

Capitolul 1

Paradigme de programare și metode de proiectare a programelor

Noțiunea de *paradigmă* se bazează pe un cuvânt ce provine din limba latină și greacă și care reprezintă un exemplu sau un model. Sensul uzual al noțiunii este dat de istoricul Thomas Kuhn în cartea sa “The Structure of Scientific Revolutions”: o *paradigmă* este o mulțime de teorii, standarde și metode ce reprezintă o modalitate de organizare a cunoștințelor.

Bazat pe această noțiune, Robert Floyd în articolul intitulat “The Paradigms of Programming”, definește noțiunea de *paradigmă de programare* ca fiind o metodă de conceptualizare a modului de execuție al calculelor într-un calculator, precum și a modului de structurare și organizare al taskurilor reponsabile cu execuția calculelor. O noțiune des utilizată în locul celei de paradigmă de programare este cea de *stil de programare*, deși semnificația sa nu este foarte clar definită.

Despre un limbaj de programare se spune că *oferă suport* pentru o paradigmă de programare, dacă acesta pune la dispoziție facilități care îl fac convenabil de a fi utilizat în acest stil. Un limbaj de programare *permite* doar acest lucru dacă efortul cerut pentru a scrie un program în stilul respectiv este mai mare, limbajul neoferind facilități suficiente.

1.1 Programarea procedurală

Aceasta este una dintre cele mai vechi și des utilizate paradigme. Ea presupune în mod uzual parcurgerea următoarelor etape :

- a) descompunerea problemei de rezolvat în subprobleme ;
- b) găsirea pentru fiecare subproblemă a unui algoritm optim de rezolvare ;
- c) implementarea fiecărui algoritm folosind funcții sau proceduri ale unui anumit limbaj de programare.

Cel mai vechi limbaj de programare procedural este FORTRAN, însă majoritatea limbajelor de programare actuale oferă suport pentru această paradigmă. Principalele probleme legate de programarea procedurală se referă la tipurile de funcții folosite (funcții, proceduri, subprograme, rutine, macrouri, etc.), la tipul și modul de transmitere ale parametrilor și la modurile de apel.

Exemplul 1.1. Definirea și utilizarea unei funcții care determină dacă un număr întreg este prim :

```
int Prim(int n) {  
    // codul pentru functie  
}
```

```

void DivizoriPrimi(int n) {
    int i;
    for (i=2; i<n/2; i++)
        if (n%i == 2 && Prim(i))
            printf("%d", i);
}

```

1.2 Încapsularea datelor (modularizarea)

De-a lungul timpului, accentul în programarea procedurală s-a deplasat de la proiectarea funcțiilor la organizarea datelor. Datele nu mai sunt privite în mod separat, ci împreună cu funcțiile care le prelucrează. A fost definită astfel noțiunea de *modul* ca reprezentând un set de funcții înrudite, împreună cu datele pe care le prelucrează.

Se poate astfel împărți un program în module componente, într-un mod mai clar și mai eficient decât împărțirea clasică în funcții sau proceduri, aici datele programului fiind încapsulate (ascuse) în modulele ce le utilizează.

În mod uzual un modul conține o parte de *interfață* în care sunt declarate datele și funcțiile accesibile din afara modulului, precum și o parte de *implementare*, proprie modulului și inaccesibilă în exterior în cadrul căreia sunt definite funcțiile ce manipulează datele din modul.

Limbajul C permite programarea modulară, pe când limbajul Pascal sau Modula-2 oferă suport real pentru un astfel de stil de programare. În limbajul Turbo Pascal noțiunea de modul corespunde celei de **unit**, părțile de interfață și implementare corespunzând respectiv secțiunilor **interface** și **implementation**.

În cazul limbajului C, partea de interfață se specifică în mod uzual într-un fișier header, care trebuie inclus în toate celelalte fișiere ale unui program care utilizează funcțiile modulului. Partea de implementare a modulului este realizată într-un fișier distinct care trebuie inclus în proiectul programului.

