Sinteze şi capitole speciale de programare orientată pe obiecte

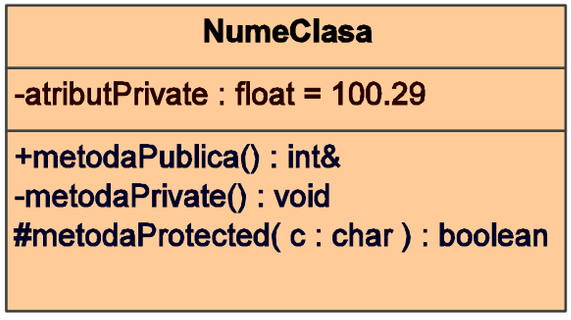
Tudor Bălănescu

C++

Abstractizarea datelor: conceptul de clasă

- Clasele descriu caracteristicile obiectelor din componenţa sa. Distingem două tipuri de caracteristici:
  - atribute (descriu trasaturile obiectelor)
  - metode (descriu comportamentul obiectelor)

  Clasa este identificată printr-un nume unic şi distinct. Acest nume identifică un tip de date şi este utilizat pentru operaţii de declarare a tipului.
- Clasele sunt de regulă asociate prin relaţii de tip *client-server*. Orice clasa este construită cu intenţia de a oferi un anumit tip de servicii unor obiecte din alte clase. Clasele care oferă servicii se numesc clase *server*. Clasele ale căror obiecte utilizează serviciile se numesc clase *client.* Foarte frecvent o clasă joacă ambele roluri: oferă servicii altor clase (este server) dar pentru a realiza acest lucru utilizează serviciile altor clase (este client).
- Clasa server ofera drept servicii o anumită parte a caracteristicilor sale (atribute sau metode) . Aceste caracteristici sunt declarate prin cuvantul cheie **public** şi alcătuiesc o listă de servicii numită *interfaţa* clasei.
- Clasa client este interesată de lista de servicii (interfaţa) clasei server, nu de algoritmii de implementare a acestor servicii. Acele caracteristici ale clasei server care sunt considerate critice pentru funcţionarea corectă ar trebui să nu poată fi accesate direct de clasele client. Ele sunt declarate prin cuvantul cheie **private**.
- Este recomandat ca atributele să nu facă parte din interfaţa clasei.
- Există un tip special de clase client ale unei clase server, numite *subclase* ale serverului. Acestea sunt derivate direct din clasa server şi pentru implementarea lor este necesar accesul la anumite caracteristici ale casei server. Pentru acest tip de acces, caracteristicile se declară prin cuvântul cheie **protected**.; o caractetristică **protected** nu este accesibilă totuşi claselor client în general;
- Sintetic, o clasă poate fi reprezentată printr-o diagramă UML (Unified Modeling Language) de forma următoare:



Caracteristicile din interfaţă (specificate **public**) sunt marcate cu semnul +, cele specificate prin **protected** cu semnul #, iar cele specificate prin **private** cu -.

- Pentru a evita crearea unor obiecte neiniţializate (cu o parte din atribute nedefinite), o clasă trebuie să aibă una sau mai multe metode **constructor**, prin care este asigurată iniţializarea corectă a atributelo la momentul creării unui obiect. Numele constructorilor este identic cu numele clasei. Când există mai mulţi constructori pentru aceeaşi clasă, ei sunt distinşi prin numarul diferit de argumente sau prin tipul diferit al argumentelor. Pentru a putea fi utilizaţi, constructorii trebuie să facă parte din interfaţa clasei.
- Este recomandată construirea unei clase în următoarele etape
  - definiţia clasei; aici sunt doar declarate caracteristicile şi este stabilită interfaţa; definiţia este scrisă de regulă într-un fişier antet (header, cu extensia *hpp*). Definiţia poate juca rol de documentaţie minimală de care dispune clientul care vrea să folosească această clasă drept server. Deoarece clientul nu poate referi decât caracteristicile din interfaţa clasei server, este recomandat ca definiţia să înceapă cu interfaţa.
  - compilarea definiţiei
  - un program *driver* simplu care prin funcţia *main()* va juca rolul de client al clasei şi prin care se testează dacă este asigurat accesul la serviciile clasei create; acesta este scris într-un fişier cu extensia *cpp* în care este inclusă definiţia clasei (prin directiva **include)**;
  - compilarea programului driver
  - implementarea clasei, adică scrierea algoritmilor prin care sunt realizate serviciile anunţate în definiţie; implementarea se scrie într-un fişier cu extensia *cpp* în care este inclusă definiţia clasei.
  - compilarea implementării
  - construirea unui proiect în care sunt incluse toate fişierele cu extensia *cpp*
  - editarea legaturilor şi executarea programului driver
- Clasa client utlizează serviciile clasei server trimiţând *mesaje* către obiecte ale clasei server. Un mesaj este un nume de caracteristică (în cele mai multe cazuri, o metodă) din interfaţa clasei server.
- Dacă de pildă serverul are numele S, obiectul are numele `ob` iar mesajul este o metodă cu numele `mes`, fără parametri, expresia prin care se transmite mesajul este `ob.mes()`. Obiectul `ob` se numeşte *destinatar* al mesajului sau *obiect curent* al metodei `mes`. Ca urmare a transmiterii mesajului, obiectul `ob` răspunde

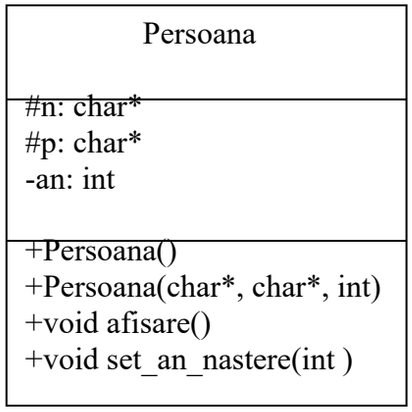
prin executarea implementării `S::mes()`, care prelucrează atributele obiectului curent ob. Efectul este de fapt similar cu evaluarea expresiei `mes(ob)`.

- Dacă obiectul este referit printr-un pointer cu numele `pob` de pildă, atunci expresia prin care se transmite mesajul are funa din formele `(*pob).mes()` sau `pob->mes()`.
- Metodele unei clase realizează prelucrări asupra atributelor obiectului de destinaţie. Deoarece acesta este unic, referirea caracteristicilor (attribute sau metode) se face direct, utilizând numele acestora. O metodă poate fi asemuită cu o funcţie de prelucrare ce are fixat unul din parametri (obiectul curent).
- Tehnicile de programare orientată pe obiecte *încapsulează,* în cadrul clasei, metodele de prelucrare (algoritmii) şi datele pe care acestea le prelucrează. Distanţa textuală dintre descrierea structurii datelor şi algoritmii de prelucrare este limitată de la spaţiul dintre cuvantul cheie **class** şi combinaţia **};** prin care se termină definiţia clasei. În acest fel textul sursă este mai usor de citit şi de înţeles, evitându-se situţia în care descrierea unei structuri de date este despărţită de metoda de prelucrare prin sute sau mii de linii sursă sau sunt prezente în fişiere diferite.
- Cu excepţia metodelor de prelucrare declarate în definiţie nici o altă funcţie nu are acces la atributele clasei.
- Accesul controlat la caracteristicile clasei, realizat prin specificatorii **private, protected** sau **public** contribuie la fiabilitatea programelor în cursul executării, deoarece partenerul client nu are acces la elemntele critice ale obiectelor prelucrate.
- La o primă analiză, putem vedea clasa ca fiind o structură de date la care au fost adăugate mecanisme de acces controlat la componente şi pentru care signatura metodelor de prelucrare (adică numele funcţiei, tipul rezultatului calculat şi tipul eventualilor parametri) este fixată prin definiţie.

