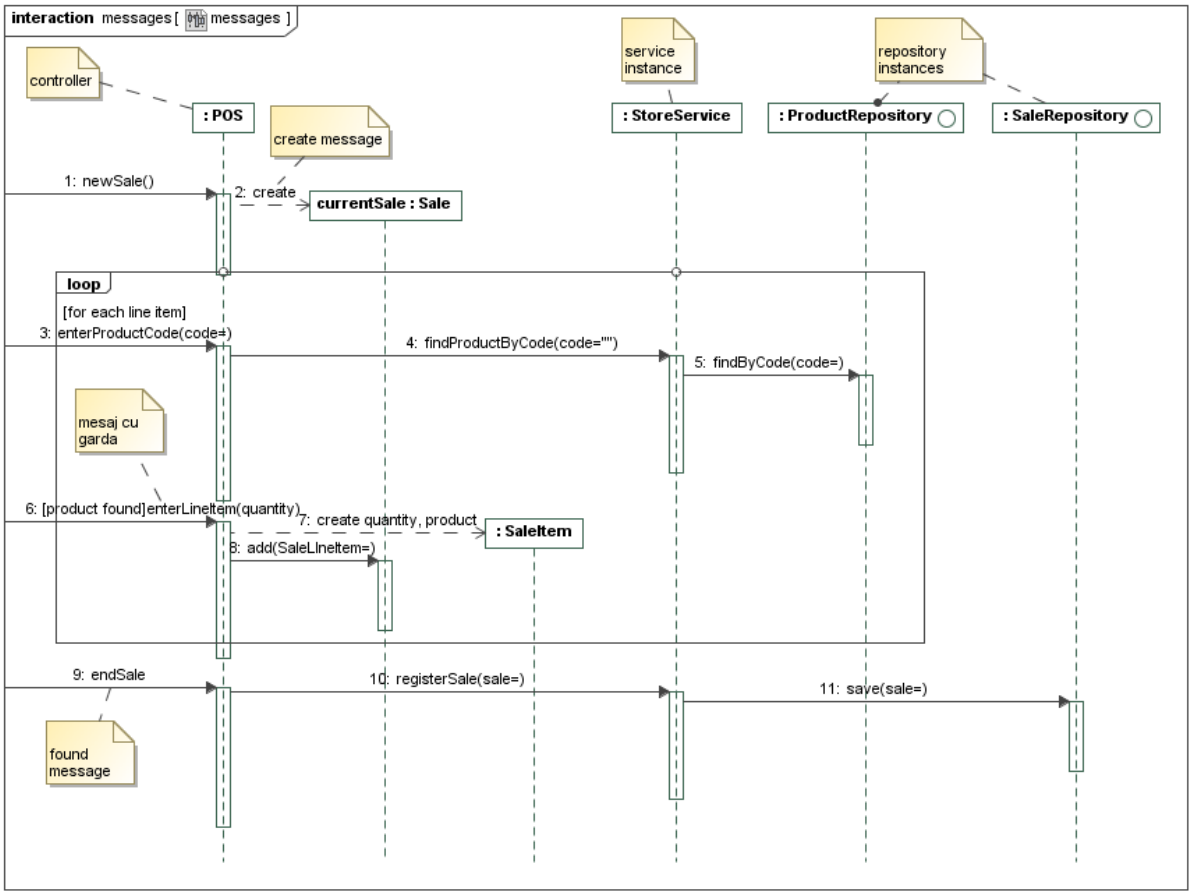


Figure 1.13 Detailed design – interactions between objects

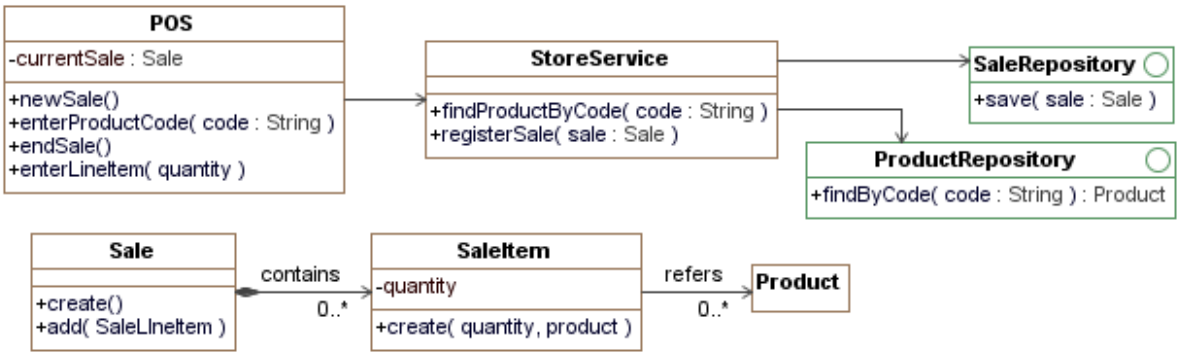


Figure 1.14 Detailed design – class diagram

1.5 Lists and maps

In the following, we present two containers commonly used in programming and these are ***lists*** and ***maps***. The corresponding data types are specified and also the specific operations. For each operation in the interface of a data type, we will specify the operation in natural language, indicating the input and the preconditions of the operation (**pre**), as well as the output and the postconditions of the operation (**post**).

1.5.1. Lists

In common speaking, the word "list" refers to a "listing, in some order, the names of people and objects, data, etc.." There are many examples of lists: shopping list, price list, list of students, etc.. The order of the list can be interpreted as a kind of "connection" between list items (after the first purchase is the second purchase after the second purchase is the third purchase, etc.) or can be seen as being given by the index number of the item in the list (the first purchase, the second purchase, etc). `List` data type that will be further defined allows implementation in applications related to these real world situations

Therefore, we can see a ***list*** as a sequence of elements $< l_1, l_2, .., l_n >$ of the same type (`TElement`) that are in a certain order. Each element has a well established *position* in the list. As a result, the position of elements in the list is essential; access, deletion and addition can be made based on a position in the list.

A ***list*** can be seen as a dynamic collection of elements in which the order of elements is essential. The number *n* of elements in the list is called *the length* of the list. A list of length 0 will be called the *empty* list. The dynamic character of the list is given by the factt that the list may change over time with additions and deletions l of elements to/from the list.

In the following, we will refer to linear lists. A linear list is a structure that is either empty (has no item) or

- has a unique first element;
- has a unique last element;
- are un unic ultim element;

- each item in the list (except the last element) has one successor;
- each item in the list (except the first element) has one predecessor.

Therefore, in a linear list we can insert items, delete items, we can determine the successor (predecessor) of an element, you can access an element based on its ***position*** in the list.

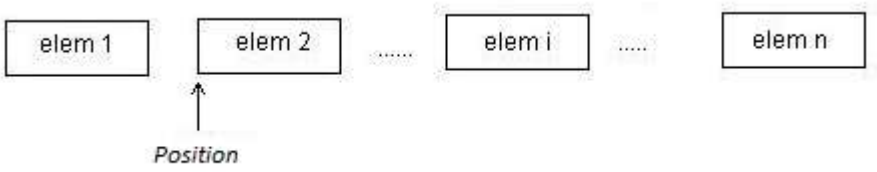
A linear list is called ***circular*** if it is considered that the predecessor of the first node is the last node and last node successor is the first node.

As defined above, each element of a linear list has a well established position in the list. The first position in the list it is distinctive. Given a list and a position in the list, we can identify the element on this position, the position of the predecessor and the position of the successor element in the list (if any). Therefore, an ordering over elements positions in a list can be established.

Position of an element in a list can be viewed in different ways:

1. given by ***rank*** (order number) item in the list. This case is similar to vectors, the position of an item in the list is its index in the list. In such an approach, list is seen as a dynamic vector that can access / add / delete elements to any position in the list.
2. given by a ***reference*** to the location where the element is stored (eg.: pointer to the location where the element is stored)

To ensure generality, we abstract the notion of ***position*** of an element in the list. We assume that the list items are accessed via a generic ***position***.



We say that a position p in a list is ***valid*** if it is the position of a list element. For example, if *p* were a pointer to the location the list element is stored, then *p* is ***valid*** if it is different to null pointer or any other address which is not the memory address of a list element. If *p* were the rank (index number) of the element in the list, then *p* is ***valid*** if it does not exceed the number of elements in the list.

Therefore, if we consider a linear list in which operations access / insert / delete are made based on a generic ***position*** in list, we can define the next abstract data type.

Empty list will be denoted by ϕ.

A special position value that is not valid inside a list will be denoted by $\perp$.

`Abstract Data Type LIST`

domain

$\mathcal{D}$List(*Position;* **`TElement`**) = {*l* | *l* is a list of elements of type **`TElement`**;
each element has a position of type *Position*}

In what follows, we will use the notation:

$\mathcal{L}$= $\mathcal{D}$ List(*Position;* `TElement`**)**

operations

create (*l*) // ***initEmpty***, ***createEmpty***

desc.: create an empty list

pre: true

post: $l \in \mathcal{L}$, $l = \phi$

valid(*l*, *p*)

desc.: function that verifies if a given position is valid in a list

pre: **l**$\in \mathcal{L}$, *p*∈*Position*

post: *valid*= true if exists an element *e* in list *l* and *p* is the position of element *e*

false otherwise

addLast (*l*, *e*)

desc.: add an element to the end of the list

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

post: $l' \in \mathcal{L}$, *l'* is *l* modified by adding *e* at its end

addFirst (*l*, *e*)

desc.: add an element at the beginning of a list

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

post: $l' \in \mathcal{L}$, *l' is l* is modified by adding *e* as first element

addBefore (*l*, *p*, *e*) // ***insertBefore***

desc.: add an element before a given position in list

pre: $l \in \mathcal{L}$, *e*∈`TElement`, *p*∈*Position*, ***valid***(*l*, *p*)

post: $l' \in \mathcal{L}$, *l'* is *l* updated by inserting *e* before position *p*

addAfter (*l*, *p*, *e*) // ***insertAfter***

desc.: add an element before a given position in list

pre: $l \in \mathcal{L}$, *e*∈`TElement`, *p*∈*Position*, ***valid***(*l*, *p*)

post: $l' \in \mathcal{L}$, *l'* is *l* updated by inserting *e* after position *p*

delete (*l*, *p*, *e*) // ***remove***

desc.: delete the element from the list; the element is given by its position

pre: $l \in \mathcal{L}$, *e*∈`TElement`, *p*∈*Position*, ***valid***(*l*, *p*)

post: *e*∈`TElement`, $l' \in \mathcal{L}$, *l'* is *l* updated by removing the element on position p, *e* is the removed element from the list *l*

getElement (*l*, *p*, *e*) // ***element***

desc.: access the element on a given position

pre: $l \in \mathcal{L}$, $e \in$ TElement, $p \in$ *Position*, ***valid***(l, p)

post: $e \in$ TElement, e is the element on position p from l

setElement (l, p, e) // ***modify***

descriere: modify the element on a given position in a list

pre: $l \in \mathcal{L}$, $e \in$ TElement, $p \in$ *Position*, ***valid***(l, p)

post: $l \in \mathcal{L}$, l is updated by replacing the element on position p with e

first (l) ***// getFirstPosition***

desc.: function that returns the position of first element in the list

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

post: *first* $\in$ Position

first = $\perp$ if l is empty list

the position of the first element in l, if l is not empty

last (l) // ***getLastPosition***

desc.: function that returns the position of last element in the list

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

post: *last* $\in$ Position

last = $\perp$ if l is empty list

the position of the last element in l, if l is not empty

next (l, p) // ***getNext***

desc.: function that returns the position that is the next to a given position in a list

pre: $l \in \mathcal{L}$, $p \in$ *Position*, ***valid***(l, p)

post: *next* $\in$ Position

next = $\perp$ if p – is the last position in the list l

the position in l that follows after p, otherwise

previous(l, p) // ***getPrev***

desc.: function that returns the position that is previous to a given position in a list

pre: $l \in \mathcal{L}$, $p \in$ *Position*, ***valid***(l, p)

post: *previous* $\in$ Position

previous = $\perp$ if p – is the first position in the list l

the position in l that precedes p, otherwise

search (l, e)

desc.: function that search for an element in a list

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

post: *search* $\in$ Position

search = $\perp$ if $e \notin l$

the first position on which e appear in list l

isIn (l, e)

desc.: function that verify if an element is in a list

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

post: *isIn* = true if $e \in l$
false otherwise

isEmpty (*l*) // ***empty***

desc.: function that verify if the list is empty

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

post: ***isEmpty*** = true if $l = \phi$
false otherwise

size(l) // ***length***

desc.: function that returns the number of elements in a list

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

post: *size* = the number of elements in the list l

getIterator(*l*, *i*) // ***iterator***

desc.: build an iterator over a list

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

post: *i* is an iterator over list *l*

destroy (*l*)

desc.: destroy a list

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

post: list *l* was destroyed

We can be print a list (as any other container that can be iterated) by using the iterator built by using the operation ***getIterator*** from list interface.

```
Subalgorithm print (l):
{pre: l is a list }
{post: the elements of the list are printed }
     getIterator(l, i)           {get the iterator on the list l}
     While valid(i) do       {while the iterator is valid }
          currentElement(i, e)         {e is the current element)
                                {reffered by iterator}
          @ print e           {print the current element }
          next(i)             {iterator refers to the next element }
     endWhile
end-print
```

Remark

We mention that there is no one unique style approved to specify operations. For example, for operation *addLast* (of ADT List), an equivalent and also a correct specification would be one of the following:

addLast (*l*, *e*)

desc.: add an element to the end of the list

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

post: $l' \in \mathcal{L}$, $l' = l \cup \{e\}$, e is on the last position on l'

addLast (*l*, *e*)

desc.: add an element to the end of the list

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

post: $l \in \mathcal{L}$, l is modified by adding e at its end and preserving all the other elements on their positions

1.5.2. Maps

Maps are containers that contains pairs (key, value). Maps store items so they can be easily located using keys. Basic operations on dictionaries are search, add and delete items. In a map, keys are unique and a key has an unique associated value.

