

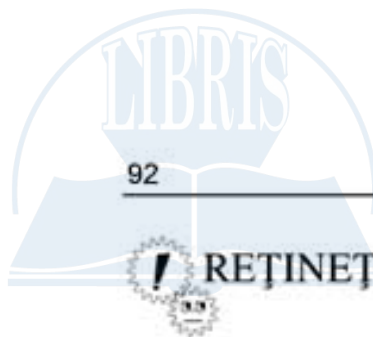


Cuprins

Prefață	7
CAPITOLUL 1. Obiectual vs procedural	11
Test-grilă	38
CAPITOLUL 2. Clase și obiecte	41
Test-grilă	50
Exerciții	51
CAPITOLUL 3. Inițializarea și distrugerea obiectelor	53
Constructori implicați	56
Constructori de copiere	58
Constructori pentru realizarea conversiei de tip (cast)	61
Destructori	64
Test-grilă	66
Exerciții	71
CAPITOLUL 4. Reutilizarea codului-sursa	73
Controlul accesului la elementele unei clase	73
Clase și funcții parametrizate	80
Agregarea	92
Moștenirea	101
Polimorfism și funcții virtuale	117
Test-grilă	130
Exerciții	133
CAPITOLUL 5. Tratarea excepțiilor	135
Throw, try și catch	138
Test-grilă	152
Exerciții	156



CAPITOLUL 6. Șabloane de proiectare	157
Șablonul de proiectare Compozit	158
Șablonul de proiectare Decorator	173
Șablonul de proiectare Observator	183
Șablonul de proiectare Strategie	190
Șablonul de proiectare Singleton	198
Șablonul de proiectare Proxy	208
Șablonul de proiectare Iterator (Cursor)	213
Test-grilă	224
Exerciții	227
CAPITOLUL 7. Dezvoltarea orientată-obiect a aplicațiilor	229
Exemplu de proiectare - analizor sintactic	231
Analiza cerințelor	232
Elaborarea proiectului	233
Implementarea soluției	238
Întreținerea soluției	250
Exemplu de proiectare - generarea cheilor RSA	250
Noțiuni fundamentale	252
Dezvoltarea aplicației	252
Analiza cerințelor	253
Elaborarea proiectului - etapa 1	254
Implementarea soluției - etapa 1	257
Întreținerea soluției	261
Elaborarea proiectului - etapa 2	261
Implementarea soluției - etapa 2	263
Exerciții	265
ANEXĂ. Dezvoltarea aplicațiilor C++ utilizând mediul de dezvoltare Microsoft Visual C++	267
Bibliografie	275

**! REȚINEȚI!**

Clasele și funcțiile parametrizate ne permit să elaborăm porțiuni de cod care pot fi configurate pentru a lucra cu tipuri de date diferite

- Ele devin utile în cazul în care aceleași secvențe de prelucrare trebuie aplicate unor obiecte de tipuri diferite.
- Compilatorul va genera secvențe de cod distincte pentru fiecare set de parametri cu care este utilizată o clasă sau o funcție 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 și 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 științifice. O astfel de sesiune este caracterizată de o denumire de tip `string`, de numele persoanei care moderează sesiunea, de asemenea de tip `string` și de lista participanților 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` și `list`. Fiecare obiect de tip `Sesiune` include propriile `subobiecte` `_denumire`, `_moderator` din clasa `string`, precum și propria listă de participanți. Aceste subobiecte sunt create în momentul creării obiectului `Sesiune` și sunt distruse odată cu acesta.

Varianta de agregare în care subobiectele agregate aparțin în mod exclusiv agregatului din care fac parte, iar durata lor de existență coincide cu cea a agregatului este denumită și compoziție. Următoarea diagramă de clase UML ilustrează relația de compoziție între clasele *Sesiune* și *list*. Compoziția este reprezentată cu ajutorul unei linii continue care are un romb plin la capătul dinspre noua clasă obținută prin compoziție (la capătul dinspre clasa container - agregat).