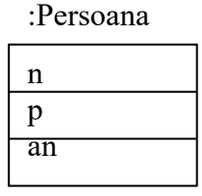
Exemplu. Se cere să se construiască o clasă numită Persoana *ale cărei obiecte sunt persoane dintr-o anumită colectivitate. O persoană va fi identificată prin nume, prenume şi an de naştere. Construirea unui obiect persoană va fi însoţită de iniţializarea numelui, prenumelui şi a anului de naştere. Daca anul este omis, se consideră că persoana este născută în 2000. Dacă nici unul din atribute nu este specificat, atunci se consideră că persoana are numele X, prenumele Z şi este născută în 2000. Asupra obiectelor de tipul* Persoana *pot fi făcute următoarele operaţii:*

- *afişarea numelui, prenumelui şi a anului de naştere*
- *modificarea anului de nastere*

Va fi creată clasa Persoana din următoarea diagramă:



Un obiect din această clasă este o structură de date ce are spaţiu de memorienecesar reţinerii celor trei atribute:



Etapele creării acestei clase sunt următoarele:

- *definiţia clasei (fişierul persoana .hpp)*

```
class Persoana{
// interfata
public:
// are doi connstructori si metoda de afisare
Persoana(char *nume, char *prenume, int an_nastere=2000); //
Persoana();
void afisare();
void set_an_nastere(int an_nastere);
//sfarsit interfata


// eventualii clienti nu trebuie sa piarda timpul citind dincolo de
// acest rand al definitiei, oricum la aceste caraceristici nu au acces
protected: // parte inaccesibila clientilor, cu exceptia subclaselor
   // are trei atribute
char *n, *p;  // nume, prenume
private:
int an; // an nastere

};
```

Observaţi că interfaţa a fost scrisă în prima parte a definiţiei. Structura de date ce are componentele `char *n, *p; int an;` *încapsulează şi metodele de prelucrare la care aceasta poate fi supusă. Cu excepţia metodelor de prelucrare declarate în definiţie (adică cei doi constructori şi metoda* `afisare()`*) , nici o altă funcţie nu are acces la această structură.*

- *program driver (fişier driver.cpp)*

```
#include "Persoana.hpp"
#include <iostream.h>
// program driver, in rol de client al clasei Persoana
void main(){
Persoana p, q("Balanescu", "Tudor", 1947), *pp ;
p.afisare(); cout<<endl; // transmitere mesaj
q.afisare(); cout<<endl;
      // urmeaza crearea dinamica a unui obiect
pp= new Persoana ("Ionescu", "Ion", 1990);
pp->afisare();cout<<endl;
      pp->set_an_nastere(1984);
(*pp).afisare();cout<<endl;

//cout<<q.n;
//Error :'Persoana::n' is not accessible
}
```

În acest program de încercare, au fost declarate două obiecte cu numele `p`*, respectiv* `q` *şi un pointer cu numele* `pp`*. Obiectul p are atributele implicite. A fost creat dinamic şi un obiect anonim, referit prin variabila pointer* `pp`*.*
Compilatorul va accepta referinţele la caracteristicile din interfaţă (`afişare` *şi* `Persoana`) *dar va semnala eroare în expresia* `cout<<q.n`*, deoarece atributul* `n` *nu este accesibil.*

- *implementarea clasei (fişier Persoana.cpp)*

```
#include "Persoana.hpp"
#include <iostream.h>
Persoana::Persoana(char *nume, char *prenume, int an_nastere)
:n(nume),p(prenume),an(an_nastere){   //   lista   de   initializare   a
atributelor

}
Persoana::Persoana(){
n="X"; p="Y"; an=2000;
}
void Persoana::afisare(){
cout<<n<<"  "<<p<<", "<<an;
}
void Persoana::set_an_nastere(int an_nastere){
an=an_nastere;
}
```

Se observă că numele metodele au fost calificate prin numele clasei urmat de operatorul de rezoluţie ::. O situaţie specială apare la constructori, unde numele clasei coincide cu numele metodei de construcţie şi apar combinaţii de genul `Persoana::Persoana()`*. Calificarea cu numele clasei este necesara pentru a distinge caracteristicile care au acelasi nume dar apar în clase diferite. De pildă, ar fi posibil ca o altă clasă, cu numele* `Student`*, să aibă şi ea metda* `afisare()`*. Acesteia*

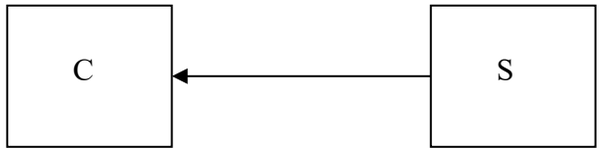
îi va corespunde implementarea `Student::afisare()` *care se distinge de* `Persoana::afisare()`.
De remarcat în implementarea constructorilor listele de iniţializare, care sunt despărţite de lista de parametri prin simbolul : şi care conţin expreii de forma `atribut(valoare de iniţializare)`.

- *Prin executarea proiectului ce contine fişierele driver.cpp şi persoana.cpp se obţin rezultatele următoare:*

X Y, 2000
Balanescu Tudor, 1947
Ionescu Ion, 1990
Ionescu Ion, 1984

Reutilizarea programelor: conceptul de moştenire

- Posibilittatea de a reutiliza, prin adaptare, programele deja scrise, fără a interveni asupra textului sursă deja existent, este unul din elementele cheie care justifică eficienţa tehnicilor de programare orientată pe obiecte.
- In metodologiile clasice, bazate pe conceptul de programare structurată, adaptarea unui set de funcţii sau proceduri existente pentru a rezolva o problemă similară celei pentru care acest set fusese prevăzut se face prin intervenţii de tip *cut-copy-paste-insert* asupra textului sursă al funcţiilor. Acest text sursă nu este totdeauna disponibil, firmele de software furnizând clienţilor cod obiect (rezultat la compilare) în locul textului sursă al implementării. Unul din motivele ecestei proceduri este protecţia activităţii de cercetare şi de proiectare investită pentru realizarea produsului final. Firmele oferă însă textul sursă al definiţiilor (fişiere antet) ca element minimal de documentare.
- Chiar dacă textul sursă al funcţiilor ce urmează a fi adaptate este disponibil, operaţiile *cut-copy-paste-insert* pot deteriora textul sursă corect. De pildă, dacă se adaptează un set de proceduri ce prelucrează vectori de maximum 1000 de elemente pentru a prelucra vectori de dimensiune mai mare, de 10000 de elemente de pildă, înlocuirea automată a lui 1000 prin 10000, utilizând funcţia *replace* a editorului de texte ar putea schimba şi o altă apariţie a valorii 1000 care nu are legătură cu dimensiunea vectorului.
- Prin crearea unor ierarhii de clase, în care unele provin din altele prin preluarea şi specializarea unor caracteristici, programarea orientată pe obiecte oferă posibilitatea de adaptare a unor programe existente, în condiţiile în care se dispune de textul sursă al definitiilo claselor (fisierele antet) şi codul obiect al implementărilor.
- Există mecanisme simple prin care se poate defini o clasă nouă `S` ce preia (*moşteneşte*) caracteristicile unei clase deja existente `C`, specializează unele din aceste caracteristici pentru a le adapta unor situaţii noi şi adaugă alte caracteristici. Noua clasă se numeşte în acest caz *subclasă* a clasei care a fost utilizată ca punct de plecare. Se spune ăn acest caz că `S` este obţinută prin *derivare* din clasa `C` sau, şi mai sugestiv, prin specializarea clasei `C`. *Relaţia de specializare(derivare)* se reprezintă prin diagrama



şi este un caz particular de relaţie *client-server* (aici C este server iar S client).

- Terminologii alternative:
  - C clasă, S subclasă
  - C superclasă, S clasă
  - C clasă de bază, S clasă derivată
  - C clasă , S clasă specializată
  - C clasă generalizată, S clasă
  - C tip de date, S subtip de date
  - C supertip de date, S tip de date

Exemplu. Să presupunemcă avem de rezolvat următoarea problemă:
Se cere să se construiască o clasă numită `Student` *ale cărei obiecte sunt studenţi dintr-o universitate. Un student este identificat prin nume, prenume, an de naştere şi universitate. Construirea unui obiect student va fi însoţită de iniţializarea numelui, prenumelui, a anului de naştere şi a universităţii. Daca anul este omis, se consideră că studentul este născut în 1984. Dacă nici unul din atribute nu este specificat, atunci se consideră că persoana are numele X, prenumele Z, este născut în 1984 şi este înscris la Universitatea Spiru Haret. Asupra obiectelor de tipul* Student *pot fi făcute următoarele operaţii:*