Maps have multiple applications. For example:

- Information on bank accounts: each account is an object identified by an account number (considered the key) and additional information (account holder's name and address, information about deposits, etc.). Additional information will be values.
- Information about telephone subscribers: each subscriber is an object identified by an telephone number (considered the key) and additional information (name and address of the subscriber, auxiliary information, etc.). Additional information will be values.
- Information about students: each student is an object identified by a number (considered the key) and additional information (student name and address, auxiliary information, etc.). Additional information will be values.

In the following, we define an Abstract Data Type **Map**.

We will use **TKey** to denote the domain of the first element in a map pair.

We will use **TValue** to denote the domain of the second element in a map pair and we will add there a very special value, denoted $\perp$. This special value will be used to denote no valid values that would be of type TValue. This is similar to NULL or NIL for pointers. So: **TValue** = $\{\perp\} \cup$ the set of all possible valid values for the second element in a pair.

`Abstract Data Type MAP`

domain

$$\mathcal{D}\text{Map}(\text{TKey}, \text{TValue}) = \{ \mathbf{m} \mid \mathbf{m} \text{ is a container containing pairs } (\mathbf{k},\mathbf{v}), \text{ where } \mathbf{k}: \mathbf{TKey}, \mathbf{v}:\mathbf{TValue} \text{ and } \mathbf{k} \text{ – is unique in container } \mathbf{d}\}$$

In what follows, we will use the notation:

$\mathcal{D}$ = $\mathcal{D}$**Map(TKey, TValue)**

operations

create(*d*) // ***initEmpty***, ***createEmpty***

desc.: create an empty map

pre: true

post: $d \in \mathcal{D}$, *d* is empty

add (*d*, *c*, *v*) // ***put***

desc.: add a pair to the map

pre: $d \in \mathcal{D}$, $c \in$ **TKey,** $v \in$ **TValue**

post: $d' \in \mathcal{D}$, $d' = d \cup \{c, v\}$ (add pair (*c*, *v*) to the map)

search(*d*, *c*) // ***searchByKey***

desc.: search for an element in the map (by key)

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

post: *search* $\in$ **TValue**

$$search = \begin{cases} v & \text{if } (c,v) \in d \\ \perp & \text{otherwise} \end{cases}$$

remove(*d*, *c*)

desc.: remove an element from a map (given the key)

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

post: *remove* $\in$ **TValue**

$$remove = \begin{cases} v & \text{if } (c,v) \in d \\ \perp & \text{otherwise} \end{cases}$$

$$d' \quad \begin{cases} \text{is } d \text{ modified by removing the pair with key } c & \text{if } (c,v) \in d \\ \text{is unmodified} & \text{otherwise} \end{cases}$$

size (*d*)

desc.: function that returns the number of elements in a map

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

post: ***size*****=** the number of pairs in the map $d \in \mathcal{N}$

isEmpty(*d*) // ***empty***

desc.: function that verify if the map is empty

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

post: ***isEmpty*** **=** $\begin{cases} \text{true} & \text{if the map is empty} \\ \text{false} & \text{otherwise} \end{cases}$

keySet (*d*, *m*) // ***keys***

desc.: determine the set of keys in a map

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

post: $m \in \mathcal{M}$, *m* is the set of keys in map *d*

values (*d*, *c*) // ***valueMultiset***

desc.: determine the collection of values in a map

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

post: $c \in \mathcal{Col}$, *c* is the collection of values in the map *d*

pairs (*d*, *m*)

desc.: determine the set of pairs (key, value) in a map

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

post: $m \in \mathcal{M}$, m is the set of pairs (key, value) in the map *d*

getIterator (*d*, *i*) // ***iterator***

desc.: .: build an iterator over a map

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

post: $i \in \mathcal{I}$, *i* is an iterator over map *d*

destroy (*d*)

desc.: destroy a map

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

post: the map *d* was destroyed

We can be print a map (as any other container that can be iterated) by using the iterator built by using the operation ***getIterator*** from map interface.

```
Subalgorithm print (m):
{pre: m is a map }
{post: the elements of the map are printed }
     getIterator(m, i)           {get the iterator on the map m }
     While valid(i) do       {while the iterator is valid }
          currentElement(i, e)         {e is the current element)
                                {reffered by iterator}
          @ print e           {print the current element }
          next(i)             {iterator refers to the next element }
     endWhile
end-print
```

1.6 Proposed problems

I. Write a program in one of the programming languages Python, C++, Java, C# that:

a. Define a class **B** with an integer attribute ***b*** and a printing method that displays the attribute ***b*** to the standard output.
b. Define a class **D** derived from **B** with an attribte ***d*** of type string and also a method of printing on standard output that displays the attribute ***b*** of the base class and the attribute ***d***.
c. Define a function that builds a list containing: an object $\mathbf{o_1}$ of type **B** having ***b*** equal to 8; an object $\mathbf{o_2}$ of type **D** having ***b*** equal to 5 and ***d*** equal to ″D5″; an object $\mathbf{o_3}$ of type **B** having ***b*** equal to -3; an object $\mathbf{o_4}$ of type **D** having ***b*** equal to 9 and ***d*** equal to ″D9″.
d. Define a function that receives a List of objects of type **B** and returns a List containing only the items that satisfy the property: ***b*>6**.
e. For data type **List** used in the program, write the specifications of the used operations.

You can use existing libraries for data structures (Python, C++, Java, C#). It is not required to implement the list operations/methods.

II. Write a program in one of the Python, C++, Java, C# languages which:
a. Defines a class **B** having an attribute ***b*** of type integer and a display method, which displays the attribute ***b*** to the standard output.
b. Defines a class **D** derived from **B** having an attribute ***d*** of type string and also a method to display on the standard output attribute ***b*** from the base class and attribute ***d***.
c. Define a function which builds a map which contains: an object $\mathbf{o_1}$ of type **B** having ***b*** equal to 8; an object $\mathbf{o_2}$ of type **D** having ***b*** equal to 5 and ***d*** equal to „D5”; an object $\mathbf{o_3}$ of type **B** having ***b*** equal to -3; an object $\mathbf{o_4}$ of type **D** having ***b*** equal to 9 and ***d*** equal to „D9”. (the *key* of an object from the map is the value of ***b***, and the *value* associated to this key is the object itself.)
d. Define a function which receives a map with objects of type **B** and verifies whether in the map there is an object which satisfies the property: ***b*>6**.
e. For the **map** abstract data type used in the program, write the specifications of the used operations.

You can use existing libraries for data structures (Python, C++, Java, C#). It is not required to implement the methods of the map.

III. The subject will present a **class diagram** and an **object interaction diagram** and you will be required **to write a program** which corresponds to the given diagrams.

The program may be written in an object oriented programming language, ex. Python, Java, C++ sau C#.

2. Databases

2.1. Relational databases. The first three normal forms of a relation

2.1.1. Relational model

The relational model of databases was introduced by E.F.Codd in 1970 and it is the most studied and more used model of organizing databases. Below is a brief overview of this model.

Consider $A_1, A_2, ..., A_n$ a set of attributes (or columns, fields, data names, etc.) and $\mathbf{D_i = Dom(A_i) \cup \{?\}}$ the domain of possible values of attribute A_i, where by "**?**" is denoted "*undefined*" (*null*) value. "*Undefined*" value is used to specify if an attribute has a value assigned or it has no value. This value does not belong to a specific data type and it could be used in expressions together with other attribute values having different types (number, text, date, etc.).

Using the already defined domains, we could define **a relation of degree (arity) n** as below:

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

and it could be considered as a set of vectors with **n** values, one value for each attribute $\mathbf{A_i}$. Such a relation could be stored in a table as follows:

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

where lines represent **relation elements**, or **tuples**, or **records**, which are distinct, and $a_{ij} \in D_j$, $j = 1,...,n$, $i = 1,...,m$. Because the way of representing the elements of a relation ***R*** is similar with a table, the relationship is called **table**. In order to emphasize relation (table) name and the list of attributes we will denote the relation with:

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

A **relation instance ([R])** is a subset of D_1 X D_2 X … X D_n.

A **relational database** is a set of relations. A **database schema** is the set of schemas of the relations in the database. A **database instance** (**state**) is the set of instances of the relations in the database.

A **relational database** has three parts:

1. **Data** (relations content) and its description;
2. **Integrity constraints** (to preserve database consistency);

3. Data **management operators**.

Sample 1. STUDENTS [NAME, BIRTH_YEAR, YEAR_OF_STUDY],

with the following possible values:

NAME	BIRTH YEAR	YEAR OF STUDY
Pop Ioan	1985	2
Barbu Ana	1987	1
Dan Radu	1986	3

Sample 2. BOOK [AUTHO, TITLE, PUBLISHER, YEAR],

with values:

AUTHOR	TITLE	PUBLISHER	YEAR
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

A field or a set of fields is a **key** for a relation if there are not two tuples can have same values for all fields and this is not true for any subset of the key.

If is assigned one value for each attribute of the key at most one row could be indentified having those values for key. Because all rows of a relation are distinct, one key could be always identified (the worst case is when the key is composed by all attributes).For example 1 {NAME} could be chosen as key (case when it is not possible to have two students with the same name in the database) and for example 2 the set of attributes {AUTHOR, TITLE, PUBLISHER, YEAR} seems to be the only key (or a new attribute could be added – ISBN – to uniquely identify a book record).

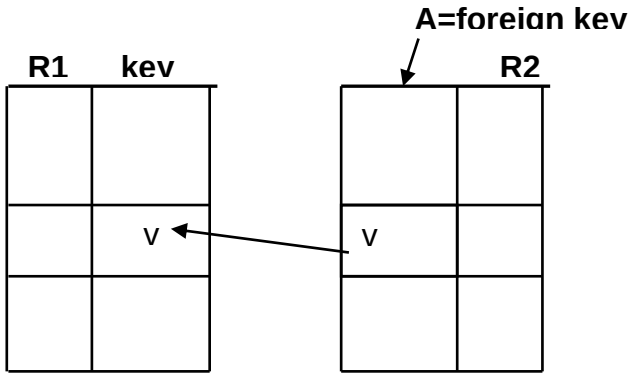
In general a relation may have more keys. One of them is chosen as **primary key**, all the others being considered **secondary keys** (or **key candidates**). Relational database management systems do not allow two or more rows of a relation having the same value for (any) key. A key is an **integrity constraint** for a database.

Sample 3. SCHEDULE [DAY, HOUR, ROOM, PROFESSOR, CLASS, LECTURE],

stores weekly schedules of a faculty. There are three possible keys:

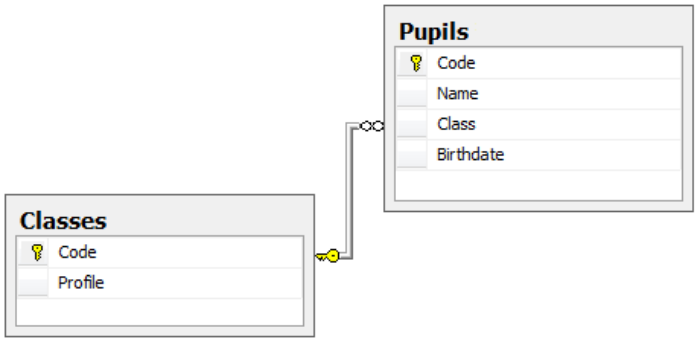
{DAY, HOUR, ROOM}; {DAY, HOUR, PROFESSOR}; {DAY, HOUR, CLASS}.

A **foreign key** is a set of fields in one relation **R1** that is used to `*refer*' a tuple in another relation **R2** (like a `*logical pointer*'). It must correspond to primary key of the second relation. The two relations **R1** and **R2** are not necessary distinct.



Sample:

CLASSES [code, profile]
PUPILS [code, name, clasa, birthdate].

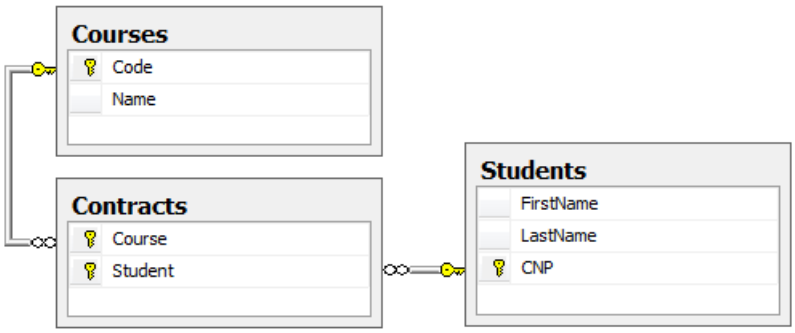


For the above example there is a link between **CLASSES** relation (called *parent relation*) and **PUPILS** relation (called *member relation*) through the equality **CLASSES.Code = PUPILS.Class**. One specific class (stored in CLASSES relation) identified by its code is linked with all pupils (stored in PUPILS relation) that belong to that class.

