

Capitolul 7

Compunerea obiectelor

O problemă importantă în cazul paradigmei programării este cea referitoare la **reutilizarea codului**. În cazul proiectării unei clase noi, este mai simplă și mai eficientă folosirea anumitor clase create și implementate deja, decât rescrierea de către programator a segmentelor de cod corespunzătoare acestora.

În mod uzual se folosesc două metode principale când se reutilizează un cod existent: **compunerea obiectelor** și **moștenirea claselor**. În cazul primei metode, obiectele instanță ale unei anumite clase sunt compuse din unul sau mai multe obiecte aparținând altor clase, ceea ce impune ca în declarația clasei respective anumite date membru să fie obiecte din alte clase.

Metoda reutilizării codului prin compunerea obiectelor nu este o metodă specifică programării orientate pe obiecte, deși ea este des folosită în cadrul acestei paradigme. Moștenirea claselor însă este o metodă specifică acestei paradigme și se bazează pe un mecanism de ierarhizare și derivare a claselor. O clasă derivată poate moșteni anumite date sau funcții membru ale clasei din care este derivată.

O deosebire importantă între cele două metode constă în modul de utilizare a claselor existente. În cazul moștenirii se dorește în general să se utilizeze interfața acestora, relația de moștenire fiind o relație de rafinare: o clasă derivată este privită ca un subtip al unei clase de bază, care adaugă la aceasta o informație mai specifică. În cazul compunerii, relația folosită este una de apartenență. O clasă compusă utilizează proprietățile claselor componente, fără să moștenească și interfața acestora.

7.1 Definirea și utilizarea obiectelor compuse

Compunerea este o *relație de asociere* a claselor, în care clasa compusă trebuie să-și gestioneze obiectele membru componente, cum este de exemplu operația de creare și distrugere a acestora.

În limbajul UML relația de compunere se reprezintă grafic astfel:



semnul grafic fiind orientat spre clasa compusă.

Fiind o relație de asociere, în mod adițional, pe extremitățile drepte ce unește cele două clase se pot specifica numărul obiectelor instanță care se asociază prin această relație. De exemplu, pentru două clase, A și B, între care există relația specificată în figura 7.1, o

instanță a clasei A se asociază cu 7 instanțe ale clasei B, iar fiecare instanță a clasei B se asociază cu o singură instanță a clasei A.

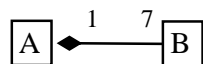


Figura 7.1

De exemplu, pentru ierarhia de clase de mai jos:

```
class A;
class B;
class X {
    A a;
    B b[12];
    // ...
};
```

diagrama de clase se poate reprezenta ca în figura 7.2.

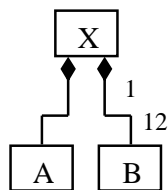


Figura 7.2

Obiectele ce sunt date membru într-o clasă compusă pot fi date publice sau private. În mod uzual, obiectele componente dintr-o clasă compusă sunt private, deoarece acestea sunt utilizate în clasa compusă doar funcționalitățile claselor asociate, nu și interfața lor.

În cazul obiectelor publice, membrii acestora pot fi referiți direct prin intermediul operatorului de rezoluție.

Exemplul 7.1.

```
class A {
public:
    int n;
    A();
    // ...
};

class B {
public:
    double z;
    void f();
    B();
    // ...
};
```

```

class X {
    char c;
public:
    A a;
    B b;
    X();
    // ...
};

void Prelucrare () {
    X x;
    x.a.n = 3;
    x.b.z = 4.5;
    x.b.f();
};

```

În cazul în care obiectele componente sunt private, membrii acestora nu mai pot fi accesați direct prin intermediul operatorului de selecție. Clasa compusă trebuie să conțină în plus funcții proprii de accesare a membrilor obiectelor componente.

Funcțiile de acces ale clasei compuse pot eventual să supraîncarce funcțiile membru ale claselor componente datorită regulii domeniului de vizibilitate. Dar în cazul în care clase componente diferite conțin membrii cu același nume, funcțiile de acces din clasa compusă trebuie să fie distincte.

Exemplul 7.2.

```

class A {
    int n;
public:
    int N() const { return n; }
    void g() { cout << "functia g in clasa A" << endl; }
    A(int k = 0) { n = k; }
    // ...
};

class B {
    double z;
public:
    double Z() const { return z; }
    void f() { cout << "functia f in clasa B" << endl; }
    void g() { cout << "functia g in clasa B" << endl; }
    B(double k=0) { z = k; }
    // ...
};

class X {
    char c;
    A a;
    B b;
public:

```

```

    void f() { b.B::f(); }
    void ga() { a.A::g(); }
    void gb() { b.B::g(); }
    X();
    // ...
};

void Prelucrare () {
    X x;
    x.f();
    x.ga();
    x.gb();
    // ...
}

```

7.2 Crearea și distrugerea obiectelor compuse. Liste de inițializare a membrilor

În cazul în care într-o clasă există ca date membru obiecte instanță ale altor clase, constructorii și destructorii clasei respective trebuie să gestioneze și apelul constructorilor și destructorilor claselor componente.

În momentul creării unui obiect compus, este natural ca înainte de inițializarea celorlalți membri să se inițializeze membrii obiectelor componente.

Limbajul C++ permite această operație prin intermediul *listei de inițializare a membrilor*. O asemenea listă se specifică între antetul constructorului și corpul acestuia. Lista este precedată de caracterul `:` și elementele listei sunt separate prin virgulă.