Exemplul 1.2. Definirea și utilizarea unui modul pentru operațiile cu numere întregi :

```

// fisierul sir.h - interfata modulului
#define max_dim 100
void Initializare();
int Suma();
void Sortare();
void AdaugaElement(int);
void Listare();

// fisierul sir.c - implementarea modulului
#include "sir.h"
static int dim;
static int v[max_dim];
void Initializare() { dim = 0; }
void AdaugaElement(int k) { v[dim++] = k; }

```

```

int Suma() {
    // codul pentru calculul sumei elementelor sirului
}
void Sortare() {
    // codul pentru sortarea elementelor sirului
}
int Listare() {
    // codul pentru listarea elementelor sirului
}

// fisierul pr.c - utilizarea modulului sir
#include "sir.h"
void Prelucrare() {
    int i, s, k, n = 0;
    Initializare();
    for(i=0; i<n; i++) {
        scanf("%d", &k);
        AdaugaElement(k);
    }
    s = Suma();
    printf("\nSuma=%d", s);
    Sortare();
    Listare();
}

```

1.3 Abstractizarea datelor

În exemplul precedent se utilizează doar un singur șir de numere. Dacă se dorește să se lucreze cu mai multe șiruri, încapsularea datelor nu este suficientă.

Noțiunea de **abstractizare a datelor** presupune posibilitatea definirii unor tipuri de date utilizator, împreună cu un set de operații aferente fiecărui tip. Limbajele ce oferă suport pentru încapsularea datelor permit și abstractizarea, dar nu o garantează în general. De exemplu, în C, fișierele antet permit declararea împreună atât a tipurilor de date cât și a funcțiilor.

Un **tip de date abstarct** (ADT – Abstract Data Type) este definit printr-o mulțime de operații ce se pot efectua asupra elementelor sale, care formează **interfața** tipului de date și este singurul mod de acces la tipul de date, precum și printr-o mulțime de **axiome** și **precondiții**, care este privată tipului de date și reprezintă modul de descriere ale proprietăților și operațiilor tipului. Implementarea unui tip de date abstract într-un limbaj de programare reprezintă un **tip de date**.

Exemplul 1.3. Definirea în limbajul C a unui tip de date reprezentând numerele fracționare:

```

// fisierul fractie.h - interfata tipului fractie
typedef struct {
    int p, q;           // numaratorul si numitorul
} fractie;

```

```

// declararea principalelor operatii
fracție Suma(fracție, fracție);
fracție Diferența(fracție, fracție);
fracție Produs(fracție, fracție);
fracție Raport(fracție, fracție);

// fisierul fracție.c - implementarea tipului fracție
#include "fracție.h"

fracție Suma(fracție f1, fracție f2) {
    fracție f;
    f.p = f1.p * f2.q + f2.p * f1.q;
    f.q = f1.q * f2.q;
    return f;
}

fracție Produs(fracție f1, fracție f2) {
    // codul pentru produs
}

fracție Diferența(fracție f1, fracție f2) {
    // codul pentru diferența
}

fracție Raport(fracție f1, fracție f2) {
    // codul pentru raport
}

// fisierul pr.c - utilizarea tipului fracție
#include "fracție.h"

void Prelucrare() {
    fracție f1 = {1, 2}, f2 = {7, 4}, f3;
    // f3 = f1*f2 + f1/f2
    f3 = Suma(Produs(f1, f2), Raport(f1, f2));
    printf("\nf3 = %d/%d", f3.p, f3.q);
}

```

Limbaje precum Ada și C++ permit utilizatorilor să definească tipuri de date care se comportă la fel ca și tipurile predefinite, datorită posibilității utilizării claselor și a supraîncărcării operatorilor.

Implementarea unui tip de date utilizator într-un limbaj de programare se poate realiza cel mai eficient prin intermediul noțiunii de **clasă**. Clasele reprezintă cea mai utilizată metodă de reprezentare a tipurilor de date abstracte. Ele permit atât specificarea interfeței tipului de date, cât și definirea operațiilor permise de acesta.

O **clasă** poate fi privită ca o extensie a noțiunii de structură din limbajul C, care permite definirea în interiorul clasei atât a datelor, cât și a funcțiilor care utilizează aceste date. **Supraîncărcarea** unui operator reprezintă o funcție a carei semnificație este definită de programator, dar al carei apel trebuie să corespundă sintaxei limbajului.