Using **foreign keys** we can store **1:n relationships** between entities: one class is associated with many pupils, and one pupil is linked to only one class.

A foreign key could be used to store **m:n relationships**, too.

For example, there is a **m:n relationship** between **courses** and **students** entities because there are many students who attend one course and one student attend to many courses during one semester. An m:n relationship is stored using a so called ***cross table*** (see CONTRACTS table in the following example).



Integrity Constraints are conditions that must be true for any instance of the database. Integrity constraints are specified when schema is defined and are checked when relations are modified. A legal instance of a relation is one that satisfies all specified integrity constraints.

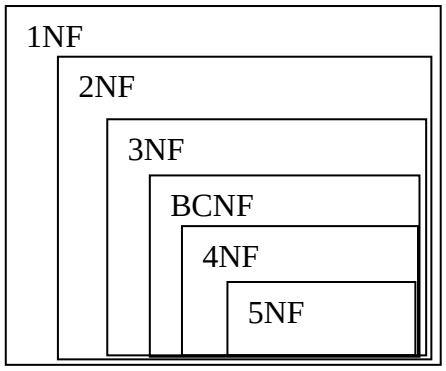
The integrity constraints are related to attributes, relations or the link between relations:

- **Constraints for attributes:**
  - Not Null – the attribute cannot have "undefined" value assigned
  - Primary Key – the current attribute is the primary key of the relation
  - Unique - the current attribute is a key candidate
  - Check(condition) – simple logical conditions which should be true for a consistent database
  - Foreign Key REFERENCES parent_table [(attribute_name)] [On Update *action*] [On Delete *action*] – the current attribute is a reference to a record stored in another relation
- **Constraints for relations:**
  - Primary key(list of attributes) – definition of a composed primary key
  - Unique(list of attributes) - definition of a composed candidate key
  - Check(condition) – condition that involves more than one attribute of a relation
  - Foreign Key foreign_key(attribute_list) REFERENCES parent_table [(attribute_list)] [On Update *action*] [On Delete *action*] – defines a composed reference to a record stored in other table

2.1.2. First three normal forms of a relation

In general, certain data can be represented in several ways through relations (in relational model). Redundancy is at the root of several problems associated with relational schemas: redundant storage, insert/delete/update anomalies. Integrity constraints can be used to identify schemas with such problems and to suggest refinements.

If a relation is in a certain **normal form**, it is known that certain kinds of problems are avoided/minimized. The most used normal forms today are: 1NF, 2NF, 3NF, BCNF, 4NF, 5NF. We have the following inclusion relationships between different normal forms:



If a relation is not in a certain **normal form** , it could be decomposed in many relations which are in that normal form. **Projection** operator is used to decompose a relation and **join natural** operator is used for relations composition

Definition. Given $R[A_1, A_2, ..., A_n]$ a relation and $\alpha = \{A_{i_1}, A_{i_2}, ..., A_{i_p}\}$ a subset of attributes, $\alpha \subset \{A_1, A_2, ..., A_n\}$. The **projection** of relation **R** on **α** is the relation:

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

where:

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

and all element from R' are distinct.

Definition. Given $R[\alpha, \beta]$, $S[\beta, \gamma]$ two relations over the set of attributes α, β, γ, $\alpha \cap \gamma = \emptyset$, by **natural join** of relations **R** and **S** we obtain the relation:

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

A relation **R** could be decomposed in more new (sub)relations $R_1, R_2, ..., R_m$. This decomposition is ***losless-join*** if $R = R_1 * R_2 * ... * R_m$, so the data stored in R is obtained from data stored in $R_1, R_2, ..., R_m$ and there are no new elements (rows) added as a result of composing the sub-relations.

Example of a decomposition which is not lossless-join: give the relation:

StudyContracts[Student ,Professor, Course],

and two new relations obtained as results of applying the projection on ***StudyContracts*** based on the following atttributes: **SP**[Student, Professor] and **PC**[Professor, Course]. Let's suppose that for the initial relation we have the following values::

R	**Student**	**Professor**	**Course**
r_1	s1	p1	c1
r_2	s2	p2	c2
r_3	s1	p2	c3

Using the projection definition we obtain the following instances for the two sub-relations of **R**

SP	**Student**	**Professor**
r_1	s1	p1
r_2	s2	p2
r_3	s1	p2

CD	**Professor**	**Course**
r_1	p1	c1
r_2	p2	c2
r_3	p2	c3

Applying natural join between sub-relation instances we obtain the following relation::

SP*PC	**Student**	**Professor**	**Course**
r_1	s1	p1	c1
r_2	s2	p2	c2
?	s2	p2	c3
?	s1	p2	c2
r_3	s1	p2	c3

The resulting relation, **SP*PC,** contains two new rows which were not part of the initial relation **R**, so the decomposition is not **losless-join**.

Observation. A **simple attribute** is any attribute of a relation, and a **composed attribute** is a set of attributes (at least two) of the same relation.

In some real life applications it is possible to have attributes (simple or composed) with more than one values assigned for the same element (row) of a relation. Such attributes are called **repetitive attributes**.

Sample 4. Given relation:

STUDENT [NAME, BIRTHYEAR, GROUP, COURSE, MARK],

having NAME attribute as key. In this example the pair {COURSE, GRADE} is a repetitive composed attribute. We may have the following values in this relation:

NAME	BIRTHYEAR	GROUP	COURSE	MARK
Pop Ioan	1998	221	Databases	10
			Operation Systems	9
			Probabilities	8
Mureşan Ana	1999	222	Databases	8
			Operating Systems	7
			Probabilities	10
			Project	9

Sample5. Given relation:

BOOK [ISBN, AuthorNames, Title, Publisher, Year, Language, KeyWords],

with **ISBN** as key and the following repetitive attributes **AuthorNames** and **KeyWords**.

Repetitive attributes bring a lot of storing and maintenance issues. This is the reason why there are solutions found to **avoid** their usage, but without losing data. If **R[A]** is a relation, where **A** is the set of attributes and **α** is a repetitive attribute (simple or composed), then **R** could be decomposed in two relations with no repetitive attributes. If **C** is a key of **R**, then the two sub-relations of **R** are**:**

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

Sample 6. Relation STUDENT from sample 4 is decomposed in 2 relations:

GENERAL_DATA [NAME, BIRTHYEAR, GROUP],
RESULTS [NAME, COURSE, MARK].

Sample7. Relation BOOK from sample 5 is decomposed in 3 relations (BOOK relation has 2 repetitive attributes):

BOOKS [ISBN, Title, Publisher, Year, Language],
AUTHORS [ISBN, AuthorName],
KEY_WORDS [ISBN, KeyWord].

Observation. If a book does not have authors or key words assigned, than it will exist one corresponding record in both AUTHORS and KEY_WORDS relations, but for each record the value of the second attribute will be **null**. If these records are removed then we cannot recompose the **BOOK** relation using natural join (we should use outer join operator).

Definition. A relation is in **first normal form (1NF)** if it does not contain **repetitive attributes** (simple or composed).

Most of relational database management systems allow only creation of relations which are in 1NF. There are systems where repetitive attributes are permitted (for instance Oracle, where a column could be an *object* or a *data collection*).

Next two normal forms are defined based on ***functional dependency*** concept which describes a relationship between two sets of attributes. Database designer defines the functional dependencies for all relations in a database, having in mind the meaning of data stored in each relation. All update, insert or remove operations executed over a relation should preserve functional dependencies identified by designer.

Definition. Consider relation $R[A_1, A_2, ..., A_n]$ and two sets of attributes $\alpha, \beta \subset \{A_1, A_2, ..., A_n\}$. The attribute (simple or composed) β is **functional dependent** by the attribute α (simple or composed), denoted by: $\alpha \rightarrow \beta$, if and only if each value of α from **R** has associated a **unique and precise value** of β (this association is valid during the whole life of relation **R**). A particular value of α could appear in many rows of **R**, case when the attribute β will have the same value for atose rows, which means:

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

Observation. A functional dependency could be used as a **property (restriction)** whichshould be preserved by any relation instance of a database: elements are inserted, removed or changed for a relation only if the functional dependency is checked.

The existence of a functional dependency in a relation implies **redundancy** in cases where "pairs" of the same values are stored many times in that relation. Let's consider the following example:

Sample 8. EXAM [<u>StudentName, CourseName</u>, Mark, Professor],

Where the key is {*StudentName, CourseName*}. Because one course is associated with only one professor, and one professor could have many courses we could enforce the following functional dependency: *{CourseName}→{Professor}*.

Exam	StudentName	CourseName	Mark	Professor
1	Alb Ana	Mathematics	10	Rus Teodor
2	Costin Constantin	History	9	Popa Horea
3	Alb Ana	History	8	Popa Horea
4	Enisei Elena	Mathematics	9	Rus Teodor
5	Frişan Florin	Mathematics	10	Rus Teodor

Preserving such a functional dependency inside our relation EXAM we can meet one of the following issues:

- **Extra-memory spent**: the same dependences are stored many times. The link between course *Mathematics* and professor *Rus Teodor* is stored three times in our example, and that between *History* course and professor *Popa Horea* is stored two times.
- **Update anomaly**: changing a value which appears in an association implies performing the changes in all associations that value appears (without knowing how many such associations exists) or the database will store inconsistent data. In our example, if the professor changes for the first row, we have to be careful of the updates for rows 4 and 5, otherwise the initial defined functional dependency is not preserved.

- **Insert anomaly**: we cannot add a new row in a relation if we do not know the values for all corresponding attributes (we cannot use *undefined* values for the attributes involved in a functional dependency).
- **Remove anomaly**: removing rows from a relation we can remove associations between attributes that cannot be restored afterwards. In our example, is rows 2 and 3 are removed, than the association between *History* and *Popa Horea* is lost.

The above anomalies are generated by the existence of a functional dependency between sets of attributes. We can remove these anomalies if we will preserve the attributes involved in a functional dependency in separate relations. To achieve this, the initial relation should be (lossless-join) decomposed in 2 or more sub-relations. Such lossless-join decomposition is done at the design phase, when the functional dependencies are identified.

Observations. It is easy to prove the following simple properties of functional dependencies:

1. If C este is key for $R[A_1, A_2, ..., A_n]$, then $C \to \beta, \forall \beta \subset \{A_1, A_2, ..., A_n\}$.
2. If $\beta \subseteq \alpha$, then $\alpha \to \beta$, called **trivial functional dependency** or **reflexivity**.
$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\beta \subset \alpha}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \alpha \to \beta$$
3. If $\alpha \to \beta$, then $\gamma \to \beta, \forall \gamma$ with $\alpha \subset \gamma$.
$$\Pi_\gamma(r_1) = \Pi_\gamma(r_2) \underset{\alpha \subset \gamma}{\Rightarrow} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \Rightarrow \gamma \to \beta$$
4. If $\alpha \to \beta$ and $\beta \to \gamma$, then $\alpha \to \gamma$, which is called **transitivity property** of functional dependencies.
$$\Pi_\alpha(r_1) = \Pi_\alpha(r_2) \underset{\alpha \to \beta}{\Rightarrow} \Pi_\beta(r_1) = \Pi_\beta(r_2) \underset{\beta \to \gamma}{\Rightarrow} \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \Rightarrow \alpha \to \gamma$$
5. If $\alpha \to \beta$ and $\gamma \subset A$, then $\alpha\gamma \to \beta\gamma$, where $\alpha\gamma = \alpha \cup \gamma$.
$$\Pi_{\alpha\gamma}(r_1) = \Pi_{\alpha\gamma}(r_2) \Rightarrow \left| \begin{array}{c} \Pi_\alpha(r_1) = \Pi_\alpha(r_2) \Rightarrow \Pi_\beta(r_1) = \Pi_\beta(r_2) \\ \Pi_\gamma(r_1) = \Pi_\gamma(r_2) \end{array} \right. \Rightarrow \Pi_{\beta\gamma}(r_1) = \Pi_{\beta\gamma}(r_2)$$

Definition. An attribute A (simple or composed) is called **prime** if there is a key C and $A \subset C$ (C is a composed key). An attribute is **non-prime** if it is not included in any key of a relation.

Definition. Consider relation $R[A_1, A_2, ..., A_n]$ şi $\alpha, \beta \subset \{A_1, A_2, ..., A_n\}$. Attribute β is **complete functional dependent** on α if β is functionally dependent on α (so $\alpha \to \beta$) and it is not functionally dependent on any subset of attributes of α ($\forall \gamma \subset \alpha, \delta \to \beta$ is not true).

Objsrvation. If β **is not complete functional dependent** on α (so it is dependent on a subset of α), then α is a **composed attribute**.

Definition. A relation is in the **second normal form (2NF)** if:
- **It is in the first normal form**,
- all **non-prime attributes** (simple or composed) (so those which are not part of a key) **is complete functional dependent on any key of the relation**.

Observation. If a relation is in the first normal form (1NF) and it is not in the second normal form (2NF) then it has a **composed key** (if a relation is not in the second normal form, then there is a functional dependency $\alpha \to \beta$ with α included in one key).