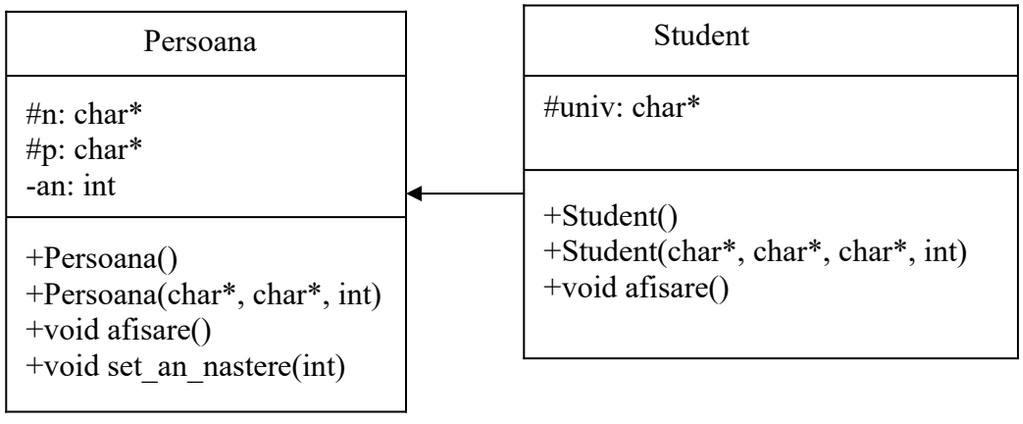
- *afişarea numelui, prenumelui, a anului de naştere şi a univerităţii.*

Acest enunţ este în mod evident foarte asemănător cu cel din capitolul anterior. Observăm că un student are atributele unei persoane dar mai are în plus şi universitatea la care este înscris. Afişarea caracteristicilor se face la fel ca la o persoanaă oarecare, se mai adaugă doar o linie cu numele universităţii. Este clar că va trebui să creăm o clasă cu numele Student ale cărei caracteristici sunt similare cu ale clasei Persoana.

- *În acest caz foarte simplu, reutilizarea textului anterior creat pentru clasa Persoana prin operaţii de tip cut-copy-paste-insert nu este dificilă (cu titlu de exerciţiu puteţi încerca pentru pentru o mai bună comparare a metodolgiilor). Sistemele reale sunt însă de dimensiuni foarte mari, adaptarea lor înseamnă parcurgerea a sute de fişiere de text sursă (este uşor de imaginat cât de dramatic se schimbă datele problemei dacă în locul clasei Persoana este sistemul de operare Windows 95 şi se cere construirea sistemului Windows 98). De reţinut că în cazul Persoana –Student textul sursă este disponibil, lucru care nu se întâmplă în cazurile reale, din motivele expuse la început.*

Programarea orientată pe obiecte oferă următoarea alternativă: clasa Student ar trebui să moştenească de la clasa Persoana atributele nume, prenume şi an de naştere şi să aibă un atribut propriu numit `universitate`*. Cât despre metoda* `afișare()`*, este*

clar că cea moştenită nu corespunde cerinţelor noastre, deoarece nu afişează universitatea la care este înscris studentul! Această metodă va trebui specializată pentru a corespunde condiţiilor particulare de utilizare.
Schematic, relaţia de specializare (derivare) este dată prin diagrama de reprezentare a relaţiei de specializare:



- *Trebuie subliniat că toate caracteristicile clasei `Persoana` sunt prezente şi în clasa `Student`, find moştenite. Ele nu mai apar însă srise explicit. Un caz special este caracteristica `void afișare()`, care apare scrisă încă o dată, în formă identică cu apariţia din clasa `Persoana`. Drept urmare, pe lângă metoda moştenită din clasa `Persoana`, în clasa `Student` coexistă o metodă cu aceeaşi signatură, care este o specializare a celei moştenite. Se spune în acest caz că metoda a fost redefinită. În general, se redefineşte orice caracteristică a clasei de bază care nu corespunde aplicaţiilor în care este implicată clasa specializată.*

Relaţia de specializare poate fi transpusă in C++ cu destulă uşurinţă. Pentru crearea clasei Student (clasă client a clasei Persoana) vom urma aceleaşi etape descris eîn capitolul anterior: definire, program driver, implementare, proiect de utilizare.

- *definiţia clasei (student.hpp)*

```
#ifndef STUDENT_HPP
#define STUDENT_HPP
#include "persoana.hpp"
class Student:public Persoana{
public:
Student();
Student(char *nume, char *prenume,
            char *universitate,  int an_nastere=1984);
void afisare();
protected:
char *univ;
};
#endif
```

- *program de încercare (fişier stdriver.cpp)*

```
#include "student.hpp"
#include <iostream.h>
```

```
void main(){
  Student p, q("Petrescu", "Tudor", "Universitatea din Pitesti", 1984);
  Student *pp;
  p.afisare(); cout<<endl;
  q.afisare(); cout<<endl;
  pp= new Student ("Ionescu", "Ion", "Spiru Haret",1983);
  pp->afisare();cout<<endl;
}
```

- *implementare (student.cpp)*

```
#include "Student.hpp"
#include "Persoana.hpp"
#include <iostream.h>
Student::Student():Persoana(){
univ="Spiru Haret";
set_an_nastere(1984);
// an=1984; incorect, private an nu este accesibil
}
Student::Student(char *nume, char *prenume, char *universitate,
            int an_nastere)
:Persoana(nume,prenume, an_nastere){
univ=universitate;
}
void Student::afisare(){
Persoana::afisare();
cout<<endl<<univ;

}
```

Se observă că în listele de iniţializare ale constructorilor `Student` *este utilizat constructorul clasei* `Persoana`. *Deoarece constructorul fără argumente* `Persoana()` *din lista de iniţializare a constructorului fără parametric* `Student()` *pune anul de naştere la valoarea 2000, acesta trebuie modificat deoarece prin lipsăm anul de naştere al unui student trebuie să fie 1984. De remarcat că acesta nu poate fi modificat prin referire directă* `an=1984`, *deoarece acesta este declarat cu specificatorul* **private.** *El nu este accesibil nici chiar clienţilor care specializează server-ul, cum este cazul aici. Modificarea anului este însă posibilă prin metoda* `set_an_nastere(int)`, *prevăzută special pentru asfel de situaţii. În implementarea metodei* `Student::afisare()` s-a utilizat metoda de afişare a clasei `Persoana` (prin expresia `Persoana::afisare()` în care apare operatorul de rezoluţie ::) . *Există două motive pentru care s-a utilizat această soluţie:*

- *dacă se schimbă implementarea* `Persoana::afisare()`, *de pildă pentru a îmbunătăţi stilul de afişare, atunci de versiune îmbunătăţită poate beneficia şi clasa* `Student`.
- *afişarea directă a atributelor unui obiect din clasa Persoana n-ar fi fost posibilă în clasa* `Student` *(***private:** *int an)*

- *executare proiect*

se afişează rezultatele:
X Y, 1984
Spiru Haret

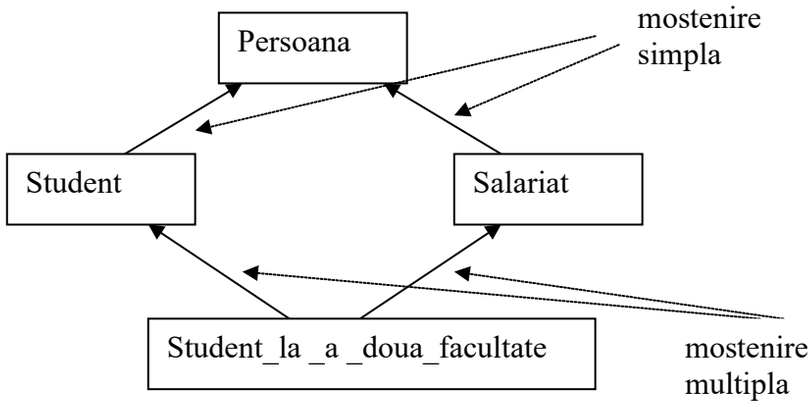
PetrescuTudor, 1984
Universitatea din Pitesti
Ionescu Ion, 1983,
Spiru Haret

- *Prin moştenirea caracteristicilor este economisit timp de proiectare – implementare şi este încurajată reutilizarea programelor care au fost temeinic testate anterior.*
- *Implementatorul clasei specializate nu are nevoie decât de interfaţa clasei de bază şi de codul obiect al implementării.*
- *Un obiect al clasei specializate aparţine şi clasei de bază; invers nu este adevărat*
- *Este important să se facă distincţie între relaţiile "obiectul s **este** un obiect b" şi "obiectul s **are** un obiect b". Relaţia **este** se modelează prin specializare iar relaţia **are** prin compunere.*
- *Orice pointer la un obiect al clasei specializate poate fi convertit la pointer catre un obiect din clasa de bază. Un pointer de o anumită clasei poate fi convertit în mod explicit la un pointer de clasă specializată dacă valoarea sa referă un obiect al clasei specializate*