Exemplul 1.4. Redefinirea în limbajul C++ a tipului utilizator *fractie*:

```
// fisierul frac.h - definirea clasei fractie
class fractie {
    int p, q;
// interfata clasei
public:
    // constructorul clasei
    fractie(int a = 0, int b = 1) { p = a; q = b; }
    // declararea operatorilor supraancarcati +, -, * si /
    friend fractie operator + (fractie, fractie);
    friend fractie operator - (fractie, fractie);
    friend fractie operator * (fractie, fractie);
    friend fractie operator / (fractie, fractie);
    void Print();
};

// fisierul frac.c - implementarea clasei fractie
#include "frac.h"
#include <iostream.h>

fractie operator+(fractie f1, fractie f2) {
    fractie f;
    f.p = f1.p * f2.q + f2.p * f1.q;
    f.q = f1.q * f2.q;
    return f;
}

fractie operator*(fractie f1, fractie f2) {
    // codul pentru functie
}

fractie operator-(fractie f1, fractie f2) {
    // codul pentru functie
}

fractie operator/(fractie f1, fractie f2) {
    // codul pentru functie
}

void print() { cout << p << q; }

//fisierul pr.c - utilizarea clasei fractie
#include "frac.h"

void Prelucrare() {
    fractie f1(1, 2), f2(7, 4), f3;
    f3 = f1*f2 + f1/f2;
    f3.Print();
}
```

```
    // ...  
}
```

O problemă importantă privind abstractizarea datelor o constituie *parametrizarea datelor*. Atât limbajul Ada, cât și C++ oferă suport pentru parametrizare.

Problema parametrizării apare atunci când se dorește definirea unor tipuri de date generice, la care tipul elementelor componente este neprecizat (el este generic, fiind privit ca un parametru). În acest mod un tip de date abstract parametrizat reprezintă un *tip de date abstract generic*.

Exemplul 1.5. De exemplu, dacă se dorește definirea unei clase *vector* la care tipul componentelor este generic, în C++ se poate proceda astfel:

```
class vector<class T> {  
    T *v;    // tabloul componentelor cu tipul generic T  
    int dim; // dimensiunea vectorului  
public:  
    // constructor  
    vector(int n) {  
        if (n > 0)  
            v = new T[dim = n];  
    }  
    // supraancarcarea operatorului de indexare  
    T& operator[](int k) { return v[k]; }  
    int dimensiune() { return dim; }  
};
```

Se pot defini acum vectori care conțin elemente aparținând unui tip specificat:

```
vector<int> v1(20); // vector cu 20 componente întregi  
vector<double> v2(10); // vector cu 10 componente reale  
v1[7] = 5;  
v2[7] = 2.3;
```

1.4 Programarea orientată pe obiecte

După cum s-a specificat anterior, noțiunea de clasă nu este specifică doar programării orientate pe obiecte, ea fiind de fapt o modalitate de implementare a unui tip abstract de date. Proprietățile unei clase se pot descrie atât prin date (*atribute*), cât și prin funcții (*metode*).

Instanțele unei clase sunt numite *obiecte* și din acest motiv, o clasă reprezintă un mod de descriere a proprietăților și a modului de comportare al unei mulțimi de obiecte. Un obiect este identificat în mod unic prin *numele* său, ceea ce impune faptul că nu pot exista două obiecte diferite ale unei clase cu același nume.

De exemplu, definirea a trei obiecte instanță ale clasei *fractie* se pot realiza astfel:

```
fractie f1(1, 2), f2(7, 4), f3;
```

Deoarece o clasă încapsulează atât date cât și funcții, pentru un obiect instanță pot fi accesate atât atributele, cât și metodele. De exemplu, secvența :

```
f3.Print();
```

apelează metoda *Print* a clasei *fractie*.

Un obiect al unei clase definește de fapt **starea obiectului** respectiv, reprezentată de valorile atributelor sale la un anumit moment de timp. În timpul execuției unui program, starea unui obiect instanță se poate modifica prin modificarea valorilor atributelor.