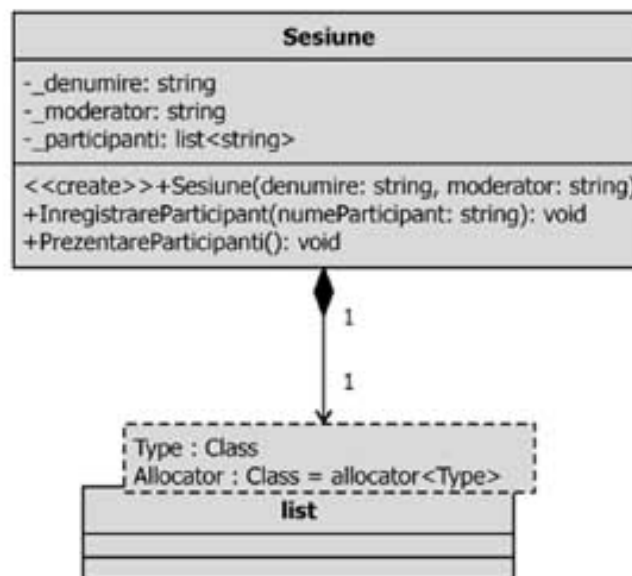


Figura 14. Clasa *Sesiune* este construită cu ajutorul compoziției pornind de la clasele *string* și *list*

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

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

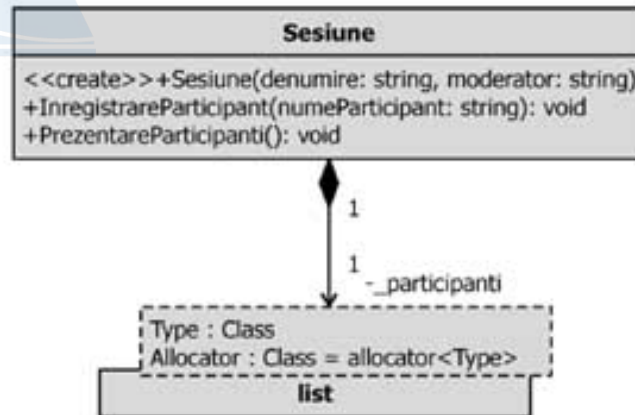


Figura 15. Data membru `_participanti` poate fi asociată în mod explicit cu relația de compoziție

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

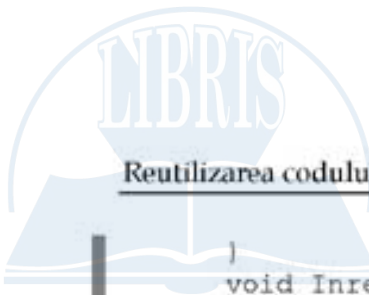
```

//Nume fisier: sesiune.h

#ifndef __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 PrezentareParticipantii() const;
private:
    string _denumire;
    string _moderator;
    /**
     * Fiecare sesiune posedă propria listă de
     * participanți.
     */
    list<string> _participanti;
};

#endif

//Nume fisier: sesiune.cpp

#include "sesiune.h"

void Sesiune::InregistrareParticipant(const string&
numeParticipant)
{
    // Aduga noul participant la sfarsitul listei.
    _participanti.push_back(numeParticipant);
}

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

    cout<<"Sesiunea: "<< _denumire<<endl;
    cout<<"Moderator: "<< _moderator<<endl;
    cout<<"Lista participanți"<<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* incorporând 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`.

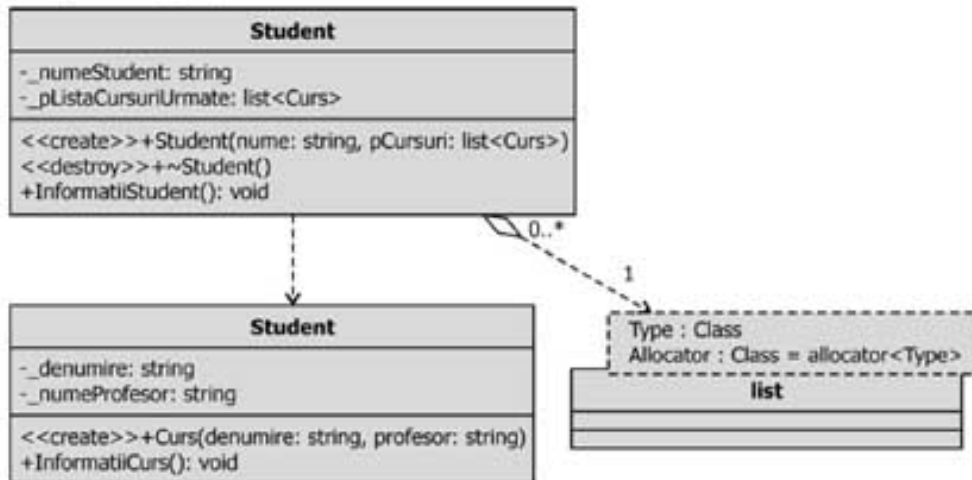


Figura 17. Între clasele `Student` și `list` există o relație de agregare

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:
```