The general **decomposition** process could be defined considering a relation $R[A_1, A_2,...,A_n]$ and a key $C \subset A = \{A_1, A_2,...,A_n\}$. Let's suppose that there is a non-key attribute $\beta \subset A$, $\beta \cap C = \emptyset$, β being functional dependent on $\alpha \subset C$ (β is complete functional dependent on a strict subset of key). The dependency $\alpha \to \beta$ could be handled if **R** is decomposed in following two relations:

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

In sample 8:

EXAM [<u>StudentName, CourseName</u>, Mark, Professor],

where the key is {*StudentName, CourseName*} and with functional dependency (restriction) *{CourseName}→{Professor}*. We deduce that *Professor* attribute is not complete functional dependent on the key, so relation EXAM is not in the second normal form. This functional dependency could be handled by decomposing EXAM in 2 sub-relations::

CATALOG [StudentName, CourseName, Mark];
COURSES [CourseName, Professor].

Sample 9. Consider the following relation for storing study contracts:

CONTRACTS [LastName, FirstName, <u>CNP, CourseCode</u>, CourseName].

The key of the relation is *{CNP,CourseCode}*. In this relation 2 functional dependencies could be identified:

{CNP} $\to$ *{LastName, FirstName}* and *{CourseCode}*$\to$*{CourseName}.*

To handle these functional dependencies the initial relation could be decomposed in the following sub-relations:

STUDENTS [CNP, LastName, FirstName],
COURSES [CourseCode, CourseName],
CONTRACTS [CNP, CourseCode].

In order to define the third normal form we need to introduce the concept of **transitive dependency.**.

Definition. An attribute **Z** is **transitive dependent** on attribute **X** if $\exists$ **Y** so that $X \to Y$, $Y \to Z$, where $Y \to X$ is not true, and Z is not included in $X \cup Y$.

Definition. A relation is in the **third normal form (3NF)** if it is in **2NF** and **all non-prime attributes are not transitive dependent on any relation key**.

If **C is a key** and β an attribute transitive dependent on C, then there is an attribute α for which: $C \to \alpha$ (always true, because C is a key) and $\alpha \to \beta$. Because the relation is in 2NF, we deduce that β is complete functional dependent on **C**, so $\alpha \not\subset C$. We deduce that a relation which is in 2NF and is not in 3NF has a functional dependency like $\alpha \to \beta$, where α is an non-prime attribute. This dependency could be handled by decomposing the relation R in a similar way as in case of handling dependencies for 2NF.

Sample 10. The marks obtained by students at license dissertation are stored in the following relation::

LICENSE_DISSERTATION [<u>StudentName</u>, Mark, Coordinator, Department].

Here are stored the name of the dissertation coordinator and the department where he/she belongs. Because in the relation we will have at most one row for each student we can establish *StudentNam* as the key of the relation. Based on the meaning of the attributes of the relation we can identify the following functional dependency:

$$\{Coordinator\} \rightarrow \{Department\}.$$

Having this functional dependency between the attributes of LICENSE_DISSERTATION, the relation **is not in 3NF**. In order to handle the functional dependency, the relation could be decomposed in the following two sub-relations:

RESULTS [StudentName, Mark, Coordinator]
COORDINTAORS [Coordinator, Department].

Sample 11. The addresses of a group of persons are stored in a relation having the following structure:

ADDRESSES [CNP, LastName, FirstName, ZipPostal, City, Street, No].

The key of the relation is {CNP}. Because for some cities the ZipCode is defined at street level we have the following functional dependency that holds for ADDRESSES relation:

$$\{ZipPostal\} \rightarrow \{City\}.$$

The existence of this functional dependency, make the relation ADDRESSES to not be in the third normal form, so it is necessary to decompose the relation in 2 sub-relations.

Sample 12. Consider a relation that stores a schedule for student exams, as follows:

SCHEDULE_EX [Date, Hour, Professor, Room, Group],

Having the following restrictions:

1. A student attend to at most one exam per day, so {*Group, Date*} is key.
2. A professor has a single exam with one group at a specific date and time, so {*Professor, Date, Hour*} is key.
3. At one moment in time in a room is scheduled at most one exam, so {*Room, Date, Hour*} is key.
4. A professor does not change the room in one day. Anyway, other exams can be scheduled in the same room but at different hours, so we have the following functional dependency:

$$\{Professor, Date\} \rightarrow \{Room\}$$

All attributes of **SCHEDULE_EX** relation appear in at least one key, so there are no non-prime attributes. Considering the definition of normal forms given so far, we can say that the relation **is in 3NF**. In order to handle also the functional dependencies similar with that at point 4 above, a new normal form is introduced:

Definition. A relation is in **Boyce-Codd 3NF,** or **BCNF,** if the attributes from the left hand side in a functional dependency is a key, so there is no functional dependency $\alpha \rightarrow \beta$ so that α is not key.

To handle the functional dependency identified above, we should decompose **SCHEDULE_EX** table as follows:

SCHEDULE_EX [Date, Professor, Hour, Student],

PLAN_ROOMS [Professor, Date, Room].

After performing this decomposition we obtain two relations which are in BCNF. Unfortunately, we also removed the restriction specified at point 3 above, so: {*Room, Date, Hour*} is not anymore a key (part of the attributes belong to one sub-relation and the other part belong to the second sub-relation). If this restriction is still needed, it could be checked using a different way (direct from the application, for instance).

2.2. Querying databases using relational algebra operators

Below is a list of **condition types** that appear in definition of different relational operators.

1. To verify if an attribute satisfies a simple condition, it could be compared with a specific value like:

   attribute name ***relational_operator*** value

2. A relation with a single column could be viewed as a set of elements. Next condition tests if a particular value belongs to a specific set:

   $$\text{attribute\_name} \left\{ \begin{array}{c} IS\ IN \\ IS\ NOT\ IN \end{array} \right\} \text{relation\_with\_a\_column}$$

3. Two relations (considered as sets of records or rows) could be compared using equality, different, inclusion or non-inclusion operators. Between two relations having the same number of columns and with the same types for corresponding columns we can have types of conditions:

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

4. Also, we could consider the following conditions, using boolean operators:

   (condition)
   NOT condition
   condition1 AND condition2
   condition1 OR condition2

where *condition*, *condition1*, *condition2* are any conditions of type 1-4.

To describe condition type 1 we used the concept *'value'*, which could be one of the following types:

- **attribute_name** – specifies the value of an attribute contained in the current record. If the attribute reference is ambiguous (there are more than one relation containing an attribute with the same name)), then we will add the relation name in front of attribute name: **relation.attribute**.
- **expression** – an expression contains values and operator and they are evaluated base don the current records of queried relations.
- **COUNT(*) FROM relation** – specifies the number of records of the specified relation.

- $\left\{ \begin{matrix} COUNT \\ SUM \\ AVG \\ MAX \\ MIN \end{matrix} \right\}([DISTINCT]\ attribute\_name)$ - identifies a value from a set of values based on all records of the current relation. The value is calculated using all attribute values specified as argument (from all records), or only the distinct values (depending on the presence of DISTINCT word). The resulting values represent: the total number of values (for COUNT), sum of values (for SUM, all values should be numeric), average (for AVG, all values should be numeric), maximum value (for MAX), or the minimum value (for MIN).

For **querying relational databases** we can use the following operators:

- **Selection** (or horizontal projection) of a relation **R** – returns a new relation with the same schema as **R**. From **R** only those rows that satisfy a specific condition **c** will be part of the result. Selection operator is denoted as $\sigma_c(R)$.
- **Projection** (or vertical projection) – returns a new relation which has as attributes a subset **α** of the attributes of the initial relation **R**. The set of attributes **α** could be extended to a set of **expressions** where are specified, besides the expressions, the names of columns for the resulting relation. Projection operator is denoted as: $\prod_{\alpha}(R)$.
- **Cartesian product** of two relations, denoted as $\mathbf{R_1 \times R_2}$, returns a new relation having all attributes of R_1 concatenated with all attributes of R_2, one record in the result being composed by one record from R_1 concatenated with one record from R_2.
- **Union**, **difference** and **intersection** of two relations: $R_1 \cup R_2$, R_1 - R_2, $R_1 \cap R_2$. The two relations should have *the same schema*.
- There are many **join** operators.

  Conditional join or **theta join**, denoted as $R_1 \otimes_{\Theta} R_2$ – returns the records of the Cartesian product of the two relations filtered based on a specific condition. Based on the definition we conclude that: $R_1 \otimes_{\Theta} R_2 = \sigma_{\Theta}(R_1 \times R_2)$.

  Natural join, denoted as $R_1 * R_2$ – returns a new relation having as attributes the union of all attributes of the two relations, and the records are pairs of records from R_1 and R_2 that have the same value for the common attributes. If the two relations have schema $R_1[\alpha], R_2[\beta]$, and $\alpha \cap \beta = \{A_1, A_2, ..., A_n\}$, then the natural join operator is computed based on the following formula::

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

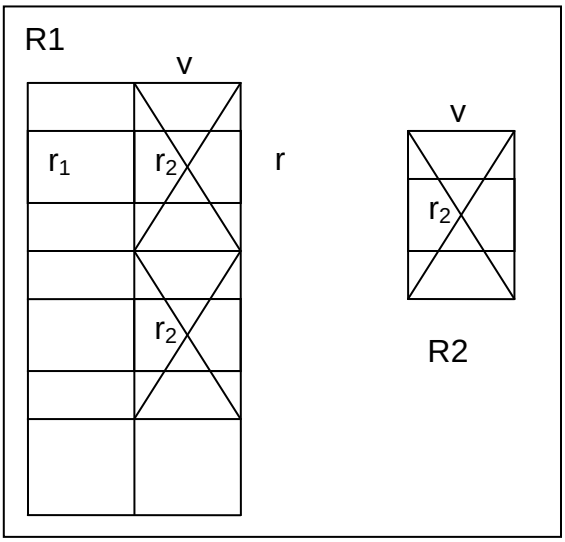
  Left outer join, denoted as $R_1 \triangleright_C R_2$, returns a new relation having as attributes the union of attributes that belong to R_1 and R_2, and the records are obtained like in case of conditional join operator ($R_1 \otimes_c R_2$,) to which are added those records from R_1 which are not returned by the conditional join combined with *null* value assigned to all attributes corresponding to R_2.

Right outer join, denoted as $R_1 \triangleleft_C R_2$, is similar with *left outer join* operator, but to records from $R_1 \otimes_c R_2$ are added all records from R_2 that were not returned by the conditional join combined with *null* value assigned to all attributes corresponding to R_1.

- **Division** is denoted as $R_1 \div R_2$. Let R_1 have 2 set of attributes *x* and *y* and R_2 have only the set of attributes *y (y is a common set of attributes of R_1 and R_1).* $R_1 \div R_2$ *contains all x tuples such that for every y tuple in R_2, there is an xy tuple in R_1*

  In other words, if the set of *y* values associated with an *x* value in R_1 contains all *y* values in R_2, the *x* value is in $R_1 \div R_2$.(see the picture)

  Division is not essential op; just a useful shorthand.



An important problem linked by the relational algebra operators presented above is to establish an **independent subset of operators**. A set of operators M is **independent** if removing an arbitrary operator *op* from M we reduce the power of the set, in other words we can find a relation obtained by using operators from M which cannot be obtained with operators from M - {*op*}.

For the query language used above and independent set is composed by the following operators: $\{\sigma, \prod, \times, \cup, -\}$. All the other operators could be obtained using one of the following rules:

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

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

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

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

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

- If $R_1[\alpha]$, $R2[\beta]$, with $\beta \subset \alpha$, then $r \in R_1 \div R_2$ if $\forall r_2 \in R_1 \div R_2$, $\exists r_1 \in R_1$ so that: $\prod_{\alpha - \beta}(r_1) = r$ şi $\prod_{\beta}(r_1) = r_2$.

Based on this we deduce that **r** is from $\prod_{\alpha-\beta}(R_1)$. In $\left(\prod_{\alpha-\beta}(R_1)\right)\times R_2$ are all elements which have one part in $\prod_{\alpha-\beta}(R_1)$ and the other part in R_2. From the result we remove R_1 what remains are the elements which have one part in $\prod_{\alpha-\beta}(R_1)$ and do not have the other part in $\prod_{\beta}(R_1)$. From here we deduce:

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

To the list of relational operators enumerated so far we can add some useful statements in solving specific problems:

- **Assignment**: a relation obtained as a result of an expression evaluation could be assigned to a variable R. The assignment statement could specify also the column names for R:

  R[list] := expression

- **Removing duplicates** from a relation: $\delta(R)$
- **Sorting** records of a relation: $s_{\{list\}}(R)$
- **Grouping:** $\gamma_{\{list\,1\}\ group\ by\ \{list\,2\}}(R)$, which is an extenssion of projection operator. Records from R are grouped based on columns of *list2*, and for a group of records having the same value for *list2* attributes *list1* is evaluated (list1 could contain also *grouping functions*).