Comportamentul unui obiect este definit prin mulțimea metodelor ce se pot aplica asupra acestuia. Aceste metode nu sunt specifice obiectului, ci sunt comune într-una din clase de obiecte. O metodă descrie de fapt modul în care un obiect reacționează în momentul în care obiectul recepționează un anumit **mesaj**. Un mesaj este o cerere către un obiect al unei clase de invocare a unei metode specifice a obiectului.

În terminologia utilizată în domeniul paradigmei programării orientate pe obiecte, apelul unei funcții membru a unui obiect instanță înseamnă **trimiterea unui mesaj** acelui obiect. Din acest punct de vedere, scopul utilizării acestei paradigme constă în dezvoltarea unor aplicații care creează anumite mulțimi de obiecte cărora apoi le transmit diferite mesaje.

Deși elementele definite anterior sunt fundamentale pentru aplicațiile care lucrează cu clase și obiecte, ele nu reprezintă aspectele esențiale ale **paradigmei programării orientate pe obiecte**.

Un prim element esențial al programării orientate pe obiecte este faptul că *permite exprimarea distincției între proprietățile generale și cele particulare ale obiectelor*. De exemplu, toate autoturismele de tip Dacia folosesc au 4 roți (proprietate generală), dar autoturismele Dacia 1310 au capacitatea motorului de 1300 cm³, pe când cele de tip Dacia 1410 au capacitatea de 1400 cm³ (proprietate particulară).

Rezultă de aici o proprietate importantă a limbajelor orientate pe obiecte : permit împărțirea obiectelor în clase, precum și un mecanism de **moștenire** a proprietăților unei clase de către alte clase. În acest mod, clasele de obiecte pot forma **ierarhii de clase**. De exemplu, în cazul autoturismelor Dacia, dacă se definește o ierarhie formată din 3 clase, Dacia, Dacia1310 și Dacia 1410, ultimele două clase vor moșteni proprietățile primeia.

Exemplul 1.6. Ierarhia de clase pentru autoturismele Dacia se poate descrie astfel :

```
class Dacia {
    // ...
    int nr_roti;
    Dacia() { nr_roti = 4; }
    // ...
};

class Dacia1310: public Dacia {
    // ...
    int capacitate;
    Dacia1310() { capacitate = 1300; }
    // ...
};
```

```

class Dacia1410: public Dacia {
    // ...
    int capacitate;
    Dacia1410() { capacitate = 1400; }
    // ...
};

```

Exemplul 1.7. Să considerăm următoarele figuri poligonale în plan : triunghiuri, patrulatere, patrulatere, pentagoane, etc., fiecare poligon fiind descris prin coordonatele vârfurilor în ordine trigonometrică. Se poate forma o ierarhie de clase, care în vârful ierarhiei clasa *poligon*, celelalte clase moștenind-o. Să presupunem că pentru clasa *poligon* se specifică coordonatele primelor 2 vârfuri ale poligonului, $P_0(x_0, y_0)$ și $P_1(x_1, y_1)$, celelalte clase trebuind să memoreze pe rând coordonatele celorlalte vârfuri.

```

class Poligon {
    // ...
    double x0, y0, x1, y1;
    virtual double Perimetru();
    virtual double DouaLaturi() = 0;
    // ...
};

class Triunghi: public Poligon {
    // ...
    double x2, y2;
    double Perimetru();
    double DouaLaturi();
    double OLatura();
    // ...
};

class Patrulater: public Triunghi {
    // ...
    double x3, y3;
    double Perimetru();
    double DouaLaturi();
    // ...
};

```

Toate poligoanele au un perimetru, indiferent de tipul lor. Determinarea valorii acestuia se poate face incremental, observând faptul că perimetrul unui poligon cu $n + 1$ vârfuri (P_0, P_1, \dots, P_n) se poate obține din perimetrul poligonului cu n vârfuri (P_0, P_1, \dots, P_{n-1}) la care se adauga lungimile laturilor P_0P_n și $P_{n+1}P_n$. Funcția *Perimetru* determină valoarea perimetrului curent, iar funcția *DouaLaturi* determină lungimile laturilor P_0P_n și $P_{n-1}P_n$.