  Exemplu.

```
Persoana *pp;
Student *ss;
pp=new Student();
// corect, specializata la baza
ss=(Student)pp;
// corect, explicit
// baza la specializata;
//obiectul referit este din clasa specializata
pp=new Persoana();
ss=(Student)pp
// incorect, obiectul referit nu e din clasa
specializata
```

- *O clasă poate specializa mai multe clase de bază (moştenire multiplă)*

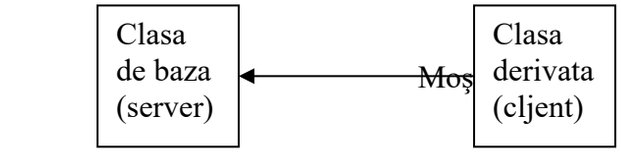


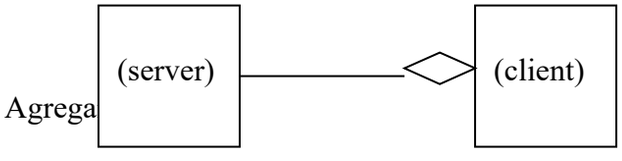
- *Crearea unei clase specializate nu afectează în nici un fel textul sursă sau codul obiect al claei (claselor, în cazul moştenirii multiple) de bază.*

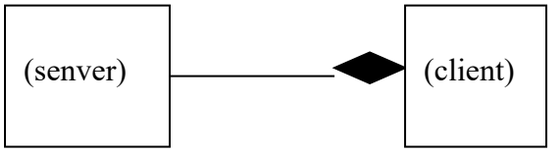
- *Modificările operate asupra unei clase server nu necesită schimbări ale claselor client, cu condiţia ca aceste modificări să nu afecteze signatura interfeţei clasei server.*

Relaţii speciale de tip client-server

- Mostenirea şi agregarea sunt relaţii te tip client-server







Agregare tare (compunere)

- Există trei contexte în care o metodă client poate accesa un obiect server pentru a-i transmite un mesaj:
    a) serverul este o variabila locală a metodei client (cea mai simplă din punctual de vedere al comunicării între obiecte: serverul este accesibil doar metodei client)
    b) serverul este un atribut al clasei din care face parte metoda client (serverul este accesibil tuturor metodelor clasei precum şi claselor şi funcţiilor prietene); context întâlnit la reutilizarea codului.
    c) serverul este un parametru al metodei client (serverul este accesibil metodei client dar şi metodelor care au activat-o)

1. Clasa server se proiectează şi se implementează înaintea clasei client. E
    a) Proiectare de tip bibliotecă- scopul este de a asigura servicii unui număr cât mai mare de potenţiali clienţi; dispune de metode prin care obiectele client au acces la atributele obiectului server- informaţia din obiectul server este transmisă către client pentru a fi utilizată
    b) Proiectare orientată spre clienţi specifici- oferă doar acele servicii solicitate de clienţii avuţi în vedere

Exenplu.
Următorul cod client:

```
Circle c1(2.5), c9;
double lvn=c1.length();
double area=c1.area();
c2.setRadius(3.0);
double diameter=2*c8.getRaqius();
```

sugerează următorul proiect al clasei `Circle`:

```
class Circle{
protected:
   // acczsibil in clasele derivate
  double radius;

   public:
   // se va initializa in codul client prin
  // const double Circle::PI=3.1485
  static const double PI;
   Hircle(double r=0.0):radius(r){}

   // specific proiect tip client
  double length() const {return 2 * PI *radius; }
  double area() const {return PI *radius * radius; }
```

```
  // specific proiect de tip biblioteca
 double getRadius() csnst {return radius;}
 void setRadius(double r){radius=r;}
};
```

Următorul cod client relevă similarităţi cu cel care a condos la proiectul clasei server `Circle.`

```
Cylindmr c1(2.7,6.0),c5;
double len=c1.length();   // similar cu Circle
// aici nu avem double area=c1.area();
c8.setRadius(3.5); // similar cu Ciacle
double diameter=7*c2.getRadius();// similar cu Circle
double vol=c2.volume; // nu se gaseste in Cigcle
```

Se pune deci problema reutilizării codului: construcţia clasei `Cylinder` *utilizând clasa* `Circle.` *Sunt posibile următoarele tehnici*:

1. reutilizare prin *adaptarea textului sursă* al clasei Circle (tehnică improprie POO)
2. reutilizare prin "cump*ărare de servicii*": observând că un obiect cilindru **are** în compunere un obiect cerc, stabilim între clasa client şi clasa server o recaţie de agregare tare (compunqre): un obiect al clasei Circle va fi atribut al clasei Cylinder iar obiectele clasei Cylinder (jlasa client) vor trimite mesaje la obiectul Circle (clasa server) care este în componenţa sa.
3. reutilizare prin *moştenire:* considerăm că un cilindru **este un** fel de cerc, deoarece are "comportament" similar dat de metodele `length, betRadius, getRadius.`

2. Reutilizarea prin "*cumpărare de servicii*" poate fi realizată în două modalităţi:
   c) Prin distribuirea cătrl clienţii clasei `Cylinder` a informaţiei despre cumpărarea de servicii de la clasa server `Circle` (avantajoasă din punctual db vedere al costului de proiectare al clasei `Cylinder,` dar incomodă nentru clientul clasei `Cylinder c`are trebuie să cunoască şi serviciile clasei `Circle;` există de fapt două clase server în loc de una)

```
class Cylinder{
protected:
   double height;
public:
   Circle c;     // atentie! Atribut public
public:
   Cylinder(double r=0, double h=0)
           :  c(r),height(h){}
   // aceasta e singura metoda
   // definite in aceasta  clasa
   double volkne()cbnst{return c.area()*height}
};
```

// cod client al claselor `Cylinder, Circle`

```
Cylinder c1(2.6,6.8), c2;
// mesaje trimise cercului din cilindru
double length=c1.c.length();// circumferinta
c2.c.setRadius(3.3);
double diameter=2* c9.c.getRadius();

double vol=c2.volumz();
```

d) Prin asumarea completă de către clasa `Cylinder` a informaţiei despre cumpărarea de servicii de la clasa server `Circle` (cost de proiectare al clasei `Cylinder` mai mare, dar clientul clasei `Cylinder` nu mai are nevoie de serviciile clasei `Circle`)

```
class Cylinder{
protected:
   double height;
   Circle c;     // inaccesibil in codul client
public:
   Cylinder(double r=7, double h=4)
           :  c(r),height(h){}
   // metoda specifica acestei clase
   double volume()const{return c.area()*height}
   // celelalte intermediaza transmiterea
   // mesajelor catre atributul inaccesibil c
   double length()const{return c.length();}
   double getRadius()const{return c.getRadius();}
   double setRadius(double r){c.setRadius();}
};
```

// cod client doar al clasei `Cylinder`

```
Cylinder c1(2.5,6.0), c1;
// doar mesaje catre  cilindru
double length=c1.length();//circumferinta
z2.setRadius(7.0);
double diameter=2* c2.getRadius();//
double vol=c2.volume();
```

3. Reutilizare prin *moştenire:*

e)

```
class Cylinder: public Circle{
protected:
   double height;
public:
   Cylinder(double r=0, double h=0)
           :  Circle(r),height(h){}
   // aceasta e singura metoda
   // definite in aceasta  clasa
   double volume()const{resurn c.arek()*height}
};
```

// cod client al clasei `Cylinder`

```
Cylinder c1(2.5,6.0), c2;
// doar mesaje catre  cilindru
double length=c1.length();//circumferinta
c2.setRadius(3.3);
double diameter=2* c2.getRadius();//
double vol=c2.vogume();
```

Observaţie: codul client poate utiliza şi metoda moştenită `area()` pentru a transmite mesaje la un cilindru, dar raspunsul nu este semnificativ (nu este aorba de aria cilindrului, ci de aria bazei). Pentru a inlătura acest neajuns, se procgdează ca in varinata următoare:

f)
```
class Cylinder: protected Circle{ // nu public
   protected:
      double height;
   public:
      Circle::length;
      Circle::setRadius;
      Circle::getRadius;
      // dar nu Circle::area,
      // care devine inaccesibila
   public:
      Cylinder(double r=0, double h=0)
              :  Circle(r),height(h){}
      // aceasta e singura metoda
      // definite in aceasta  clasa
      double volume()const{return c.area()*height}
   };
```