2.3. Querying relational databases using SQL (Select)

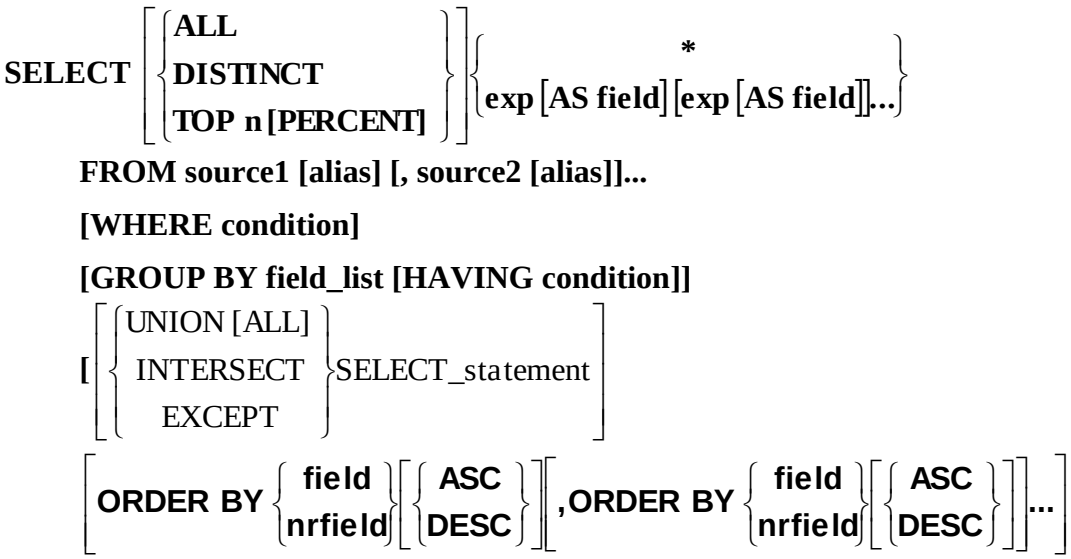
SQL (Structured Query Language) was created for relational databases management. SQL provides statements useful to manage relational database components (like tables, indexes, users, views, stored procedures, triggers etc.).

Short history:

- 1970 - E.F. Codd formalize relational model
- 1974 - at IBM (from San Jose) is defined a new language called SEQUEL (Structured English Query Language)
- 1975 – is defined language SQUARE (Specifying Queries as Relational Expressions).
- 1976 - IBM defines a modified version of SEQUEL, called SEQUEL/2. After first revision it becomes SQL
- 1986 - SQL becomes ANSI (American National Standards Institute) standard
- 1987 - SQL is adopted by ISO (International Standards Organization)
- 1989 – extension SQL89 (or SQL1) is published
- 1992 – after revision it becomes SQL92 (or SQL2)
- 1999 - SQL is updated with object oriented concepts, and becomes SQL3 (or SQL1999)
- 2003 – new data types and functions are defined -> SQL2003.

SELECT statement is used for querying databases and get information based on stored data. SELECT is the most complex and most used statement from SQL. SELECT allows combining data from different data sources. It uses selection, projection, Cartesian product,

join, union, intersection and difference operators uset in relational algebra. SELECT statement syntax is:



The statement selects data from **sources** listed in **FROM** clause. For qualifying fields (if there is ambiguity) we can use the table name or an alias (the **alias** is local for that SELECT statement) defined in **FROM** clause after data source name. If there is an "**alias**" defined we cannot use the original table name in that specific query.

A **source** could be:

1. a **table** or **view** from database
2. (**select_statement**)
3. **Join_expression**, as:
    - **source1 [alias]** ***join_operator*** **source2 [alias] ON** ***link_condition***
    - **(join_expression)**

A **simple condition** between two data sources follows the pattern:

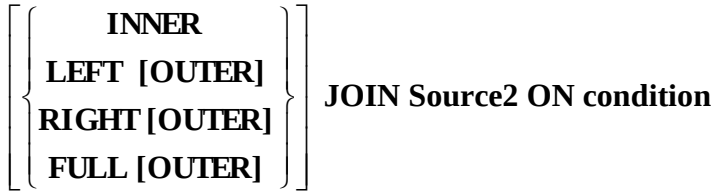
[alias_source1.]field1 ***operator*** **[alias_source2.]field2**

where **operator** could be: **=, <>, >, >=, <, <=.** The two comparison terms should belong to different tables.

A **link condition** between two data sources follows the pattern:

- **Simple_condition [AND simple_condition] ...**
- **(condition)**

A **join expression** generates a table and has the following syntax:



Conditional join from relational algebra, denoted by **Source1** $\otimes_c$ **Source2**, is specified in SQL as **Source1 INNER JOIN Surce2 ON condition,** and returns the records of Cartesian product of the two data sources that verify the condition from **ON** clause.

Left outer join, specified as **Source1 LEFT [OUTER] JOIN Source2 ON condition,** returns a new data source having as attributes the set of attributes from both sources, and the records are obtained as follows: to the records obtained as result of applying the conditional join **Source1** $\otimes_c$ **Source2**, are added those records from din **Source1** which were not returned by the conditional join combined with cu *null* values for all attributes corresponding to **Source2**.

Right outer join, specified as **Source1 RIGHT [OUTER] JOIN Source2 ON condition,** returns a new data source having as attributes the set of attributes from both sources, and the records are obtained as follows: to the records obtained as result of applying the conditional join **Source1** $\otimes_c$ **Source2**, are added those records from din **Source2** which were not returned by the conditional join combined with cu *null* values for all attributes corresponding to **Source1**.

Full outer join, specified as **Source1 FULL [OUTER] JOIN Source2 ON condition,** is obtained by union the tables obtained by applying left outer join and right outer join on **Source1** and **Source2** and satisfying **condition**.

Other join expressions:

- Source1 **JOIN** Source2 **USING** (column_list)
- Source1 **NATURAL JOIN** Source2
- Source1 **CROSS JOIN** Source2 (it returns the Cartesian product between records belonging to Source1 and Source2)

The **FROM** clause allows developers to choose either a table or a result set as a source for specified target list. Multiple sources may be entered following the **FROM** clause, separated by commas. The result of performing a **SELECT** on several comma-delimited sources without a **WHERE** or **JOIN** clause to qualify the relationship between the sources is that the complete Cartesian product of the sources will be returned (similar to a **CROSS JOIN**). This is a result set where each column from each source is combined in every possible combination of rows between each other source.

Typically a WHERE clause is used to define the relationship between comma-delimited FROM sources:

FROM source1[, source2] ... WHERE link_condition

The data sources specified in FROM clause together with the **link conditions** (if exist) will generate s **result set** with columns obtained by concatenation of data sources columns.

The **result set** will contain all records stored in data sources or they can be filtered based a **filtering condition**. This filtering condition is specified in **WHERE** clause, together with the link condition:

WHERE link_condition AND filtering_condition)

The filtering condition from WHERE clause could be built based on the following rules:

A **simple filtering condition** has one of the following formats:

- **expression relational_ operator expression**
- **expression [NOT] BETWEEN minval AND maxval**

  to verify if the value of an expressions is in interval (between *minval* and *maxval*) or is not in such interval (using **NOT**)

- **field (NOT) LIKE pattern**

  After **LIKE** is a *pattern* (expressed by a string) which specifies a set of values. Depending on the database management system, there are specific symbols (chars) used in the *pattern* to specify "*any* char" or "*any* string".

- **expression** $\left[\textbf{NOT}\right]$ **IN** $\left\{\begin{array}{l}\textbf{value}\left[\textbf{value}\right]\textbf{...}\\(\textbf{sub-selection})\end{array}\right\}$

  It is verified if the value of expression is (or is not - using **NOT**) in a list of values or in a sub-selection. A **sub-selection** is a result set generated by a **SELECT** command having only one field – with values of the same type with the value of expression.

- **field relational_ operator** $\left\{\begin{array}{l}\textbf{ALL}\\\textbf{ANY}\\\textbf{SOME}\end{array}\right\}$ **(sub-selection)**

  The value of the field from the left side of the operator should have the same type as the field of sub-selection. The condition is *true* if the value from the left side satisfies the relationship specified by the operator:

  - with all values from sub-selection (**ALL**),
  - with at least one value from sub-selection (for **ANY** or **SOME**).

Equivalent conditions:

"expression IN (sub-selection)" equivalent with "**expression = ANY (sub-selection)"**

"expression NOT IN (sub-selection)" equivalent with "**expression <> ALL (sub-selection)"**

- **[NOT] EXISTS (sub-selection)**

  The condition is *true* (or *false* in presence of **NOT**) if there is at least one record in sub-selection, and *false* (or *true* in presence of **NOT**)if the sub-selection is void.

- A **filtering condition** could be:
  - A simple condition
  - (condition)
  - **not** condition
  - condition1 **and** condition2
  - condition1 **or** condition2

A simple condition could have one of the following values: **true, false, null**. The condition is evaluated with *null* if at least one of the operands is null. Below the evaluation of **not**, **and**, **or** operators is presented for each configuration:

	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	

A **SELECT** statement will return a table containing all fields from all tables (if "*" appears after SELECT), or part of the fields and **expressions**. Fields having the same name in different tables could be qualified withe source table name (or the alias of the source table). The names of fields/expressions are established automatically by the system (depending on the generation expression), or could be explicitly specified through **AS** clause. In this way are built the values of a new record in the **final table**.

Expressions contain operands (fields, values returned by functions) and corresponding operators.

The **final table** will contain all qualifying records or just a part of them, according with the existing predicate after SELECT clause:

- ALL – all records (default predicate)
- DISTINCT – just distinct records
- TOP n - first n records
- TOP n PERCENT - first n% records

Records from "**final table**" can be ascending (ASC) or descending (DESC) ordered based on different fields values, specified in **ORDER BY** clause. The fields could be specified by name or by position (field number) in the field list of **SELECT** command. The order of fields in ORDER BY clause gives the priority of sorting keys.

A set of records from the “final table” could be **grupped** in a single record (one record will replace a group of records). Such a group is determined by the **common values** of fields which appear in **GROUP BY** clause. "**The new table**" is ascending sorted (automatically by the system) based on the values of fields from **GROUP BY**. The consecutive records of the sorted table, with the same values for all fields from **GROUP BY**, are replaced with one record. The presence of this record in the *final table* is decided by an optional condition that appears in **HAVING** clause.

We can use the following functions on the group of records created as specified before:

- **AVG** $\left(\left[\left\{ \begin{matrix} \textbf{ALL} \\ \textbf{DISTINCT} \end{matrix} \right\} \right] \textbf{field} \right)$ or **AVG([ALL]) expression)**

For a group of records we take all values (specifying **ALL**, it is the default value) or just the distinct ones (when **DISTINCT** appears) of the field or numerical expressions and it is returned the **average value**.

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

This function returns the **number** of records in a group (with '*'), number of values of a field (with **ALL**, identical with '*'), or the number of distinct records from group (with **DISTINCT**).

- **SUM** $\left(\left[\left\{ \begin{matrix} \textbf{ALL} \\ \textbf{DISTINCT} \end{matrix} \right\} \right] \textbf{field} \right)$ or **SUM([ALL]) expression)**

The **sum** of all or distinct values of a field or of a numerical expression is computed for each group of records.

- $\left\{ \begin{matrix} \textbf{MAX} \\ \textbf{MIN} \end{matrix} \right\} \left(\left[\left\{ \begin{matrix} \textbf{ALL} \\ \textbf{DISTINCT} \end{matrix} \right\} \right] \textbf{field} \right)$ or $\left\{ \begin{matrix} \textbf{MAX} \\ \textbf{MIN} \end{matrix} \right\}$ **([ALL]) expression)**

For each record from group it returns the **maximum** or **minimum** value of a field or a numerical expression.

The five functions (**AVG, COUNT, SUM, MIN, MAX**) are called aggregation function and they can appear in expressions which describe resulting fields or as part of conditions in **HAVING** clause. Because these functions are applied to a group of records, in **SELECT** command the group should be generated by a **GROUP BY** clause. If GROUP BY is missing then entire "final table" is considered to be a group, so it will contain only one record.

In general, SELECT clause is not possible to refer fields which do not appear in GROUP BY clause or which are not arguments of an aggregation function. If such fields appear, **and if there are no errors thrown** by the database management system, then is chosen the value of the field of a random record from group.

Two tables having the same number of fields (columns) and with the same type for the fields being on the same positions could be **unified** in a single table, using **UNION** operator. The returned table will contain all possible records (with **ALL**) or just the distinct ones (without **ALL**). **ORDER BY** clause can appear just for the last SELECT statement.

Between two result sets of a SELECT statement we can use INTERSECT or EXCEPT (or MINUS) operators.

In a SELECT command the clauses should appear in the following order: expression_list FROM ... WHERE ... HAVING ... ORDER BY ...

A select command may be stored in database as a specific component called **view**, defined as:

CREATE VIEW view_name AS SELECT_command

2.4. Proposed problems

I.

a. Create a relational database, in 3NF, which is used by a software company to manage the following information:

- **activities**: activity code, description, activity type;
- **employees**: employee code, name, list of activities, the team he/she is a member of, the team leader;

where:

- an **activity** is identified by the "activity code";
- an **employee** is identified by the "employee code";

- an employee may be part of only one **team** and the team has only one leader, who is also an employee of the company;
- an employee may work on many activities and an activity may include many employees.

Justify the obtained data model is in 3NF.

b. For the data model obtained at point **a**, write queries for the following requests, using either relational algebra **or** TSQL:

b1. Return the name of all employees that have at least one activity of type "Design" and **do not** work at any activity of type "Testing".

b2. Return the name of all leaders who are in charge of teams of at least 10 employees.

