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 şi mai eficientă folosirea anumitor clase create şi 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*** şi ***moştenirea claselor***. În cazul primei metode, obiectele instanţă ale unei anumite clase sunt compuse din unul sau mai multe obiecte aparţinând altor clase, ceea ce impune ca în declaraţia clasei respective anumite date membru să fie obiecte din alte clase.

Metoda reutilizarii codului prin compunerea obiectelor nu este o metodă specifică programării orientate pe obiecte, deşi ea este des folosită în cadrul acestei paradigme. Moştenirea claselor însă este o metodă specifică acestei paradigme şi se bazează pe un mecanism de ierarhizare şi derivare a claselor. O clasă derivată poate moşteni anumite date sau funcţii membru ale clasei din care este derivată.

O deosebire importatntă între cele două metode constă în modul de utilizare a claselor existente. În cazul moştenirii se doreşte în general să se utilizeze interfaţa acestora, relaţia de moştenire fiind o relaţie de rafinare: o clasă derivată este privită ca un subtip al unei clase de bază, cara adaugă la aceasat o informaţie mai specifică. În cazul compunerii, relaţia folosită este una de apartenenţă. O clasă compusă utilizează proprietăţile claselor componente, fără să moştenească şi interfaţa acestora.

7.1 Definirea şi utilizarea obiectelor compuse

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

În limbajul UML relaţia de compunere se reprezintă grafic astfel:

semnul grafic fiind orientat spre clasa compusă.

Fiind o relaţie de asociere, în mod adiţional, pe extremităţile dreptei ce uneşte cele două clase se pot specifica numărul obiectelor instanţă care se asociază prin această relaţie. De exemplu, pentru două clase, A şi B, între care există relaţia 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.

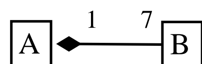

Figura 7.1

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.

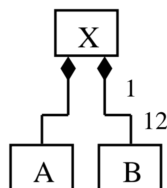

Figura 7.2

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 şi 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 accesaţi direct prin intermediul operatorului de selecţie. Clasa compusă trebuie să conţină în plus funcţii proprii de accesare a membrilor obiectelor componente.

Funcţiile de acces ale clasei compuse pot eventual să supraâncarce funcţiile membru ale claselor componente datorită regulii domeniului de vizibilitate. Dar în cazul în care clase componente diferite conţin membrii cu acelaşi nume, funcţiile 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 şi distrugerea obiectelor compuse. Liste de iniţializare a membrilor

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

În momentul crearii unui obiect compus, este natural ca înainte de iniţializarea celorlalţi membri să se iniţializeze membrii obiectelor componente.

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

Un element din lista de iniţializare este asociat unui membru al clasei şi el se specifică prin numele membrului şi 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* şi *c* pot fi priviţi ca urmatoarele definiţii:

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

Numele claselor componente nu mai trebuie specificât în lista de iniţializare, deoarece nu pot exista două obiecte membre cu acelaşi nume în clasa compusă, deci compilatorul poate determina clasa de apartenenţă a mambrilor din numele acestora.

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

Observatii:

1. Lista de inițializare a membrilor poate aparea atât în cadrul constructorilor `inline`, cât şi 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 şi pentru membrii ce aparţin 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++ şi pentru crearea unor variabile aparţinând 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 implicit sau cu parametri impliciti şi se doreşte apelul acestuia (şi nu al unui alt constructor), elementul corspunzător 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 dupa 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 şi ordinea în care apar elementele în listă; constructorii şi 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 corecta 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 eliminata, toți constructorii claselor compuse vor apela constructorii obiectelor componente în aceeaşi ordine (cea a apariției obiectelor în declarația clasei).
3) Destructorii obiectelor componente sunt apelați în ordinea inversa 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;
  }
```

Ieşirea 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.
```