§5. Moştenire simplă.

Obiectele din domeniul de problemă nu sunt statice unul față de altul, ci inter-acționează între ele. Vom spune că obiectele din cadrul unei probleme se află în anumite relații.

Vom evidenția o serie de relații dintre obiecte:

• relații de asociere, când un obiect transmite un mesaj altui obiect;

• relații de agregare, când un obiect este parte componentă a altui obiect;

• relații de tip client-server, când un obiect foloseşte serviciile oferite de alt obiect;

• relații de moştenire, când un obiect transmite o serie de proprietăți şi operații altui obiect.

Fiindcă obiectele de acelaşi tip formează o clasă de obiecte, se poate vorbi şi despre relații între clase, generalizându-le pe cele enumerate anterior la nivel de clase.

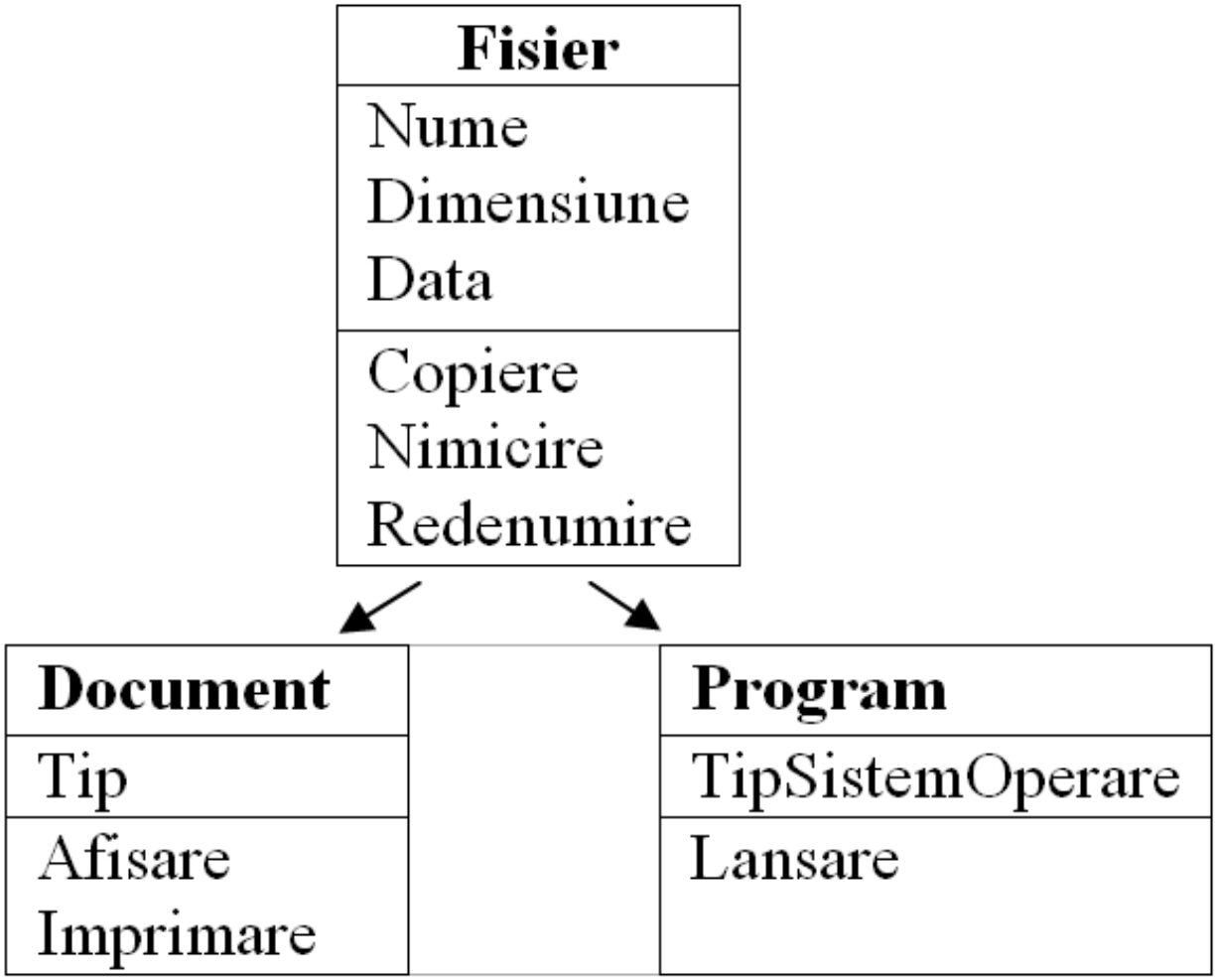
Utilizând relația de moştenire în procesul programării se obțin aplicații eficiente din punctul de vedere al mărimii codului obținut şi al lipsei de erori.