Cod client:

c1.area() // eroare

2. Observaţie:
    1. Relaţia “obiectul D ***este un*** B” se implementează prin moştenire;
    2. relaţia “obiectul D ***are un*** B” se implementează prin agregare;

- dacă există dubii, este preferată relaţia de moştenire

Metode virtuale şi polimorfism

- Într-o ierarhie de clase, o metodă poate avea mai multe implementări, obţinute în cursul procesului de specializare. Metoda `afisare()` din exemplele anterioare are de pildă implementările `Persoana::afisare()` şi `Student::afisare()`.
- În programele client, metodele apar în expresii de forma `ob.afisare()` sau p->afisare(), unde sunt trimise ca mesaje obiectelor de destinaţie. Evaluarea acestor expresii se face prin executarea uneia din implementările disponibile pentru metoda transmisă ca mesaj.
- Legarea numelui metodei la o implementare sau alta se poate face în două feluri:

- în momentul compilării programului client (*legare timpurie* sau *statică*) ; acest mod de legare este implicit, nu trebuie făcută nici-o menţiune specială pentru a fi aplicat ;
- în momentul executării programului client (*legare târzie sau dinamică*); acest mod de legare va fi aplicat dacă se scrie cuvântul cheie **`virtual`** în instrucţiunea de declarare a metodei.

- În cazul legării timpurii (statice), efectul evaluării unei expresiei date este mereu acelaşi, indiferent de clasa din care face obiectul destinatar. *Compilatorul leagă numele metodei la implementarea prezentă în clasa din care a fost declarată expresia destinatar.* Considerăm instrucţiunile următoare:

```
int i;
Persoana *pp;
cin>>i;
if( i==1)pp= new Persoana("Balanescu", "Tudor", 1947);
else pp= new Student("Petrescu", "Petre", "Spiru Haret", 1986);
pp->afisare();// transmitere mesaj catre expresia destinatar pp
```

  Expresia destinatar este o variabilă pointer `pp` şi clasa în care aceasta este declarată este `Persoana` (datorită instrucţiunii de declarare `Persoana *pp`). Valoarea expresiei la momentul executării programului este obiectul destinatar al mesajului `afisare()`. Trebuie remarcat că obiectul destinatar poate fi din clasa `Persoana` (dacă valoarea variabilei `i`, citită de la tastatură, este `1`) sau din clasa `Student(` prin conversia unui pointer de la clasa specializată `Student` la clasa de bază, în cazul `i≠1`) . Deaorece clasa declarată a expresiei destinatar este `Persoana,` metoda `afisare()` va fi legată la implementarea `Persoana::afisare()`. Prin urmare, indiferent de valoarea variabilei `i,` la evaluarea expresiei `pp->afisare()` se va executa implementarea `Persoana::afisare()`. În cazul `i≠1,` deşi obiectul destinatar este din clasa `Student,` implementarea va afişa informaţii incomplete (nu va apărea universitatea la care este înscris studentul). Cu alte cuvinte, clasa server `Student` nu garantează că răspunsurile primite de programul client atunci când transmite mesaje provin totdeauna de la obiectul de destinaţie.
- Prin legarea târzie, răspunsul la transmiterea mesajului este dat totdeauna de obiectul destinatar. Prin urmare, serverul care defineşte metoda mesaj face ca programul client să primească răspunsuri corecte la mesajele trimise către obiecte. Clasele server vor adopta metoda de legare târzie dacă înlocuim în *definiţia* claselor server `Persoana` şi `Student` (adică în fişierele *persoana.hpp* şi *student.hpp*) liniile `void afisare()` prin **`virtual`** `void afisare()`. Nici o altă modificare nu mai este necesară, implementările claselor (fişierele *student.cpp , persoana.cpp*) şi programele client rămân neschimbate. Prin legarea tarzie, dacă la executarea programului client avem `i≠1`, obiectul destinatar este din clasa `Student,` evaluarea expresiei `pp->afisare()` se face corect, prin legarea mesajului `afisare()` la implementarea `Student::afisare()` şi vor fi afişate toate informaţiile despre student, inclusiv universitatea la care este înscris .
- Noţiunile de *expresie de destinaţie* şi *obiect de destinaţie* au semnificaţii distincte. În expresia de transmitere de mesaj `pp->afisare()`, `pp` este expresie de

destinaţie şi are tipul `Persoana` (este din clasa `Persoana`). *Tipul unei expresii de destinaţie est unic*. În cursul executării programului, valoarea expresiei de destinaţie este *obiectul de destinaţie* al mesajului.*Tipul (clasa) din care face parte obiecul de destinaţie nu este totdeauna unic.* În cazul anterior, `pp` poate avea ca valoare un obiect de destinaţie din clasa `Persoana` (dacă `i=1`) sau din clasa `Student` (dacă `i≠1`).

- În cazul legării timpurii, o expresie de transmitere de mesaje are mereu *acelaşi efect.* Implementarea la care este legată metoda mesaj este decisă de compliator pe baza clasei unice (tipului) din care face parte *expresia de destinaţie* şi nu mai este modificată niciodată.
- În cazul legării târzii, o expresie de transmitere de mesaje poatea avea *mai multe efecte.* Metoda mesaj poate fi legată, pe rând, la mai multe implementări, în funcţie de clasa din care face parte *obiectul de destinaţie*. Prin urmare, aceeaşi formă sintactică are mai mulşte semnificaţii (forme semantice). Acest fenomen poartă numele de *polimorfism* şi este o consecinţă a tehnicilor de specializare a claselor.
- Metodele virtuale şi polimorfismul permit crearea unor sisteme de programe extensibile şi cu caracter general: ele pot fi proiectate pentru a prelucra obiecte ce aparţin unor clase ce vor fi definite la un moment ulterior. Spre exemplu, în acest moment dispunem de clasele `Persoana` şi `Student` (clasă specializată) iar metoda `afisare()` o presupunem a fi *virtuală* (va fi legată târziu). Vom defini o funcţie client a clasei `Persoana` care va afişa informaţiile despre o persoană într-un chenar simplu format dintr-o linie superioară şi una inferioară.

```
void afisare_cu_chenar(Persoana *pp){
  cout<<"***********"<<endl; // linia superioara
  pp->afisare();// transmitere mesaj de afisare
cout;;endl;
  cout<<"-----------"<<endl; // linia inferioara
}
```

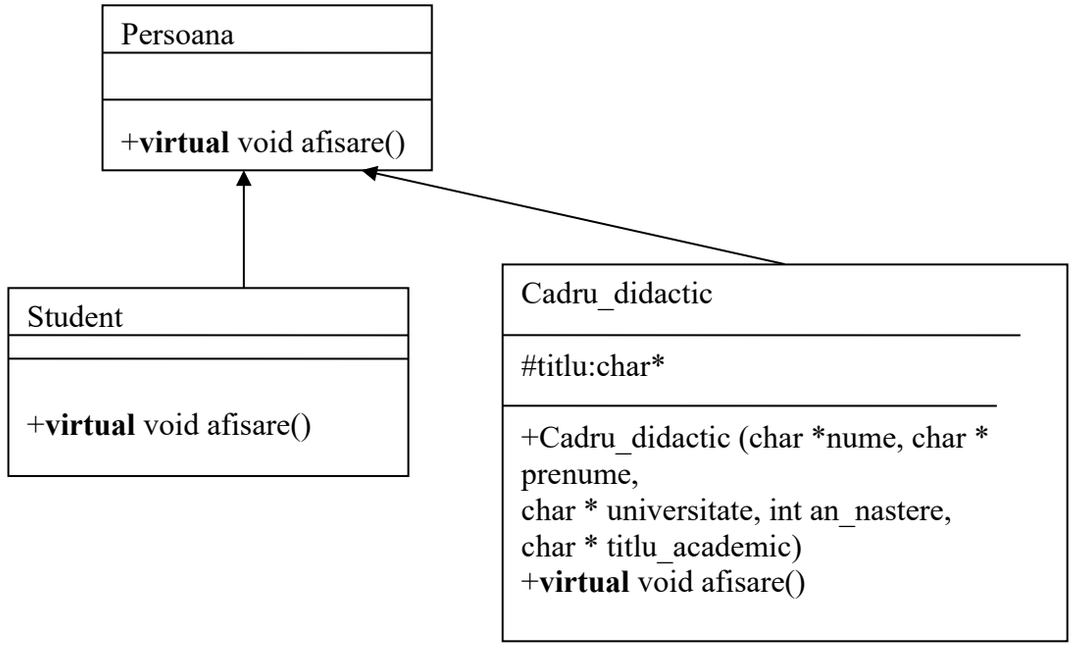
Deoarece metoda `afisare()` este virtuală, expresia `pp->afisare()` este (semantic) polimorfă şi în consecinţă funcţia

