Cuprins

REȚINEȚI!

Clasele şi funcţiile parametrizate ne permit să elaborăm porţiuni de cod care pot fi configurate pentru a lucra cu tipuri de date diferite

- Ele devin utile în cazul în care aceleaşi secvenţe de prelucrare trebuie aplicate unor obiecte de tipuri diferite.
- Compilatorul va genera secvenţe de cod distincte pentru fiecare set de parametri cu care este utilizată o clasă sau o funcţie parametrizată.

Agregarea

Prin agregare înțelegem procesul de definire a unor noi clase de obiecte prin înglobarea în cadrul acestora a unor date-membru care au ca tip clase deja existente. Nu se impun restricții referitoare la numărul şi tipurile datelor-membru utilizate în definirea unei clase prin intermediul agregării. Cu alte cuvinte, în definirea unei clase noi putem folosi oricâte alte clase deja existente ca tipuri pentru datele-membru ale noii clase. Agregarea ne permite să construim clase complex structurate pornind de la clase mai simple.

Exemplu: considerăm cazul unei sesiuni de comunicări ştiinţifice. O astfel de sesiune este caracterizată de o denumire de tip `string`, de numele persoanei care moderează sesiunea, de asemenea de tip `string` şi de lista participanţilor la sesiune. Toate aceste trei date-membru au ca tip clase care sunt deja definite cum ar fi clasa `string` sau clasa `list`, ambele definite în bibliotecile standard C++. Astfel, putem defini o clasă nouă `Sesiune` pornind de la clasele deja existente `string` şi `list`. Fiecare obiect de tip `Sesiune` include propriile *subobiecte* `_denumire`, `_moderator` din clasa `string`, precum şi propria listă de participanţi. Aceste subobiecte sunt create în momentul creării obiectului `Sesiune` şi sunt distruse odată cu acesta.

Varianta de agregare în care subobiectele agregate aparţin în mod exclusiv agregatului din care fac parte, iar durata lor de existenţă coincide cu cea a agregatului este denumită şi compoziţie. Următoarea diagramă de clase UML ilustrează relaţia de compoziţie între clasele `Sesiune` şi `list`. Compoziţia este reprezentată cu ajutorul unei linii continue care are un romb plin la capătul dinspre noua clasă obţinută prin compoziţie (la capătul dinspre clasa container – agregat).

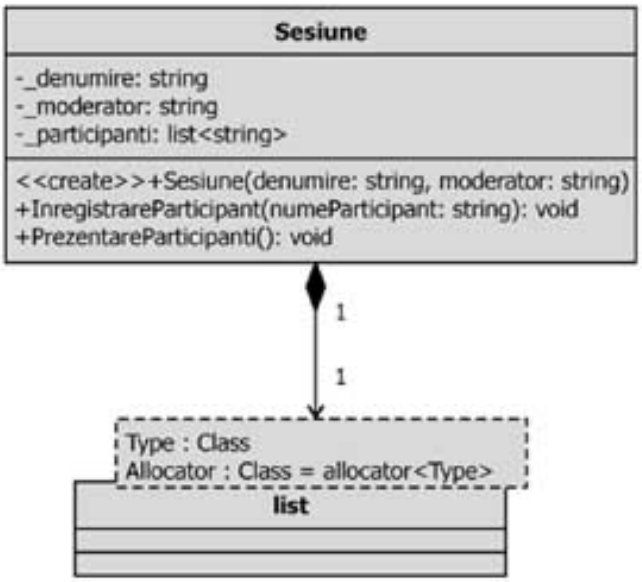

Figura 14. Clasa Sesiune este construită cu ajutorul compoziţiei pornind de la clasele string şi list

Relaţia de compoziţie mai poartă numele de relaţie *has a* (relaţie *are un/are o*). Într-adevăr, putem spune că un obiect *Sesiune* are o listă de participanţi. Un alt nume este acela de relaţie *is a part of* (relaţie *este o parte*).

Există două variante de a prezenta într-o diagramă de clase UML data-membru implicată într-o relaţie de compoziţie. Să luăm, de exemplu, data-membru `_participanti` din diagrama anterioară. Ea poate fi asociată cu linia continuă reprezentând compoziţia la capătul dinspre clasa parte `list`.

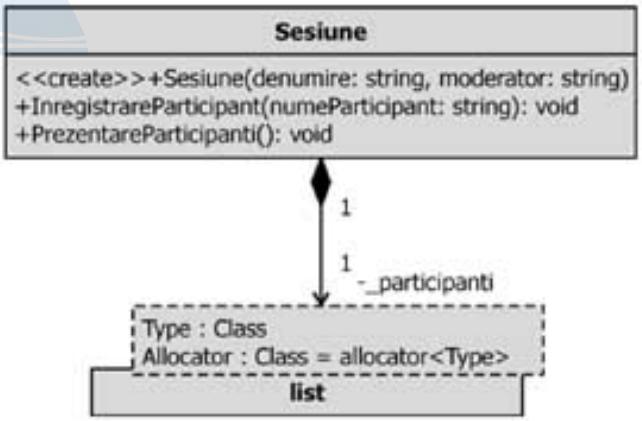

Figura 15. Data membru _participanţi poate fi asociată în mod explicit cu relaţia de compoziţie