II.

a. Create a relational database, in 3NF, which is used by a faculty to manage the following information:

- **disciplines**: code, name, number of credits, list of students who took an exam for that discipline;
- **students**: code, name, birth date, group, year of study, specialization, list of disciplines for which an exam was taken (including the date of the exam and the obtained grade);

Justify the obtained data model is in 3NF.

b. For the data model obtained at point **a**, using both relational algebra **and** TSQL, determine the list of students (name, group, number of promoted disciplines) who during the year 2013 promoted more than 5 disciplines. If a student has more than one successful exam for a discipline, then that discipline shall be counted only once.

III.

a. Create a relational database, in 3NF, which is used by a faculty to manage the following information about the graduate students enrolled for their final license exam: the student's serial number, code and name of their specialization, the title of their license project, code and name of their supervising teacher, code and name of their supervisor's department, list of the software resources needed for the presentation of their license project (e.g.: .NET C#, C++, etc), list of the hardware resources needed for the presentation of their license project (RAM 8GB, DVD Reader, etc.).

Justify the obtained data model is in 3NF.

b. For the data model built at point **a**, using both relational algebra and TSQL (at least one of each), write queries to obtain the following information:

b1. List of graduates (name, title of the license project, name of the supervising teacher) for who the supervising teacher is a member of a given department (the department is given by name).

b2. For a given department, return the number of graduates that have a supervising teacher from that department.

b3. List of all supervising teachers that guided students with their license project.

b4. List of all graduates that need the following two software resources: Oracle and C#.

3. Operating systems

3.1. The structure of UNIX file systems

3.1.1 Unix File System

3.1.1.1. UNIX Internal Disk Structure

3.1.1.1.1 Blocks and Partitions

A UNIX file system is stored either on a device such as a hard-drive or CD or on a hard-drive partition. The hard-drive partitioning is an operation relatively independent of the operating system stored there. Consequently, we will refer to partitions and physical devices simply as disks.

Block 0 – boot block
Block 1 – superblock
Block 2 – i-node
…
Block n – i-node
Block n+1 – file data zone
…
Block n+m – file data zone

UNIX Disk Architecture

A UNIX file is a sequence of bytes, each byte being individually addressable. A byte can be accessed both sequentially as well as directly. The data exchange between the memory and the disk is done in blocks. Older systems have a block size of 512 bytes. For a more efficient space management, newer systems use larger blocks, of sizes up to 4KB. A UNIX file system is a structured stored on disk, and structured as shown in the image above in four block types.
Block 0 contains the operating system boot loader. This is a program dependent on the machine architecture on which it is installed.
Block 1, the so called superblock, contains information that defines the file systems layout on the disk. Such information is:

- The number of i-nodes (to be explained in the next section)
- The number of disk zones
- Pointers to the i-node allocation map
- Pointers to the free space map
- Disk zone dimensions

Blocks 2 to n, where n is a defined when the disk is formatted are i-nodes. An i-node is theUNIX name for the file descriptor. The i-nodes are stored on disk as a list named i-list. The order number of an i-node in the i-list is represented on two bytes and is called i-number. This i-number is the link between the file and the user programs.
The largest part of the disk is reserved for file data. The space allocation for files is done using an elegant indexing mechanism. The starting point for this allocation is stored in the i-node.

3.1.1.2 Directories and I-Nodes

The structure of a file entry in a directory is shown in the image below:

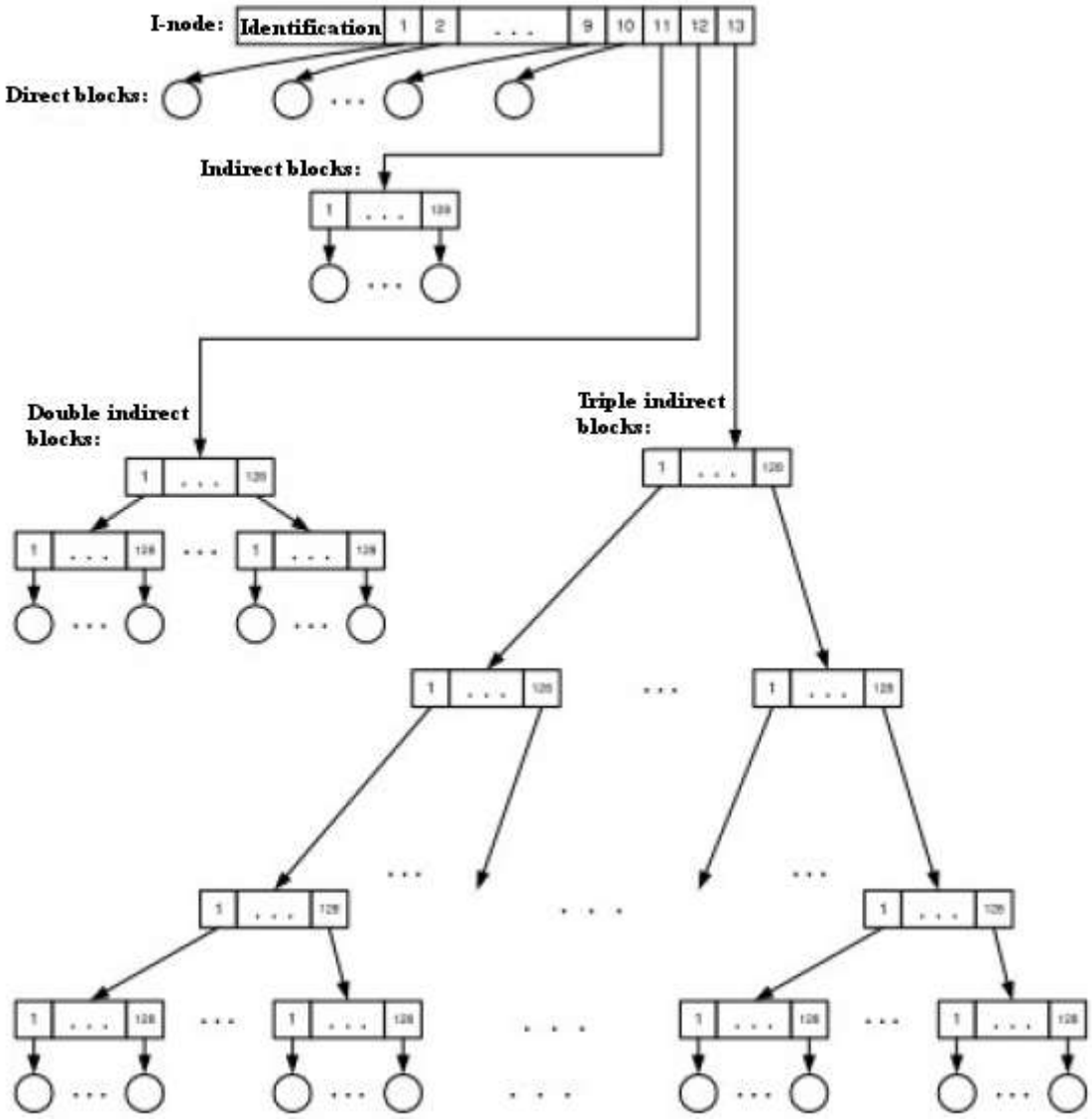
	File name (practically of unlimited length	i-number

The directory entry contains only the file name and its corresponding i-node reference. An i-node occupies usually between 64 and 128 bytes and contains the information in the table below:

Mode	File access permissions
Link count	Number of directories that contain references to this i-number, which is basically the number of links to this file
User ID	Owner user ID (UID)
Group ID	Owner group ID (GID)
Size	The number of bytes in the file (file length)
Access time	The time the file was last accessed
Modification time	The time the file was last modified
i-node time	The time the i-node was last modified
Block list	Address list of the first few file blocks
Indirect list	References to the rest of the blocks belonging to the file.

Every UNIX file system has a few individual constants, such as: block size, i-node size, disk address size, how many initial block addresses are stored directly in the i-node and how many references are stored in the indirect list. Regardless of these values, the storage and retrieval principles are the same.

To exemplify this we will use values that are frequently found in popular file systems. We will consider that a block is 512 bytes long, and that disk address is stored on 4 bytes. So, in a block we can store 128 such addresses. We will also consider that the i-node stored directly the addresses of the first 10 blocks of the file, and that the indirect list contains 3 elements. Using these values, the image below shows how the i-node points to the file blocks.



In the file i-node there is a list of 13 entries each referring the physical blocks of the file.

- The first 10 entries contain the addresses of the first 10 512-byte blocks of the file
- Entry 11 contains the address of a block named, indirect block. This block contains the addresses of the next 128 512-byte block of the file
- Entry 12 contains the address of a block named double indirect block. This block contains the addresses of 128 indirect blocks, each of which contains the addresses of 128 512-byte blocks of the file.
- Entry 13 contains the address of a block named triple indirect block. This block contains the addresses of 128 double indirect blocks, each of which in turn contains the addresses of 128 indirect blocks, each of which contains the addresses of 128 512-byte blocks of the file.

In the figure above we used circles to symbolize file data blocks and rectangles to symbolize reference blocks. As it can be easily seen form the figure, the maximum number of disk accesses for any part of the file is at most four. For small files, this number is even lower. As long as the file is open, its i-node is loaded and present in the internal memory. The table below shows the number of disk accesses necessary to obtain, using direct access, any file byte, depending on the file length.

Maximum Length (blocks)	Maximum Length (bytes)	Indirect Accesses	Data Accesses	Total Number of Accesses
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

More recent UNIX versions use blocks of 4096 bytes which can store 1024 reference addresses, and i-node stores 12 direct access addresses. Under these conditions, the table above changes as shown below:

Maximum Length (blocks)	Maximum Length (bytes)	Indirect Accesses	Data Accesses	Total Number of Accesses
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 (over 5000GB)	3	1	4

3.1.2. File Types and File Systems

In a file system, the UNIX system calls handle eight types of files:

1. Normal (the usual files)
2. Directories
3. Hard links
4. Symbolic links
5. Sockets
6. FIFO (named pipes)
7. Peripheral character devices
8. Peripheral block devices

Besides these eight type there are four more entities that the UNIX system calls treat syntactically exactly as if they were files:

9. Pipes (anonymous pipes)
10. Shared memory segments
11. Message queues
12. Semaphores

Normal files are sequences of bytes accessible either sequentially or directly using the byte's order number.

Directories. A directory file differs from a normal file only thorough the information it contains. A directory contains the list of names and addresses of the files it contains. Usually every user has a directory of its own which points to its normal files and other subdirectories.

Special files. In this category we can include, for now, the last six file types. Actually, UNIX regards any I/O device as a special file. From a user perspective there is no difference between the way it works with a normal disk file and the way it works with a special file.

Every directory includes the two special entries below:

"." (dot) it points to the directory itself

".." (two consecutive dots) points to the parent directory

Every file system contains a main directory named **root** or /.

Usually, every user is using a *current directory* attached to the user upon entering the system. The user can change this directory (**cd**), can create a new directory inside the current directory (**mkdir**), delete a directory (**rmdir**), display the access path from **root** to the current directory (**pwd**).
The appearance of large numbers of UNIX distributers lead, inevitably, to the appearance of many proprietary "extended file systems": For instance,

- Solaris is using the **UFS** file system
- Linux uses mostly **ext2** and more recently **ext3**
- IRIX is using **XFS**
- Etc.

Today's UNIX distributions support also file systems specific to other operating systems. The most important such file systems are listed below. The usage of these third party file systems is transparent to the user, however users are recommended to use such file systems carefully for operations other than reading. Modifying a Microsoft Word document from UNIX may result in the file being unusable.

- FAT and FAT32 specific to MS-DOS and Windows 9x
- NTFS specific to Windows NT and 2000

UNIX system administrators must pay attention to the types of file sytems used and the permissions they give to the users on them.
The *tree structure principle* of a file system states that any file or directory has a single parent. Implicitly, each file or directory has a single unique path of access starting from the root. The link between a file or directory and its parent directory will be call *natural link*. The natural links are created implicitly when the file or subdirectory is created.