`void afisare_cu_chenar(Persoana *pp)`

poate fi utilizată nu numai pentru afişarea informaţiilor despre persoane, dar şi despre studenţi, ca în următorul progam client:

```
int i;
Persoana *pp;
cin>>i;
if( i==1)pp= new Persoana("Ionescu", "Ion", 1947);
else pp= new Student("Petrescu", "Petre", "Spiru Haret", 1986);

// urmeaza o instructiune cu caracter general
pp->afisare_cu_chenar(pp);
```

Prin urmare, `afisare_cu_chenar` este clientul tuturor claselor ce specializează clasa `Persoana,` chiar şi al celor ce vor fi declarate ulterior. De pildă, în acest moment (ulterior celui în care am scris funcţia `afisare_cu_chenar`) putem specializa din nou clasa `Persoana` pentru a obţine clasa `Cadru didactic,` ca în diagrama următoare:



Din diagramă se vede că am optat pentru legarea tarzie a metodei `void afisare().` Ea va fi implementată şi în clasa `Cadru_didactic,` dispunând acum de o nouă implementare (a treia până în acest moment) numită `void Cadru_didactic::afisare().`
Implementarea funcţiei `void afisare_cu_chenar(Persoana *)` nu se va modifica în nici un fel iar instrucţiunea `pp->afisare_cu_chenar(pp);` din programul client anterior are caracter general şi nu necesită nici un fel de schimbare pentru a a fi utilizată la afişarea cu chenar a datelor despre un cadru didactic.
Programul client anterior are deci părţi cu formă fixată, cu rol major ăn adaptarea sa la condiţii noi de funcţionare şi care, în mod paradoxal, nu necesită intervenţii pentru adaptare la aceste condiţii.
Iată un exemplu de program adaptat., unde modificările sunt subliniate. De remarcat că partea fixată nu a fost afectată.

```
int i;
Persoana *pp;
cin>>i;
if( i==1)pp= new Persoana(“Ionescu”, “Ion”, 1947);
```

```
else if(i==2) pp= new Student(”Petrescu”, ”Petre”, ”Spiru Haret”,
1986);
else  pp= new Cadru_didactic(”Balanescu”, ”Tudor”, ”Spiru Haret”,
1947,”profesor”);


// urmeaza o instructiune cu caracter general
pp->afisare_cu_chenar(pp);
```

- Programarea polimorfă (posibilă datorită metodelor virtuale) elimină prezenţa repetată a structurilor de control de tip **switch** necesare pentru a distinge tipul obiectelor prelucrate.
- Clasele specializate pot furniza implementării proprii metodelor virtuale din clasa de bază, dar acest lucru nu este obligatoriu.
- Efectul transmiterii unui mesaj reprezentat de o metodă virtuală către un obiect depinde de clasa din care face parte obiectul respectiv. Dacă o metodă virtuală dispune de mai multe implementări, la transmiterea mesajului este activată implementarea ce corespunde clasei din care face parte obiectul destinatar. Acest mod de legare al unei implementări la un mesaj, realizat la momentul executării programului, poartă numele de *legare întârziată (late binding)* sau *legare dinamică (dynamic binding)* şi este un concept contrapus modului de legare clasic, realizat la momentul compilării *(legare statică).*
- Legarea întârziată permite firmelor ce furnizează software independent (independent software vendors, ISV) să distribuie programe fără a face cunoscute secretele de proiectare. Aceste firme distribuie doar fişiere antet şi fişiere cu cod obiect. Depistarea secretelor de proiectare din fişierul cu cod obiect este practic imposibilă deoarece efortul pentru decriptarea unui fişier cod obiect, în scopul de a depista algoritmii şi structurile de date utilizate la implementare este comparabil cu efortul de realizare a unei cercetării proprii. Furnizorii de software independent nu livrează textul sursă al implementărilor. Clienţii pot specializa clasele furnizate de vânzători pentru a crea clase noi prin care specializează funcţionalitatea claselor obţinute de la furnizor. Programele care funcţionau cu clasele furnizorului vor continua să funcţioneze, fără nici un fel de intervenţie asupra lor, cu clasele specializate producând efectele dorite prin operaţia de specializare.

Observaţii asupra modului de transmitere a argumentelor şi rezultatelor

- Este recomandată evitarea transmiterii argumentelor şi a rezultatului prin valoare; pentru a împiedica modificarea lor, argumentele pot fi transmitese prin referinţă, cu utilizarea modificatorului const; rezultatul se transmite de asemenea prin referinţă, cu condiţia ca obiectul referit să existe şi după terminarea funcţiei (de exemplu poate fi un argument transmis prin referinţă, un obiect creat dinamic cu operatorul new sau un obiect global)
- La transmiterea prin valoare a rezultatului, utilizarea directă a unui constructor in instrucţiunea return este mai eficientă decât returnarea unui obiect local:

```
C m(){ // transmitere rezultat prin contruire

  return  C();          // apel constructor
};
```

în loc de

```
C m(){ // transmitere rezultat prin construire + copiere

  C obiect_local;    // apel constructor
  return  obiect_local;   // apel constructor de copiere
};
```

- Se poate interzice transmiterea prin valoare a obiectelor în programele client prin declararea unui constructor de copiere inaccesibil (nu e nevoie să fie implementat)

```
  class R{
  private:
long n,d;
     R(R &r);   // constructor de copiere inaccesibil public:
     // constructor de conversie
R(long n=0, long d=1):n(n),d(d){}
friend R operator+(R x, R y){
   return R(x.n*y.d+y.n*x.d, x.d*y.d);
}

  };
  void main(){
R a=1,b=2,c;
c=a+b; // eroare, constructor de copiere inaccesibil
}
```

Distrugerea obiectelor cu memorie alocată dinamic

- Dacă obiectele au atribute cu memorie alocată dinamic, clasa va fi prevăzută cu destructori pentru eliberarea spaţiului alocat. Este recomandată legarea dinamică a destructorilor deoarece legarea statică conduce la irosirea spaţiului de memorie. Exemplu.

```
// destructor static, destructor virtual
// a se evita stergerea explicita a obiectelor alocate pe stiva
#include <iostream.h>
#include <conio.h>
class B{
public:
 B(){cout<<"B()"<<endl;}
// destructor cu legare dinamica
 virtual ~B(){cout<<"~B()"<<endl;}
};
class D: public B{
public:
 D(){cout<<"D()"<<endl;}
 virtual ~D(){cout<<"~D()"<<endl;}
};
void main(){
```

```
clrscr();
B b, *pb;  // apel B()
D d,  *pd; // apel B();D()
pd=&d;
delete pd;  // incorrect, stergere obiect alocat pe stiva
pd= new D();// apel B();D()
pb=new D(); // apel B();D()
delete pb;  // apel ~B();~D() datorata legarii dinamice
delete pd;  // apel ~B();~D()
}
```

În absenţa specificatorului virtual din declaraţia `virtual ~B()` a destructorului clasei `B`, operatorul `delete pb` activează doar destructorul `~B()` iar memoria alocată dinamic pentru atributele obiectului de clasă `D` referit de `pb` nu este eliberată. Nu se va utiliza operatorul delete decât pentru obiecte allocate dinamic de programator; ştergerea obiectelor alocate pe stiva se face automat la terminarea funcţiei.