De exemplu, trebuie de realizat un sistem de gestiune în cadrul sistemului de fişiere. În domeniul de problemă pot fi evidenţiate o serie de obiecte caracteristice. Ca exemplu ar putea fi evidenţiate obiectele **Program** şi **Document**. Dacă ar fi proiectate tipuri abstracte de date corespunzătoare acestor obiecte, evidenţiind proprietăţile şi operaţiile caracteristice acestor obiecte, s-ar putea propune următoarele clase:

Document	**Program**
Nume Dimensiune Data Tip	Nume Dimensiune Data TipSistemOperare
Copiere Nimicire Redenumire Afişare Imprimare	Copiere Nimicire Redenumire Lansare

Analizând clasele propuse, se poate observa că există o serie de proprietăți şi o serie de operații comune pentru ambele tipuri de obiecte. Definind funcțiile membre ale ambelor clase, vom avea o serie de funcții din diferite clase realizate identic. Astfel, se obține o dublare de cod.

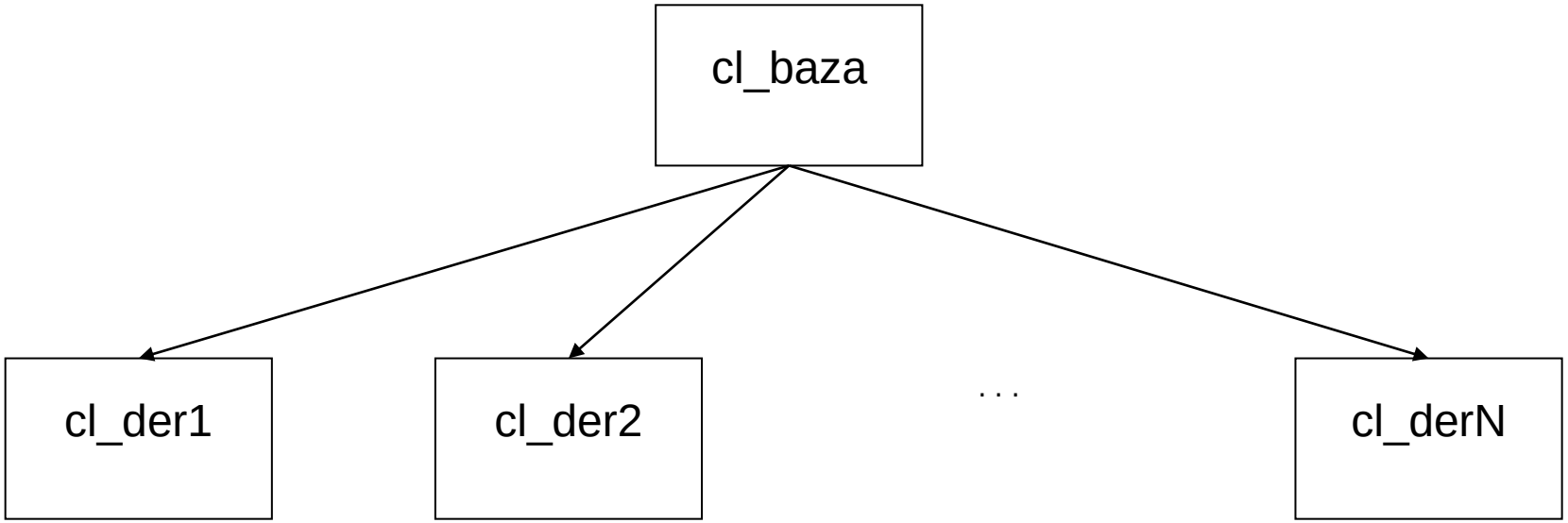
O cale mai eficientă ar fi posibilitatea de evidențiere şi implementare a unei clase generale, care ar putea transmite caracteristicile sale altor clase mai concretizate. Astfel în cazul exemplului de mai sus s-ar putea evidenția obiectul general numit **Fisier**, care va conține caracteristicile comune ale obiectelor ce țin de sistemul de fişiere:


Fisier
Nume
Dimensiune
Data
Copiere
Nimicire
Redenumire
Document
Tip
Afisare
Imprimare
Program
TipSistemOperare
Lansare

Transmiterea de proprietăți şi funcționalități dintr-o clasă numită **clasă de bază** într-o clasă numită **clasă derivată** se numeşte moştenire simplă.

Procesul de transmitere se numeşte derivare.

La nivel de diagramă, moştenirea simplă are următoarea reprezentare:



La nivel de descriere în limbajul C++, avem următoarea formă generală:

```
class cl_baza
{
.
<proprietati si metode cl_baza>
.
};

class cl_der1:[modificator_de_mostenire1] cl_baza
{
.
<proprietati si metode cl_der1>
.
};
...

class cl_derN:[modificator_de_mostenireN] cl_baza
{
.
<proprietati si metode cl_derN>
.
};
```

unde elementele `modificator_de_mostenire1, ..., modificator_de_mostenireN` reprezintă modificatorii de moştenire prin care clasa de bază este implicată în procesul de derivare. Modificatorii de moştenire pot fi unul dintre cuvintele-cheie:

`private`

`protected`

`public`

Dacă modificatorul de moştenire nu este prezent în descriere, se consideră implicit modificatorul `private`.