Deoarece aceste funcții sunt comune tuturor claselor, dar implementările specifice fiecărei clase, ele reprezintă **funcții virtuale**. Mai mult, funcția *DouaLaturi* nu se poate implementa în cadrul clasei *Poligon*, ea reprezentând o **funcție virtuală pură** în acea clasă.

Implementările acestor funcții se pot descrie astfel :


```

double Poligon::Perimetru() {
    double l = sqrt((x0-x1)*(x0-x1) + (y0-y1)*(y0-y1));
    return l;
}

double Triunghi::DouaLaturi() {
    double a, b;
    a = sqrt((x0-x2)*(x0-x2) + (y0-y2)*(y0-y2));
    b = sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
    return a + b;
}

double Triunghi::OLatura() {
    double a;
    a = sqrt((x0-x2)*(x0-x2) + (y0-y2)*(y0-y2));
    return a;
}

double Triunghi::Perimetru() {
    double p;
    p = Poligon::Perimetru() + DouaLaturi();
    return p;
}

double Patrulater::DouaLaturi() {
    double a, b;
    a = sqrt((x0-x3)*(x0-x3) + (y0-y3)*(y0-y3));
    b = sqrt((x1-x3)*(x1-x3) + (y1-y3)*(y1-y3));
    return a + b;
}

double Patrulater::Perimetru() {
    double p;
    p = Triunghi::Perimetru() -
        Triunghi::OLatura() +
        DouaLaturi();
    return p;
}

```

În cazul funcțiilor virtuale, selectarea efectivă a funcției care se va apela se realizează în mod automat de către compilator. Această proprietate care permite compilatorului să selecteze pentru execuție o anumită metodă specifică unui obiect, dintre mai multe metode cu același nume definite în ierarhia de clase se numește **polimorfism**.

În concluzie, al doilea element esențial al programării orientate pe obiecte constă în *mecanismul funcțiilor virtuale* și al *polimorfismului*, prin care apelul funcțiilor membru depinde efectiv de tipul obiectului respectiv.

De exemplu, pentru cazul poligoanelor, următoarele declarații și instrucțiuni :

```

Triunghi f1;

```

```
    Patrulater f2;  
    double p1 = f1.Perimetru();  
    double p2 = f2.Perimetru();
```

permit selectarea corectă a funcției *Perimetru* de către compilator pentru fiecare obiect.

Limbajul considerat ca un limbaj pur orientat pe obiecte este Smalltalk. Printre alte limbaje ce suportă această paradigmă se pot enumera : Ada, C++, Simula, Java.

Observație. Datorită mecanismului claselor, un limbaj ce suportă paradigma programării orientate pe obiecte, permite și abstractizarea datelor.

1.5 Proiectarea orientată pe obiecte

Cunoașterea unui limbaj de programare care oferă suport pentru paradigma programării orientate pe obiecte este necesară, însă nu și suficientă pentru realizarea unor aplicații orientate pe obiecte.

Pentru a putea dezvolta în mod eficient o asemenea aplicație, în special în cazul proiectelor mari, sunt necesare de asemenea stăpânirea unor tehnici de *analiză și proiectare software*. Acestea permit ca plecând de la o problemă reală, să se poată dezvolta o aplicație orientată pe obiecte pentru rezolvarea problemei inițiale.

În mod uzual, *ciclul de viață* al produselor software conține următoarele etape :

- analiza
- proiectarea
- codificarea
- testarea
- întreținerea

Acestea reprezintă etapele unui ciclu ideal de viață, deoarece în practică, în funcție de tipul problemelor de rezolvat și de uneltele de care se dispune, unele etape sunt nesemnificative. De exemplu, în cazul unei probleme simple pe care o poate primi un student în timpul unei ore de laborator, operația de întreținere a programului nu se justifică, iar operațiile de analiză și proiectare pe de o parte, precum și cele de codificare și testare pe de altă parte, pot fi comasate.

Dacă necesitatea cunoașterii unui limbaj orientat pe obiecte este indispensabil în etapa de codificare a unei aplicații orientată pe obiecte, cunoașterea unor metode de *analiză și proiectare orientată pe obiecte* este esențială în etapele de analiză și proiectare.