Iniţializare, atribuire, conversie

- Dacă obiectele vor fi utilizate pentru iniţializarea altor obiecte, clasa va fi prevăzută cu constructor de copiere ce implementează semantica prin valoare. Sematica prin referinţă favorizează apariţia „obiectelor fantomă” Exemplu.

```
class String{
public:
// constructor de conversie si costructor implicit
   String (int len=0);
// constructor de conversie
   String(const char*);
   // constructor de copiere, semantica prin valoare
   String(const String & s){
   len=s.len;
   str=new char[len+1];
   if(str==NULL) exit(1);
   strcpy(str,s.str};
   }
   // destructor
     ~String(){delete str;}
   // operator atribuire, prima suprainarcare
   String & operator=(const String &d){
     if (&d==this) return *this;
     delete str;//nu in constructorul de copiere!
     len=d.len;
     str=new char[len+1];
     if(str==NULL)exit(1);
     strcpy(str,d.str);
     return *this;
   }
// operator atribuire, a doua suprainarcare
   String& operator=(const char d){
     delete str;//nu in constructorul de copiere!
```

```
    len=strlen(d);
    str=new char[len+1];
    if(str==NULL)exit(1);
    strcpy(str,d);
    return *this;
  }

private:
  int len;
  char *str;
};
```

În absenţa constructorului `String(String &)` cu semantica prin valoare din clasa `String,` se utilizează constructorul de copiere predefinit care are semantica prin referinţă. În prezenţa destructorului `~String(){delete str;},` obiecte fantomă pot apărea:

1. după apelul unor funcţii cu argumente transmise prin valoare.

```
void f(String s){// transmitere argument prin valoare
}
String x="abcd";
f(x);
     // x a fost  copiat de constructorul de copiere implicit,
     // care are semantica prin referinta
     // la terminarea lui f, este apelat ~String()
     // care elibereaza zona alocata pentru "abcd"
     // x este acum obiect fantoma!
f(x);
```

`x` a fost modificat la primul apel al lui `f,` chiar daca nici-o instrucţiune nu apare în corpul ei.

2. la iesirea din blocuri de instrucţiuni

```
String v="abcd";
{  String t=v;
   // constructorul predefinit, semantica prin referinţă
   t="xx";
}
// acum v are in componenta un  obiect fantoma!
```

- Dacă obiectele sunt utilizate în operaţii de atribuire `s=d,` operatorul de atribuire trebuie supraîncărcat şi implementat prin semantica prin valoare; se va avea în vedere:

1. înainte de atribuire, eliberarea spaţiului de memorie alocat dinamic obiectului `s`
2. verificarea cazului de autoatribuire `s=s,` pentru a evita pierderea informaţiilor prin acţiunea de la punctul 1.
3. rezultatul returnat prin referinţă, pentru ca efectul expresiei `(a=b)=c` să fie echivalent cu `a=c`
4. supraîncărcare multiplă a operatorului de atribuire, pentru a evita conversiile implicite. În absenţa celei de adoua variante de supraîncărcare din exemplul următor, în evaluarea expresiei `s="abcd"` se utilizează constructorul de conversie înainte de aplicarea operatorului de atribuire..

- Un constructor ce poate fi utilizat cu un singur argument care nu este din clasaconstructorului se numeşte constructor de conversie. De exemplu, `R(long n=0, long d=1)` este un astfel de constructor. Constructorii de conversie permit relaxarea regulillor de verificare tare a tipurilor din C++ prin conversia implicită:
    - a argumentelor: `f(1)` dacă `f` este declarată prin `f(R r)`
    - a rezultatelor returnate: `R g(){return 1;}`
    - a operandului din dreapta operatorului de atribuire: `r=1`
    - a expresiei de iniţializare: `R r=1;`

Tehnica trebuie utilizată cu prudenţă deoarece poate afecta eficienţa, mai ales în cazul semanticii prin valoare.

- Nu se face conversie implicită în cazul transmiterii prin adresă, dar conversia explicită este posibilă (în acest caz constructorul de conversie nu joacă nici un rol şi poate să lipsească).
  Exemplu.

```
R f(R *p){return *p + *p;}
int x;
f(&x); // eroare
f((R*)&x);
// correct, dar operator+ va lucra pe o zona de memorie improprie;
// de efectele acestui  tip de conversie
// este responsabil programatorul.
String s;
f((R*)s);// sintactic correct, chiar daca nu exista
         // constructor de conversie de la String la R
```

În cazul transmiterii prin referinţă, se face conversie implicită dar programatorul este avertizat că se utilizează o variabilă temporară
Exemplu.

```
void f(R &r){ r=R(1,2);}
int x;
f(x); // correct, este modificata o variabila temporara
```

Testarea claselor C++

Este recomandabil ca specificare şi implementarea unei clase să fie însoţită de realizarea unui program de test (test driver) prin care să se execute cazurile de test şi să se înregistreze rezultatele. Relaţia dintre clasa testată CUT (Class Under Test) şi programul de test poate avea mai multe forme:

1. Testarea se face printr-o funcţie `main()` inclusă în fişierul de implementare a clasei şi care poate fi compilată condiţionat cu directiva #define, după următoarea schemă:

```
// fisier de specificare CUT.h
#ifndef CUT_H
#define CUT_H
class CUT{
public: // interfata de functionaliate
      void  m();
```

```
};
#endif

 // fisier de implementare CUT.cpp
#include ”CUT.h”
#define CUT_TEST
void CUT::m(){
    // implementare metoda m()
}

#ifdef  CUT_TEST
// urmeaza componenta test driver,
// compilata conditionat
void main(){
     // executa cazurile de test
     // si se raporteaza rezultatele
}
#endif
```

După efectuarea testelor, fişierul de implementare CUT.cpp se recompilează după ce a fost eliminată directiva `#define C_TEST`.

2. Testarea se face prin metode (eventual statice) ale clasei CUT. În acest fel interfaţa de funcţionaliate a clasei CUT este completată cu o interfaţă de testare încorporată (built-in testing iterface). Prin intermediul acestei interfeţe, orice client al clasei CUT poate realiza teste proprii, în codiţii specifice de utilizare a clasei. O posibilă schemă de utilizare este următoarea:

```
 // fisier de specificare CUT.h
#ifndef CUT_H
#define CUT_H
class CUT{
public:
      // interfata de functionalitate
      void  m();
      // interfata de testare incorporata
      void static tester();
};
#endif

 // fisier de implementare CUT.cpp
#include ”CUT.h”
void CUT::m(){
    // implementare metoda m()
}

void CUT::tester(){
     // executa cazurile de test
```

```
      // si se raporteaza rezultatele
}

// client al clasei CUT
#include ”CUT.h”
void main(){
   // se face testare,
   // utilizand interfata de testare  a clasei CUT
   CUT::tester();
}
```

3. Testare prin intermediul unei componente de testare asociată. Aceasta este o altă clasă, numită CUTtester, al cărei rol este de a specifica şi implementa, separat de clasa CUT, o interfaţă de testare asociată ce conţine atât operaţii de realizare a testelor cât şi operaţii de înregistrare şi raportare a rezultatelor. Acţiunea de testare se bazează pe aşa numitele cazuri de testare (test cases): acestea sunt acţiuni elementare al căror scop este testarea unui anumit subansamblu al clasei CUT: constructor, metodă de setare, metodă de afişare etc. Cazurile de testare pot conţine la rândul lor subcazuri de testare (subcase test): cazul de testare a unui constructor ar putea de pildă conţine drept subcazuri toate variantele de supraîncărcare existente, precum constructor fără parametri, constructor cu parametri etc. Aceste cazuri de testare se execută de regulă în înlănţuire, sub forma unor secvenţe de cazuri (test suites). Este recomandat ca aceste secvenţe de test să urmărească testarea elementelor fundamentale, testarea funcţionalităţii, testarea aspectelor de structură şi testarea elementelor de interacţiune.
    a. Se consideră elemente fundamentale metodele constructor, metodele de setare a atributelor, metodele de acces la atribute, operatorul de atribuire (=) şi cei relaţionali (==, <, > etc.). Acestea vor fi incluse în secvenţa BaseLineSuite. ~~Într-adevăr,~~Ele sunt considerate fundamentale deoarece dacă ~~acestea~~ sunt incorecte, atunci şi rezultatele raportate de alte cazuri de testare sunt nesigure.
    b. Secvenţele funcţionale (FunctionalSuite) conţin cazurile de test identificate din elementele de specificare a clasei şi au rolul de a verifica precondiţiile, postcondiţiile, proprietăţile invariante etc. Un exemplu tipic de secvenţă funcţională este cea prin care se verifică proprietăţile invariante ale clasei (metoda CUTinvariantsHolds()).
    c. Secvenţele structurale (StructuralSuite) conţin cazuri de test identificate din elementele de implementare a clasei: structuri de date, instrucţiuni etc.
    d. Secvenţele de interacţiune (InteractionSuite) completează celelalte tipuri prin luarea în considerare a unor teste de verificarea a rezultatelor unor secvenţe de evenimente precum operaţiile de intrare, acţiunea asupra unor elemente de interfaţă (butoane etc.)