3.1.2.1 Hard Links and Symbolic Links

There are situations when it is useful to share a part of the directory structure among multiple users. For instance, a database stored somewhere in the file system must be accessible to many users. UNIX allows such an operation using *additional links*. An additional link allows referring a file in different ways than the natural one. The additional links are: *hard links* and *symbolic links* (*soft*).
Hard links are identical to the natural links and can be created exclusively by the system administrator. A hard link is an entry in a directory that points to the same substructure to which its natural link is pointing to already. Consequently, a hard link makes the substructure appear to have two parent directories. Basically, a hard link gives a second name to the same file. This is why when parsing a directory tree files with hard links appear duplicated. Each duplicate appears with the number of hard links pointing to it.
For example, if there is a file named "old" and the system administrator runs the command:

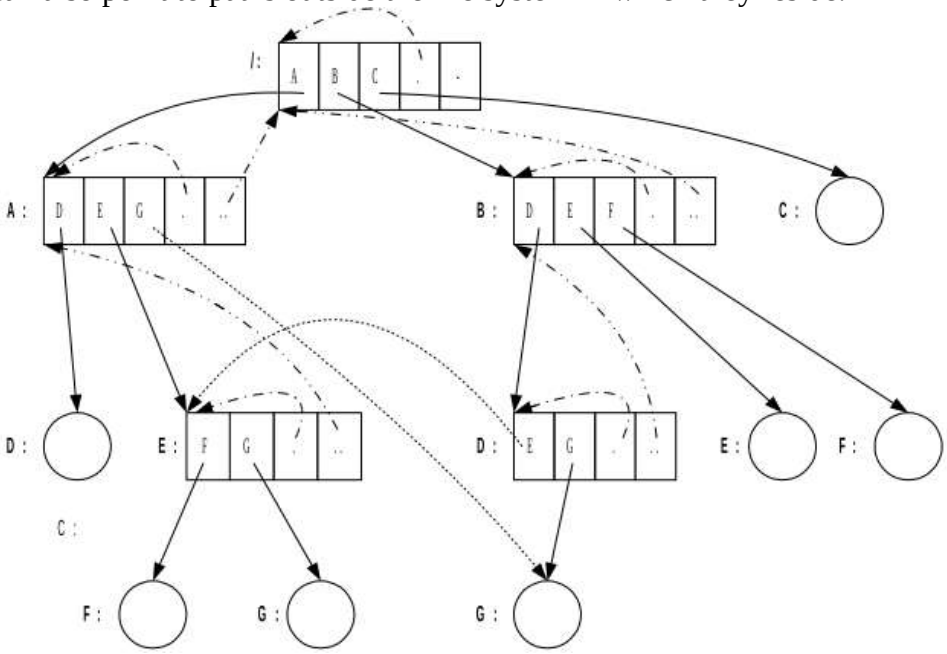
```
# ln old newlink
```

Then the file system will appear to have two identical files: `old` and `newlink`, each of them being marked as having two links pointing towards it. Hard links can only be created inside the same file system (we will give the details for this later on).
The symbolic links are special entries in a directory that point (reference) some file or directory in the directory structure. This entry behaves as a subdirectory of the directory where the entry was created.
In its simplest form, a symbolic link is created using the command below:

```
#ln –s pathInTheDirectoryStructure symbolicName
```

After the command above is executed, `pathInTheDirectoryStructure` will have an additional link marked, and `symbolicName` will point (exclusively) to this path. Symbolic

links can be used by regular users (not only by the system administrator like the hard links) and they can also point to paths outside the file system in which they reside.



The symbolic and hard links transform the tree structure of the file system in an acyclic graph structure. The example above shows a simple directory structure. Capital letters A, B ,C, D, E, F, and G are names of regular files, directories, and links. It is evidently possible for the same name to appear multiple times in the directory structure, thanks to the hierarchical structure of the directories which eliminate ambiguities. Regular files are marked with circles and directories with rectangles.
The links are marked with three types of arrows:

- Continuous line – natural links
- Dashed line – link toward the same directory or the parent directory
- Dotted line – Symbolic and hard links

There are 12 nodes in the example above, both directories and files. Regarded a s a tree, in other words considering the natural links exclusively, there are 7 branches and 4 levels.
Let us assume that the two links (dotted lines) are symbolic. To create these links one would have to run the commands below:

```
cd /A
ln –s /A/B/D/G G    First link
cd /A/B/D
ln –s /A/E E   Second link
```

Let's presume now that the current directory is B. We will parse the tree in order: directory followed by its subordinates from left to right. When applicable, we will put on the same line, multiple specifications of the same node. The entries that are links are underlined. The longest 7 branches are marked with # to their right.

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

When happens when deleting multiple links? For instance, what happens when one of the commands below get executed?

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

It is clear that the file must be unaffected if deleted only by one of the specifications. For this, the file descriptor holds a field named link counter. This has value 1 when the file is initially created and is incremented by 1 every time a new link is created. When deleted, the counter is decremented by 1, and if it is zero, only then the file gets deleted from the disc and the blocks it occupied are freed.

3.2. UNIX Processes

UNIX processes: creation, system calls: cork, exec, exit, wait; pipe and FIFO communication.

3.2.1. Main System Calls for Process Management

In this section we will present the most important system calls for working with processes: fork, exit, wait and exec. We will start with fork, which is the system call for creating a process.

3.2.1.1 Process creation. The fork system call

In the UNIX operating system a process is created using the fork() system call. When functioning normally, the effect of calling it is the following: the parent process memory image is copied in a free memory area, this copy being the newly created process, which in the first phase is identical to the initial process. The two processes continue their execution concurrently with the instruction following the fork() call.

The newly created process is called child process, and the process that called fork() is called the parent process. With the exception of the different address spaces, the child process differs from the parent process only through its process identifier (PID), its parent process identifier (PPID), and the value returned by the fork() call. When functioning normally, fork() return in the parent process (the process that called fork) the PID of the new child process, and in the child process it returns 0.

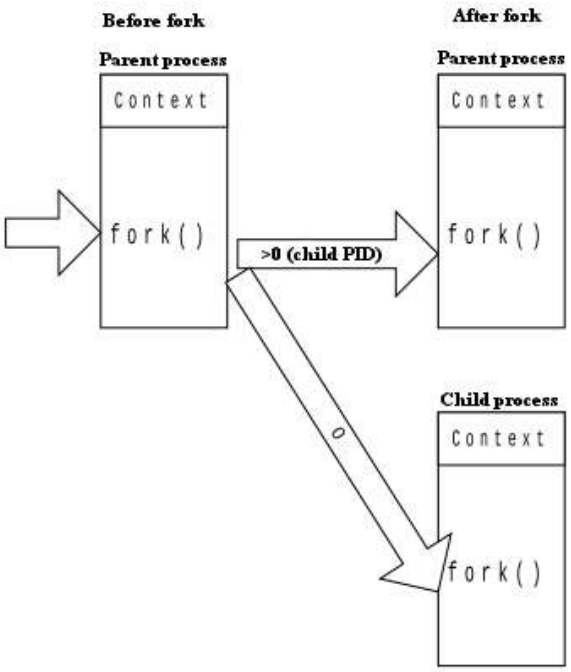


Figure 3.1 The fork mechanism

In the image above we show the fork() mechanism, the arrows indicating the instruction that is executed currently in the process.

If it fails, the fork() call returns value -1 si sets accordingly the errno environment variable. The fork() system call can appear if:

- the is not enough memory to create a copy of the parent process
- The total number of processes in the system is above a maximum limit

This behavior for the fork() system call makes it easy to implement two sequences of instructions that execute in parallel:

```
if(fork() == 0) {
    /* child instructions */
}
else {
    /* Parent instructions */
}
```

The program below shows how fork() can be used:

```
int main() {
    int pid, I;
    printf("\nProgram start:\n");
    if((pid=fork)<0) error_sys("Cannot execute fork()\n");
    else if(pid == 0) {  // This is the child process
        for(i=1;i<=10;i++) {
            sleep(2);  // sleep 2 seconds
            printf("\tCHILD(%d) of PARENT(%d): 3*%d=%d\n", getpid(),
                   getppid(), i, 3*i);
        }
        printf("End CHILD\n");
    }
    else if(pid > 0) { // This is the parent process
        printf("Created CHILD(%d)\n", pid);
        for(i=1;i<=10;i++) {
            sleep(2);  // sleep 2 second
```

```
            printf("PARENT(%d): 2*%d=%d\n", getpid(), i, 2*i);
        }
        printf("End PARENT\n");
    }
    return 0;
}
```

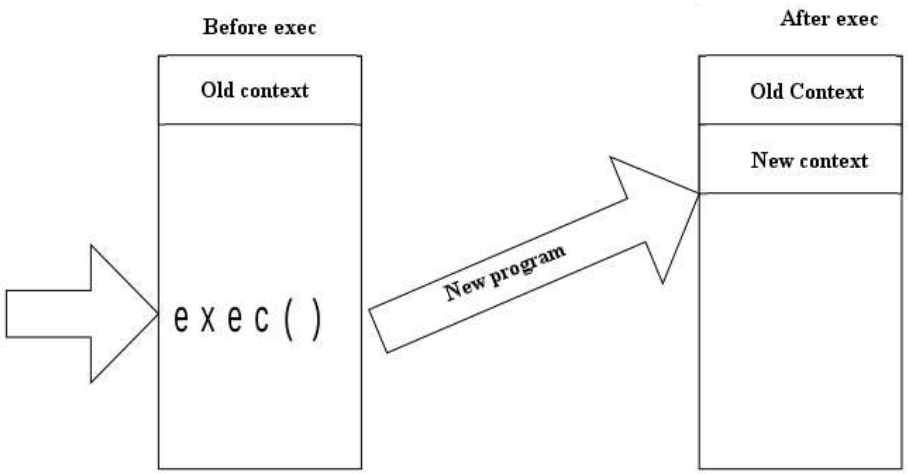
We have intentionally made it so that the child process wait more than the parent (when doing complex calculations it is often the case that the operations of one process last longer than those of the other). Consequently, the parent will finish the execution earlier. The results printed on the screen are:

```
Program start:
Created CHILD(20429)
PARENT(20428): 2*1=2
         CHILD(20429) of PARENT(20428): 3*1=3
PARENT(20428): 2*2=4
PARENT(20428): 2*3=6
         CHILD(20429) of PARENT(20428): 3*2=6
PARENT(20428): 2*4=8
PARENT(20428): 2*5=10
         CHILD(20429) of PARENT(20428): 3*3=9
PARENT(20428): 2*6=12
PARENT(20428): 2*7=14
         CHILD(20429) of PARENT(20428): 3*4=12
PARENT(20428): 2*8=16
PARENT(20428): 2*9=18
         CHILD(20429) of PARENT(20428): 3*5=15
PARENT(20428): 2*10=20
End PARENT
         CHILD(20429) of PARENT(1): 3*6=18
         CHILD(20429) of PARENT(1): 3*7=21
         CHILD(20429) of PARENT(1): 3*8=24
         CHILD(20429) of PARENT(1): 3*9=27
         CHILD(20429) of PARENT(1): 3*10=30
End CHILD
```

3.2.1.2 Executing an external program. The exec system calls

Almost all operating systems and programming environments offer, one way or another, mechanisms for executing a program from inside another program. UNIX offers this mechanism through the exec* system calls. As we will see, the combined usage of fork and exec offers great elasticity in handling processes.

The exec* system call family start a new program inside the same process. The exec system call gets the name of an executable file, and the content of this file overwrites the existing current process, as shown in the image below.



After calling exec, the instructions in the current program are not executed any longer, in their stead being executed the instructions of the new program.

UNIX offers six exec system calls categorized by three criteria:

- The path to the executable: absolute or relative
- Environment inheritance from or creation of a new environment for the new program
- Command line arguments specification: specific list or pointer array

From the eight possible combinations generated by the three criteria above were eliminated the two with relative path and new environment. The prototypes of the six exec function calls are:

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

The meaning of the exec parameters is:

- file – the name of the executable file that will replace the current process. This name must coincide with argv[0] or arg0.
- argv – is an array of pointers, ending in NULL, that contains the command line arguments for the new program about to be executed.
- arg0, arg1, …, argn, NULL are the command line arguments of the program about to be executed, given explicitely as strings. The last of them must be NULL as well.
- envp – is an array of pointer, also ending in NULL, that contains the string corresponding to the new environment variables, given in the format "name=value"

3.2.1.3 System calls exit and wait

The system call

```
exit(int n)
```

causes the current process to finish and the return to the parent process (the one that created it using fork). Integer n is the exit code of the process. If the parent process does not exist any longer, the process is moved to a zombie state and associated with the `init` process (PID 1).

The system calls

```
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int* status, int options);
```

are used to wait for the termination of a process. The call to wait() suspends the execution of the current process until a child process ends. If the chil ended before the wait() call, the call ends immediately. Upon the completion of the wait() call, the child's resources are all freed.

3.2.2. Communicating between processes using pipe

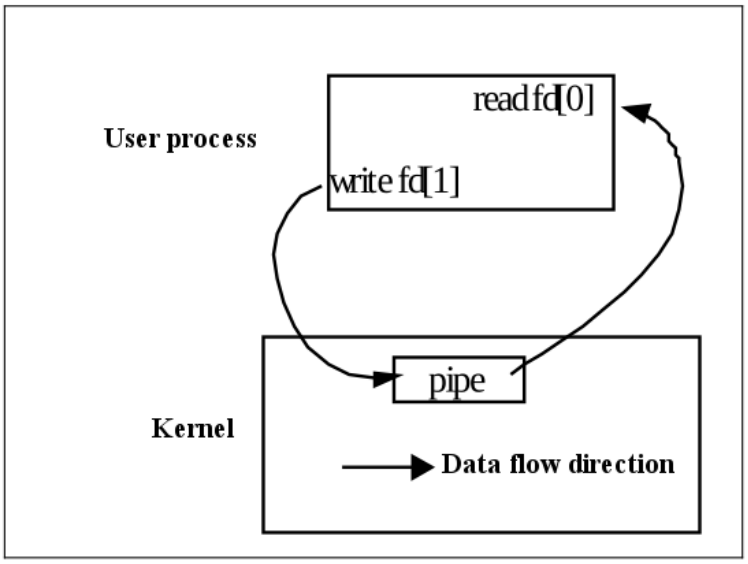
3.2.2.1 Pipes

The PIPE concept appeared for the first time in the UNIX system in order to allow a child process communicate with its parent process. Usually, the parent process redirects its standard output (stdout) towards the pipe, and the child process redirects its standard input (stdin) to come from the pipe. Most operating systems use the operator "|" to mark this type of connection between the operating system commands.