Unul dintre cele mai utilizate instrumente de analiză și proiectare orientată pe obiecte o constituie *formalismul UML (The Unified Modeling Language)*, propus de Jacobson, Rumbaugh și Booch. În continuare se prezintă câteva elemente ale formalismului UML utilizate în etapele de analiză și proiectare software.

În etapa de *analiză*, trebuie evidențiate toate elementele care trebuie să fie realizate de către aplicație. În mod uzual se pornește de la o specificație inițială a problemei, enunțată de către client, iar în această etapă trebuie create două documente : *Analiza cerințelor* și *Specificarea sistemului*.

Analiza cerințelor reprezintă o listă a cerințelor enunțate de către client și pe care aplicația trebuie să le îndeplinească, iar specificarea sistemului reprezintă o descriere a ceea ce aplicația trebuie să facă pentru a satisface cerințele clientului.

Elementul esențial în această etapă constă în descoperirea comportării aplicației conform cerințelor specificate. Cel mai utilizat instrument de proiectare în această etapă o constituie **diagramele de caz**. O diagramă de caz descrie comportarea unei părți din funcționarea unui sistem din punctul de vedere al unui utilizator extern. Ea folosește noțiunile de **actor** și **caz de utilizare (use case)**.

Un **actor** este o idealizare a unei persoane externe sau proces care interacționează cu sistemul și în mod uzual se descrie grafic ca o persoană stilizată care are asociat un nume. Un caz de utilizare reprezintă un element al funcționalității externe a sistemului, care poate interacționa cu unul sau mai mulți actori. În plus, un caz de utilizare poate avea legături cu alte cazuri de funcționare, pentru a putea descrie mai bine relațiile sale din cadrul sistemului. Grafic, un caz de utilizare se descrie printr-o elipsă ce are specificat în interior numele cazului.

Observație. O legătură între un actor și un caz de utilizare poate indica în multe cazuri un element al unei interfețe utilizator.

O **diagramă de utilizare** descrie cazurile de utilizare asociate unui sistem și le încadrează din punct de vedere grafic într-un dreptunghi reprezentând sistemul respectiv. De exemplu, în figura 1.1 este prezentată o diagramă de caz ce descrie funcționarea unui automat bancar. Automatul bancar reprezintă sistemul ce se descrie, iar actorii externi sunt clientul și banca posesoare a automatului.

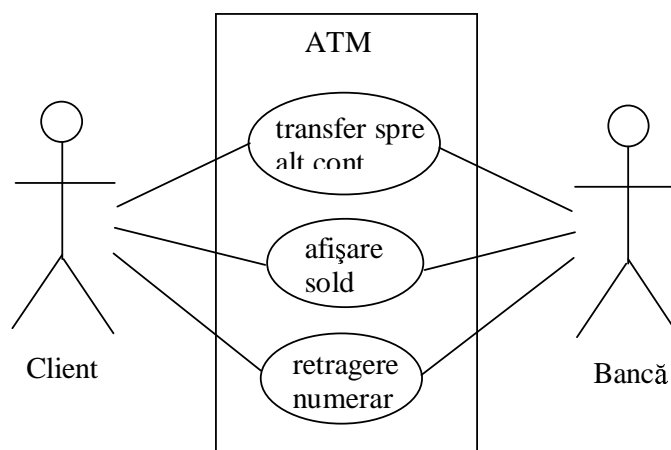


Figura 1.1

Diagramele de caz permit specificarea cerințelor și a sistemului prin determinarea tuturor interacțiunilor într-un utilizator și sistem. Pentru specificarea tuturor cerințelor, trebuie descoperite toate diagramele de cazuri ce descriu funcționarea sistemului respectiv.

În etapa de **proiectare** a aplicației trebuie determinate în primul rând componentele aplicației respective. O **componentă** reprezintă o entitate abstractă, iar aplicația poate fi privită ca fiind alcătuită din aceste componente. Din punctul de vedere al unui limbaj de programare, o componentă poate fi reprezentată de o funcție, de o structură sau o clasă, sau de o colecție de alte componente. În cele mai multe cazuri, componentele se identifică cu clasele de obiecte.