Un element din lista de inițializare este asociat unui membru al clasei și el se specifică prin numele membrului și argumentele ce se vor transmite constructorului acestuia. De exemplu, pentru clasa *X*, constructorul clasei se poate descrie astfel:

```

X::X(): a(5), b(5.3), c('x')
{
    cout << "constructorul clasei X";
}

```

Apelul constructorilor pentru obiectele *b* și *c* pot fi priviți ca urmatoarele definiții:

```

A a(5); B b(5.5);

```

Numele claselor componente nu mai trebuie specificat în lista de inițializare, deoarece nu pot exista două obiecte membre cu același nume în clasa compusă, deci compilatorul poate determina clasa de apartenență a membrilor din numele acestora.

Rezultă astfel că pentru fiecare element din lista de inițializare se apelează un constructor corespunzător al clasei de apartenență al elementului respectiv.

Observatii:

1. Lista de inițializare a membrilor poate apărea atât în cadrul constructorilor `inline`, cât și în cazul celor definiți în afara declarației clasei, cum a fost în exemplul precedent.
2. Prin extensie, în lista de inițializare pot să apară elemente și pentru membrii ce aparțin unor tipuri de date predefinite. Aceasta extensie permite o sintaxă consistentă ce tratează inițializarea variabilelor ca un pseudo-constructor. În exemplul anterior, elementul `c('x')` este privit ca o inițializare de forma `c='x'`.

Noțiunea de ***pseudo-constructor*** este utilizată în limbajul C++ și pentru crearea unor variabile aparținând unui tip predefinit în afara unei anumite clase:

```
int n(3);
int* p = new int(7);
```

Un stil consecvent de programare impune ca lista de inițializare să aibă un număr de elemente egal cu numărul de date membru ale clasei compuse, asigurând o inițializare sigură a datelor membru înainte de execuția instrucțiunilor din corpul constructorului clasei. Există cazuri însă în care apelul explicit al constructorilor nu este necesar în lista de inițializare. De exemplu, dacă un obiect component are un constructor implicat sau cu parametri implicați și se dorește apelul acestuia (și nu al unui alt constructor), elementul corespunzător de inițializare din lista de inițializare a membrilor, poate să lipsească.

După cum s-a specificat, destructorii obiectelor sunt apelați în momentul distrugerii acestora. Ordinea de apel a destructorilor este întotdeauna inversă ordinii apelului constructorilor.

Rezultă de aici câteva efecte importante în cazul obiectelor compuse:

- 1) Destructorii pentru obiectele componente se apelează după terminarea apelului destructorului obiectului compus. Dacă un obiect component este la rândul său compus din alte obiecte, acest proces are loc recursiv.
- 2) Ordinea de apel al constructorilor din lista de inițializare a membrilor nu este și ordinea în care apar elementele în listă; constructorii și pseudo-constructorii se apelează în ordinea în care obiectele membru apar în declarația clasei.
De exemplu, dacă o clasă compusă are doi constructori în care ordinea apariției componentelor în lista de inițializare este diferită, deoarece clasa poate avea maxim un destructor, acesta nu poate determina ordinea corectă de apel a destructorilor. Cu alte cuvinte, poate există o dependență a destructorilor de tipul constructorului utilizat pentru crearea unui anumit obiect compus. Pentru ca aceasta dependență să fie eliminată, toți constructorii claselor compuse vor apela constructorii obiectelor componente în aceeași ordine (cea a apariției obiectelor în declarația clasei).
- 3) Destructorii obiectelor componente sunt apelați în ordinea inversă în care obiectele sunt declarate în clasa compusă.

Exemplul 7.3.

```
#include <iostream>
using namespace std;
class A1 {
    int p;
public:
    A1(int k = 0) {
        p = k;
    }
};
```

```

        cout << "Constructorul clasei A1" << endl;
    }
    ~A1() { cout << "Destructorul clasei A1" << endl; }
};

class A2 {
    int q;
public:
    A2(int k = 0) {
        q = k;
        cout << "Constructorul clasei A2" << endl;
    }
    ~A2() { cout << "Destructorul clasei A2" << endl; }
};

class A {
    A1 a1;
    A2 a2;
public:
    A(int i = 0, int j = 0): a1(i), a2(j) {
        cout << "Constructorul clasei A" << endl;
    }
    ~A() { cout << "Destructorul clasei A" << endl; }
};

class B {
    double z;
public:
    B(double k = 0): z(k) {
        cout << "Constructorul clasei B" << endl;
    }
    ~B() { cout << "Destructorul clasei B" << endl; }
};

class X {
    A a;
    B b;
public:
    X(int i = 0, int j = 0, double k = 0): a(i, j), b(k)
    {
        cout << "Constructorul clasei X" << endl;
    }
    ~X() { cout << "Destructorul clasei X" << endl; }
};

void main() {
    cout << "Incepe functia main" << endl;
    {
        X x(7,9,3.33);
        cout << "S-a creat obiectul x" << endl;
    }
}

```

```
    cout << "Se termina functia main" << endl;  
}
```

Ieșirea programului este următoarea:

```
Incepe functia main  
Constructorul clasei A1  
Constructorul clasei A2  
Constructorul clasei A  
Constructorul clasei B  
Constructorul clasei X  
S-a creat obiectul x  
Destructorul clasei X  
Destructorul clasei B  
Destructorul clasei A  
Destructorul clasei A2  
Destructorul clasei A1  
Se termina functia main.
```