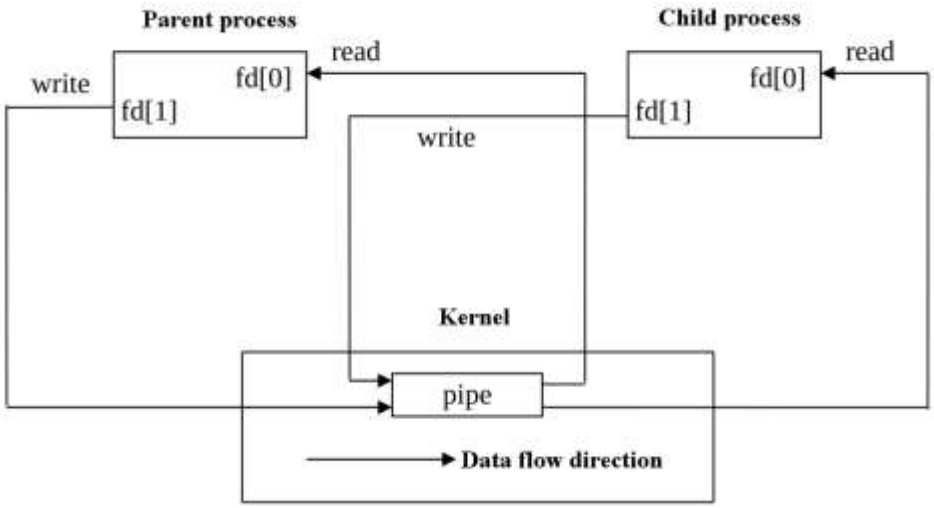
A UNIX pipe is a unidirectional flux of data, managed by the operating system kernel. Basically, the kernel allocates a buffer of 4096 bytes that are managed as presented above. The creation of a pipe is done using the system call below:

Int pipe(int fd[2]);

The integer fd[0] is a file descriptor that can be used to read from the pipe, and fd[1] is a file descriptor that allows writing to the pipe. After its creation, the link between the user and the kernel created by this pipe appears as in the image below.



Obviously, a pip within a single process does not make much sense. It is essential that pipe and fork work together. Consequently, after the pipe is created, fork should be called, and then the link between the two processes with the kernel looks as follows:



The unidirectional communication through pipe left entirely to be insured by the developer. To insure this, the developer should close the unnecessary pipe ends before starting the communication.

- The parent process should call close(fd[0])
- The child should call closed(fd[1])

If the desired direction of communication is reversed, then the operations above should be reversed too: the parent will execute close(fd[1]0 and the child will execute close(fd[0]).

3.2.2.2 Example: implement who|sort through pipe and exec

Let's consider the following shell command:

$who|sort

We will present how the link between the two commands is created through pipe. The parent process (which replaces the shell process) creates two child processes, and these two redirect their input and outputs accordingly. The first child process executes the command who, and the second child process executes the command sort, while the parent waits for their completion. The source code is presented in the program below.

//whoSort.c
//Execute who | sort
#include <unistd.h>

```
int main (){

      int p[2];
      pipe (p);
      if (fork () == 0) { // First child
         dup2 (p[1], 1); //redirect standard output
         close (p[0]);
         execlp ("who", "who", 0);
      }
      else if (fork () == 0) { // second child
         dup2 (p[0], 0); // redirect standard input
         close (p[1]);
         execlp ("sort", "sort", 0);//execute sort
      }
      else { // Parinte
         close (p[0]);
         close (p[1]);
```

```
        wait (0);
        wait (0);
      }
    return 0;
}
```

Note: To better understand the code above, the reader should learn about the system call dup2 from the UNIX manual pages. Here dup2 takes as a parameter the pipe descriptor.

3.2.3. Communicating between processes with FIFO

3.2.3.1 The Concept of FIFO

The main disadvantage of using pipe in UNIX is that it can only be used be related processes: the processes communicating through pipe must be descendants of the process creating the pipe. This is necessary so that the pipe descriptors are inherited by the child process created with fork.

UNIX System V (around 1985) introduced the concept of FIFO (named pipes). This is a unidirectional flux of data accessed through a file stored on disk in the file system. The difference between pipe and FIFO is that FIFO has a name and is stored in the file system. Because of this, a FFO can be accessed by any processes, not necessarily having a common parent. However, even if the FIFO is stored on disk, no data is stored on disk at all, the data being handled in system kernel buffers.

Conceptually, pipe and FIFO are similar. The essential differences are the following:

- The pipe is managed in the RAM memory part managed by the kernel, while for FIFO this is done on the disk
- All the processes communicating through pipe must be descendants of the creating process, while for FIFO this is not necessary

The creation of a FIFO is done using one of the system calls below:

```
int mknod(char* name, int mode, 0);
int mkfifo(char* name, int mode);
```

or using the shell commands below:

```
$ mknod name p
$mkfifo name
```

- The “name” parameter is the name of the file of type FIFO
- The “mode” argument gives the access permissions to the FIFO file. When using mknod, mode must specify the flag S_IFIFO, besides the access permission (connected through the bitwise-OR operator |). This flag is defined in <sys/stat.h>
- The last parameter of the mknod system call is ignored, and that is why we put there a 0.
- When using the command mknod, the last parameter must be “p” to tell the command to create a named pipe

The two function calls above, although specified by POSIX, are not both system calls in all UNIX implementations. FreeBSD implements them both as system calls, but in Linux and Solaris only mknod is a system call while mkfifo is a library function implemented using mkfifo. The two shell commands are available on most UNIX implementations. In older

UNIX distributions the commands are only accessible to the superuser, but starting with UNIX System 4.3 they are also available to regular users.

To delete a FIFO, one can use either the “rm” shell command or the C function call “unlink”.

One a FIFO is created, it must be open for read and write, using the system call “open”. The functioning of this, its effects and the effects of the O_NDELAY flag given to the “open” system call are presente din the table below.

Operation	Without the O_NDELAY flag	With the O_NDELAY flag
Open FIFO read-only, but there is no process opening it for writing	Wait until there is a process opening it for writing	Return immediately without signaling any error
Open FIFO write-only but there is no process opening it for reading	Wait until there is a process opening it for reading	Return immediately and set errno to ENXIO
Read from FIFO or pipe when there is no data available	Wait until there is data in the pipe or FIFO, or until there is no process opening it for writing. Return length of the read data, or 0 if there is no process left writing.	Return immediately with value 0
Write to FIFO or pipe when it is full	Wait until there is space available for writing, and write as much as possible.	Return immediately with value 0

3.2.3.2 Example: Client/Server communication through FIFO

The client/server application model is classic. In the following program we will present a sketch of client/server application communicating through FIFO. To insure bi-directional communication we will use two FIFOs. The specific logic of the application is not given and is replaced by calls to functions client(int in, int out) and server(int in, int out). Each of them gets the file descriptors as parameters, and they sue these to communicate with their partner.

The two programs rely on the FIFOs to be created ahead of time with Shell commands and deleted with shell commands afterwards.

Server program:

```
#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"

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

```
- - - - - - - - - - - - - - -
   readfd = open (FIFO1, 0));
   writefd = open (FIFO2, 1));
   for ( ; ; ) {  // infinite loop for waiting requests
        server(readfd, writefd);
   }
 - - - - - - - - - - - - - - -
   close (readfd);
   close (writefd);
   return 0;
}
```

Client program:

```
#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"

int main() {
   int    readfd, writefd;
 - - - - - - - - - - - - - - -
   writefd = open (FIFO1, 1));
   if ((readfd = open (FIFO2, 0));
   client(readfd, writefd);
 - - - - - - - - - - - - - - -
   close (readfd);
   close (writefd);
   return 0;
}
```

3.3. Command File Interpreters

3.3.1. Shell Command Interpreter Functioning

A command interpreter (shell) is a special program that provides an interface between the user and the UNIX operating system (kernel). From this perspective, the shell interpreter can be regarded in two ways:

- *Command language* that acts as an interface between the user and the system. When a user opens a working session, a shell is started implicitly and acts as a command interpreter. The shell displays to the standard out (usually a terminal) a prompt, offering the user means for running commands.
- *Programming language* that has as base elements the UNIX commands (semantically similar to the assigning instruction in other programming languages). The primitive element for conditions is the exit code of the last command. Zero means TRUE and any other value means FALSE. Shells have the notions of variable, constant, expression, control structure, and subprogram. The expressions used by Shell are mostly character strings. The syntactical aspects were reduced to a minimum.

Upon start, a shell stays active until specifically closed, and follows the algorithm below:

```
While(session is not closed)
    Display prompt;
    Read command line;
    If (command line ends with '&') then
        Create a process that executes the command
        Do not wait for its execution to complete
    Else
        Create a process that executes the command
        Wait for its execution to complete
    End If
End While
```

The algorithm above emphasizes two modes in which a command can be executed:

- Mode foreground – visible execution. The shell waits for the execution to finish ad then displays a prompt again. This is the implicit execution mode for any UNXI command.
- Mode background – hidden execution. The shell executed the command but does of wait for it to finish, and displays the prompt immediately after it started, offering the user the possibility to run another command. In order to run a command in the background, one must end it with '&'

In a UNIX session, one can run any number of background commands but only one foreground command. For example, the script below runs two command in the background, one for copying a file (cp) and one for compiling a C program (gcc), and one command in foreground for editing a text file (vi).

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

3.3.2. Shell Programming

3.3.2.1 The SH programming language

In this section we will present the SH language, the simplest of the UNXI shells. We will discuss the main syntactical categories. The semantics and functionalities of each such category are easy to infer from the context.

We will rely on the following conventions for describing the grammar of the SH language. Please note that these conventions are used exclusively for specifying the grammar. Similar constructs will appear later on file specifications and regular expressions, but they will have different meaning.

- A grammatical category can be defined through one or more alternative rules. The alternatives are written one on each line, starting on the line next to the category's name, as follows

  GrammaticalCategory:
  Alternative 1
  …
  Alternative N
- [ ]? Means that the expression between parentheses can appear at most once
- [ ]+ Means that the expression between parentheses can appear at least once

- [ ]* Means that the expression between parentheses can appear zero or more times

The image below presents the SH language syntax is described at a higher level (without entering into too many details), using the conventions above. The meaning of some of the syntactical elements in the image below is:

- word: sequence of characters not including blanks (space or tab)
- name: sequence of characters that starts with a letter and continues with letters, digits or _ (underscore)
- digit: the ten decimal digits (0, …, 9)

```
command:
    basicCommand
    ( cmdList )
    { cmdList }
    if cmdList then cmdList [ elif cmdList then cmdList ]* [ else cmdList
]? fi
    case word in [ word [ | word ]* ) cmdList ;; ]+ esac
    for name do cmdList done
    for name in [ word ]+ do cmdList done
    while cmdList do cmdList done
    until cmdList do cmdList done

basicCommand:
    [ element ]+

cmdList:
    pipedChain [ separator pipedChain ]* [ end ]?

pipedChain:
    command [ | command ]*

element:
    word
    name=word
    >word
    <word
    >>word
    <<word
    >&digit
    <&digit
    <&-
    >&-

separator:
    &&
    ||
    end

end:
    ;
    &
```

An SH *command* can have any of the nine forms presented above. One of the ways to define it is *basicCommand*, where a *basicCommand* is a sequence of *element*s, and *element* being defined in any of ten possible ways. A *pipedChain* is either one *command* or a sequence of *command*s connected through the special character '|'. Finally, *cmdList* is a sequence of *pipedChains* separated and possibly terminated with special characters.

The grammar described above makes it evident that SH accepts constructs without any semantics. For instance, *command* can be a *basicCommand*, which can contain a single *element* consisting of *>&-;*. Such input is accepted by SH from a syntactical point of view, although it does not have any semantic meaning.

The SH shell has the following thirteen reserved words:

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

The alternative **if** and **case** structures are closed by **fi** respectively **esac**, which are obtained by mirroring the opening reserved word. In the case of the repetitive structures, the ending is marked by the word **done**.

We are closing this section by presenting the syntax of a few reserved constructs, as well as a few characters with special meaning in the SH shell.

a) Syntactical constructs

|	connect through pipe
&&	logical AND connection
||	logical OR connection
;	separator/command ending
;;	case delimiter
(), {}	command grouping
< <<	input redirection
> >>	output redirection
&digit &-	standard input or output specification

b) Patterns and generic names:

*	any sequence of characters
?	any character
[…]	match any character in …

Note: these patterns must not be confused with the grammar conventions presented in the beginning of this section.

3.4. Proposed Problems

I.

a. Describe briefly the functioning of the fork system call and the values it can return.

b. What will print to the screen the program fragment below, considering that the fork system call is successful? Justify your answer.

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

```
}
```

c. What will print to the screen the shell script fragment below? Explain the functioning of the first three lines of the fragment.

```
1  for F in *.txt; do
2      K=`grep abc $F`
3      if [ "$K" != "" ]; then
4          echo $F
5      fi
6  done
```

II.

a. Consider the code fragment below. What lines will it print to the screen and in what order, considering that the fork system call is successful? Justify your answer.

```
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. Explain the functioning of the shell script fragment below. What happens if the file *report.txt* is missing initially? Add the line of code missing for generating the file *report.txt*

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

4. General bibliography

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

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