Următorul program ne arată cum se implementează efectiv relaţia de compoziţie în cadrul unei aplicaţii C++. Programul poate fi o parte a unei aplicaţii complexe utilizată de o firmă ce organizează evenimente.

```
//Nume fisier: sesiune.h

#if !defined(__SESIUNE_H__)
#define __SESIUNE_H__

#include <iostream>
#include <string>
#include <list>
using namespace std;

/**
 * Clasa Sesiune este construita cu ajutorul agregarii
 * pornind de la clasele string si list.
 */
class Sesiune
{
public:
      Sesiune(const string& denumire,
            const string& moderator)
      :_denumire(denumire),_moderator(moderator)
      {
            cout<<"Initializare instanta Sesiune."
                  <<endl<<"Denumire:"<<_denumire<<endl
                  <<"Moderator:"<<_moderator<<endl;
```

```
        }
        void InregistrareParticipant(
              const string& numeParticipant);
        void PrezentareParticipanti() const;
private:
        string _denumire;
        string _moderator;
        /**
         * Fiecare sesiune poseda propria lista de
         * participanti.
         */
        list<string> _participanti;
};

#endif
```

```
//Nume fisier: sesiune.cpp

#include "sesiune.h"

void Sesiune::InregistrareParticipant(const string&

numeParticipant)
{
        // Adauga noul participant la sfarsitul listei.
        _participanti.push_back(numeParticipant);
}

void Sesiune::PrezentareParticipanti() const
{
        list<string>::const_iterator iter;

        cout<<"Sesiunea: "<<_denumire<<endl;
        cout<<"Moderator: "<<_moderator<<endl;
        cout<<"Lista participanti"<<endl;

        for(iter = _participanti.begin();
              iter != _participanti.end(); iter++)
        {
              cout<<*iter<<endl;
        }
}
```

```
//Nume fisier: conferinta.cpp

#include "sesiune.h"
```

```
int main()
{
	Sesiune oop("Programare orientata obiect",
		"Mircea Preda");

	oop.InregistrareParticipant("Daniel Popescu");
	oop.InregistrareParticipant("Maria Marinescu");

	oop.PrezentareParticipanti();

	Sesiune pp("Programare procedurala",
		"Cristian Ionescu");

	pp.InregistrareParticipant("Paul Ionescu");
	pp.InregistrareParticipant("Daniela Iliescu");

	pp.PrezentareParticipanti();

	return 0;
}
```

```
C:\Windows\system32\cmd.exe
Initializare instanta Sesiune.
Denunire:Programare orientata obiect
Moderator:Mircea Preda
Sesiunea: Programare orientata obiect
Moderator: Mircea Preda
Lista participanti
Daniel Popescu
Maria Marinescu
Initializare instanta Sesiune.
Denunire:Programare procedurala
Moderator:Cristian Ionescu
Sesiunea: Programare procedurala
Moderator: Cristian Ionescu
Lista participanti
Paul Ionescu
Daniela Iliescu
Press any key to continue . . .
```

Figura 16. Rezultatele execuției aplicației Conferința – Sesiune comunicări

În alte contexte, obiectele-parte (subobiectele) implicate într-o relație de agregare au o existență independentă de cea a agregatului. Mai mult, un obiect parte poate fi partajat de mai multe agregate diferite. Pentru a exemplifica, să definim o clasă nouă `Student` încorporând în cadrul acesteia un pointer către o clasă preexistentă `list` reprezentând lista de cursuri urmate. Obiectul *listă de cursuri*

exista înainte ca un obiect-agregat `Student` să își înceapă existența. Mai multe obiecte `Student` conțin pointeri indicând aceeași listă de cursuri. Dacă un obiect `Student` își încheie existența, lista de cursuri inclusă nu este afectată. Relațiile de acest tip, în care obiectele parte pot fi partajate de mai multe obiecte-agregat se numesc relații de agregare propriu-zisă. Următoarea diagramă de clase UML ilustrează relația de agregare între clasele Student și list.

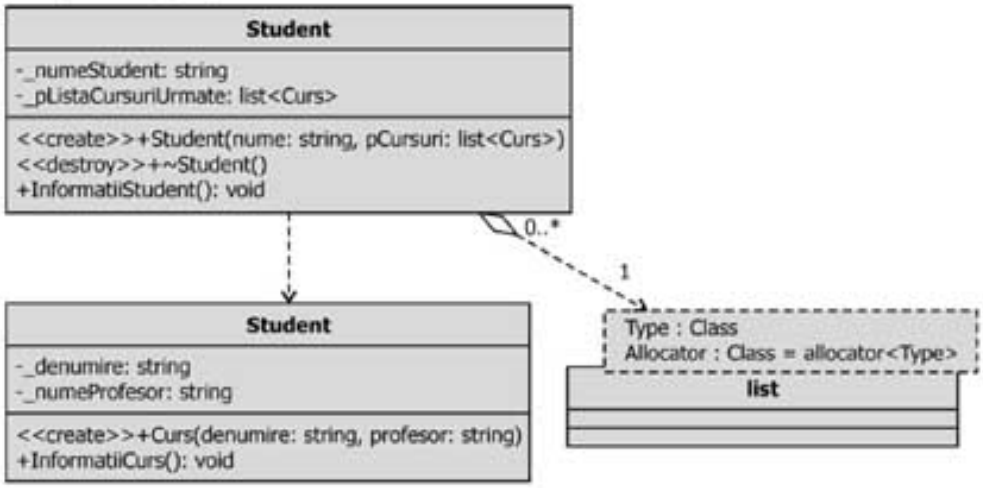

Figura 17. Între clasele Student și list există o relație de agregare

Agregarea este reprezentată printr-o linie continuă care are la capătul dinspre clasa agregată (clasa container) un romb gol. Multiplicitatea 0..* arată că aceeași listă de cursuri poate fi partajată de 0, 1 sau mai multe obiecte `Student`. Următorul program exemplifică modul de implementare a unei relații de agregare.

```
//Nume fisier: curs.h

#if !defined(__CURS_H__)
#define __CURS_H__

#include <list>
#include <string>
#include <iostream>
using namespace std;

class Curs
{
public:
```