Urmând schema generală, pentru exemplul anterior, s-ar obține următoarea descriere:

```
class Fisier
{       char Nume[256];
        unsigned long Dimensiune;
        char Data[20];
public:
        void Nimicire();
        void Copiere(char *numeFisier);
        void Redenumire(char *numeNou);
};
//---------------
class Document : public Fisier
{       char Tip[15];
public:
        void Afisare();
        void Imprimare();
};
//---------------
class Program : public Fisier
{       char TipSistemOperare[25];
public:
        void Lansare();
};
```

Un obiect creat în baza clasei derivate are o serie de caracteristici dependente de clasa de bază. De aceea, în procesul de creare a acestui obiect, va participa atât constructorul clasei de bază, cât şi constructorul clasei derivate.

Acest fapt este vizibil şi la nivel de descriere generală a constructorului clasei derivate:

```
cl_der::cl_der(param_cl_der):cl_
baza(param_cl_baza)
{
.
//instructiuni
.
}
```

De obicei, lista parametrilor clasei de bază este o submulțime a listei parametrilor clasei derivate.

Fiindcă procesul de creare a unui obiect în baza clasei derivate depinde de doi constructori, este bine a şti ordinea execuției lor: mai întâi va lucra constructorul clasei de baza, iar apoi va lucra constructorul clasei derivate.

Ordinea de execuție a destructorilor este inversă: mai întâi lucrează destructorul din clasa derivată, iar apoi lucrează destructorul din clasa de bază.

A fost menționat că o clasă derivată are o serie de caracteristici care vin din clasa de bază (le moşteneşte). Vom examina tipul de acces la aceşti membri moşteniți în clasa derivată. Combinația dintre modificatorii de acces ai membrilor clasei de bază şi modificatorul de moştenire determină tipul de acces la membrii clasei derivate.

Tabelul de mai jos descrie combinațiile posibile și tipul de acces obținut ca rezultat:

		Modificator de moştenire		
		private	protected	public
Modificator de acces (în clasa de bază)	private	inaccesibil	inaccesibil	inaccesibil
	protected	private	protected	protected
	public	private	protected	public

Exemplu. De alcătuit un program în care sunt implementate clasa `punct` şi clasa `cerc`, care este derivată din clasa `punct`.

```
class punct
{
protected:
     int x, y;
     int visibil;
public:
     punct(int x1, int y1);
     void afisare();
     void stingere();
     void miscare(int xn, int yn);
};
```

```
class cerc : public punct
{
   int r;
public:
   cerc(int x, int y, int r);
   void afisare_c();
   void stingere_c();
   void miscare_c(int xn, int yn);
};
```

```
punct :: punct(int x1, int y1)
      {
            x=x1;
            y=y1;
      }
//-------------------------
void punct :: afisare()
      {
            if (visibil == 0)
            {
                  putpixel(x, y, getcolor() );
                  visibil=1;
            }
      }
```

```
void punct :: stingere()
     {
          if (visibil==1)
          {
             putpixel(x, y, getbkcolor() );
             visibil=0;
          }
     }
//-------------------------
void punct :: miscare(int xn, int yn)
{
     stingere();
     x=xn;
     y=yn ;
     afisare();
}
```

```
cerc :: cerc(int x, int y, int r1):
punct (x,y)
{ r=r1; }
//-------------------------
void cerc :: afisare_c()
{
    if (visibil ==0)
    {
        circle(x,y,r);
        visibil=1;
    }
}
```

```
void cerc :: stingere_c()
{
     int c=getcolor();
     if (visibil == 1)
     {
          setcolor(getbkcolor());
          circle(x,y,r);
          setcolor(c);
          visibil=0;
     }
}
```

```
void cerc :: miscare_c(int xn,
int yn)
{
    stingere_c();
    x=xn;
    y=yn;
    afisare_c();
}
```

Declararea claselor şi implementarea lor vor fi memorate în fişierul `cerc.hpp`. Programul principal în care sunt utilizate obiecte create în baza clasei `cerc` va fi scris în fişierul `cerc.cpp`:

```
//cerc.cpp
#include<iostream.h>
#include<conio.h>
#include<graphics.h>
#include ”cerc.hpp”
void main()
{
  int dr = DETECT, rdr, er;
  initgraph(&dr, &rdr, ””);
  er=graphresult();
  if (er==grOk)
   {
        cerc c1(100, 100, 20);
        punct p(150, 270);
        p.afisare();
        c1.afisare_c();
        c1.miscare_c(200,300);
        getch();
   }
   else
       cout << ”Eroare de initializare a regimului grafic.\n”;
   closegraph();
}
```

(~§5)