O componentă trebuie să aibă asociată o mulțime bine definită de responsabilități, precum și o mulțime de alte componente cu care poate interacționa. O metodă foarte utilizată, în special înainte de apariția formalismului UML a fost metoda CRC (Component – Responsibility – Collaboration). Folosind această metodă, fiecare componentă se asociază cu o carte de joc pe care se scriu următoarele informații :

- numele componentei
- descrierea responsabilităților componentei respective
- lista celorlalte componente cu care ea interacționează.

Avantajul acestei metode constă în simplitatea ei și în faptul că permite operația de rafinare ori de câte ori este nevoie. Deoarece în mod uzual o componentă reprezintă o clasă de obiecte, responsabilitățile acesteia sunt de fapt metodele clasei respective. În acest mod, o carte CRC reprezintă descrierea unei clase. Există totuși un dezavantaj al metodei CRC, deoarece ea nu permite descrierea atributelor acestor clase.

Formalismul UML pune la dispoziție **diagramele de clasă** pentru descrierea claselor și a relațiilor dintre ele, cu observația că ele reprezintă descrieri statice.

Notăția UML pentru o clasă este o notație grafică, în care o clasă se specifică într-un dreptunghi prin numele său, precum și prin listele de atribute și metode specifice clasei. Colaborările dintre clase se pot specifica prin intermediul **relațiilor**. Relațiile se reprezintă grafic prin linii sau săgeți, fiecare asemenea relație legând două clase între ele. De fapt relațiile sunt generale în cadrul formalismului UML, ele putând conecta diferite tipuri de elemente, precum clase, actori, cazuri de utilizare, etc.

Principalele tipuri de relații sunt următoarele:

- *asocierea*, care descrie legăturile semantice dintre obiecte individuale ale unor clase;
- *generalizarea*, care descrie relațiile de moștenire între clase;
- *realizarea*, care descrie relația dintre o specificare și implementarea sa;
- *fluxul*, care descrie relația dintre două versiuni ale aceluiași obiect la momente de timp diferite;
- *dependența*, care pune în relație clase ale căror comportament sau implementare afectează alte clase.

Asocierea este singura relație care descrie conexiunea între obiecte sau grupuri de obiecte, toate celelalte punând în relație descrierea claselor și nu a instanțelor acestora. Un caz particular de relație de asociere reprezintă relația de *agregare* sau *compunere*. O asemenea relație descrie un obiect compus este agregat din unul sau mai multe obiecte componente.

Asupra principalelor tipuri de relație se va reveni în capitolele următoare. De exemplu, ierarhia de clase din exemplul 1.7 se poate descrie ca în figura 1.2.

Formalismul UML pune la dispoziție și alte tipuri de diagrame, care permit descrierea interacțiunilor dintre obiecte, precum și evoluția dinamică a obiectelor sistemului. De exemplu, **diagramele de colaborare** permit reprezentarea obiectelor și a interacțiunilor dintre ele, iar **diagramele de secvențe** permit reprezentarea temporală a obiectelor și a interacțiunilor lor.

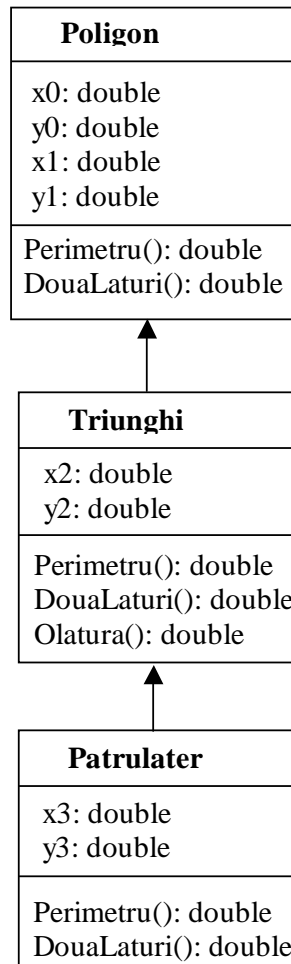


Figura 1.2