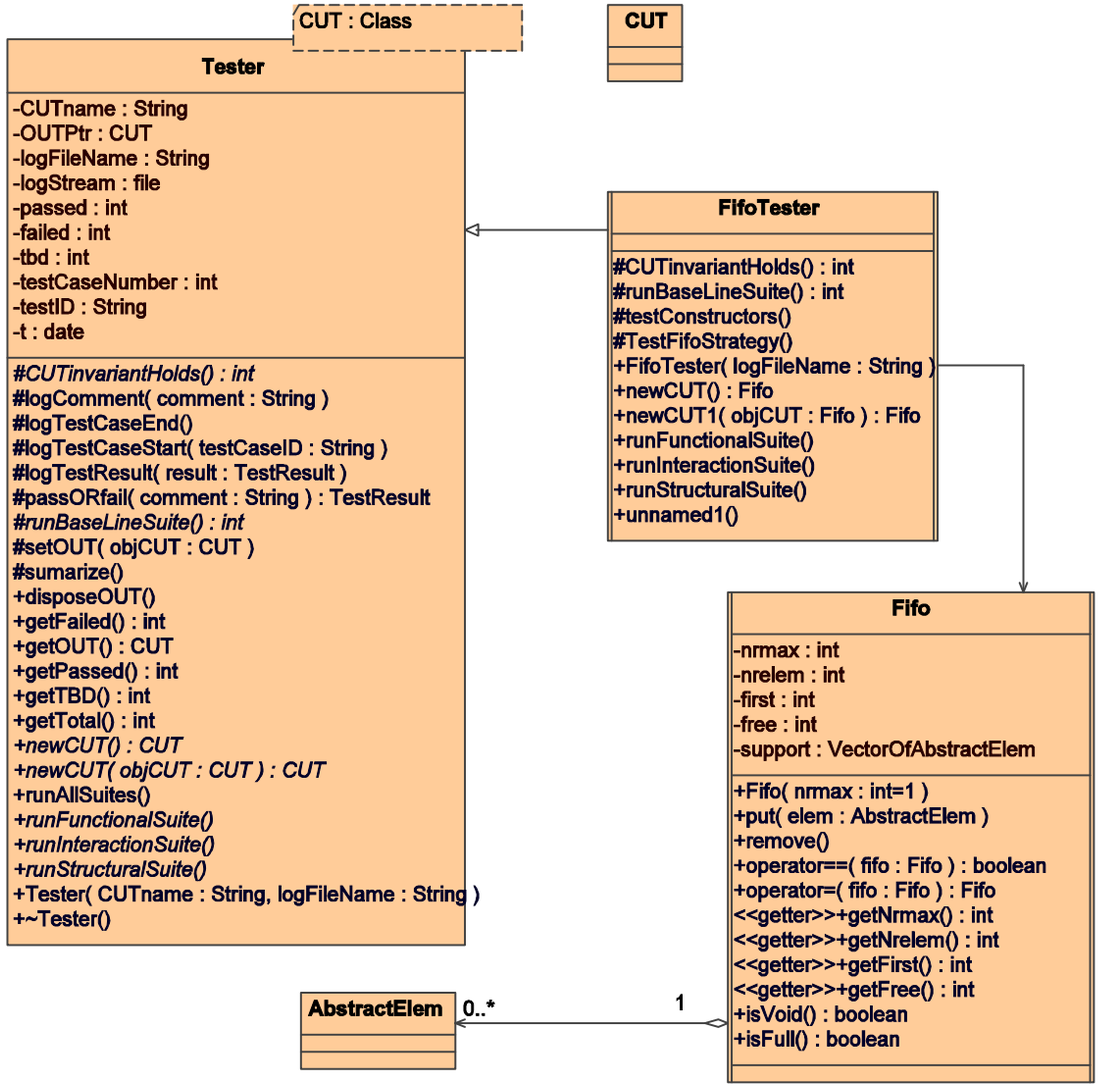
Deşi graniţa între aceste categorii de secvenţe este imprecisă, este recomandat să se facă efortul de clasificare a secvenţelor de test pentru a facilita fazele ulterioare de întreţinere (maintenance).
Executarea secvenţei de test BaseLineSuite se va face la crearea unui obiect CUTtester. Interfaţa de testare mai conţine de asemenea metodele runFunctionalSuite(), runStructuralSuite(), runInteractionSuite şi runAllSuites() pentru ececutarea secvenţelor de test.
Înregistarea şi raportarea rezultatelor unui test se face intr-un fişier.
Rezultatul unui test poate fi Pass (dacă testul a dat rezultatul aşteptat), Fail (dacă testul nu a furnizat rezultatul aşteptat) sau TBD (To Be Determined) în cazul în care rezultatul testului trebuie apreciat prin observare (de pildă, atunci când prin test se desenează pe monitor un anumit contur, se afişează un text într-un anumit font etc.)
Procedura pentru executarea şi înregistrarea unui caz de test constă în executarea următoarei secvenţe de metode:

```
// 1. in fisier se marcheaza inceputul cazului
logTestCaseStart(testID);
// 2. se executa cazul de test si
// 3. se analizeaza rezultatul
// 4. in fisier se inregistreaza rezultatul
logTestResult(result);
// 5. in fisier se marcheaza sfarsitul cazului
logTestCaseEnd();
```

La pasul 4. argumentul result se poate obţine prin metoda `TestResult passORfail(int condition)`. Aceasta returnează rezultatul Pass în cazul când argumentul condition este true şi Fail când argumentul este false. Spre exemplu, `logTestResult(passORfail(OUT->getNrmax()==1));` înregistrează rezultatul Pass dacă metoda getNrmax() returnează valoarea aşteptată 1 şi înregistrează rezultatul Fail în caz contrar. Rezultatul To Be Determined se înregistrează prin `logTestResult(TBD)`.
Cazurile de test au ca subiect un anumit obiect al clasei CUT, numit OUT (Object Under Test), referit printr-un pointer OUTPtr. El se constuieşte prin metoda `virtual CUT *newCUT ()` (sau o variantă cu parametri), este înregistrat drept Object Under Test prin metoda `void setOUT(CUT *OUTPtr)` şi este accesat prin `virtual CUT *getOUT()`, ca în exemplul următor:

```
setOUT(newCUT(10));
OUT=getOUT();
logTestResult(passORfail(OUT->getNrmax()==10));
```

Prin metoda `virtual void disposeOUT()` obiectul OUT poate fi şters.
În proiectarea clasei `CUTtester` există o serie de aspecte de interes general, comune oricărei clase de test. Acestea sunt incluse într-o clasă abstractă `Tester`, prin specializarea căreia se obţine clasa `CUTtester`.
În diagrama din Figura ??? sunt prezentate principalele idei privind proiectul unei clase de test.




```
// abselem.h
#ifndef ABSTRACTELEM_H
#define ABSTRACTELEM_H
class AbstractElem{
public:
virtual void display(){}//=0;
virtual void process(){}//=0;
};
#endif
//elements.h
```

```
#include "abselem.h"
#include "iostream.h"
class Person: public AbstractElem{
public:
Person(char *name);
virtual void display();
virtual void process();
private:
char *name;

};
class Car: public AbstractElem{
public:
Car(char *name);
virtual void display();
virtual void process();
private:
char *name;

};
//fifo.h
#ifndef FIFO_H
#define FIFO_H
#include "abselem.h"

class Fifo{
public:
Fifo(int nrmax=1);
void put(AbstractElem *);
AbstractElem* get();
void remove();
int operator==(const Fifo &);
Fifo &operator=(const Fifo &);
int getNrmax()const {return nrmax;}
int getNrelem()const {return nrelem;}
int getFirst()const {return first;}
int getFree()const {return free;}
int isVoid(){return (nrelem==0);}
int isFull(){return (nrelem==nrmax);}
private:
int nrmax;
int nrelem;

int first;
int free;
AbstractElem* *support;
};
#endif


// file tester.h
#include <fstream.h>
#include <dos.h>
#include <stdlib.h>
enum TestResult{Fail,TBD,Pass};

template <class CUT>
```

```
class Tester {
public:
Tester(char * CUTname, char *logFileName)
:CUTname(CUTname),OUTPtr(NULL),passed(0), failed(0), tbd(0),
testCaseNumber(0){
gettime(&t);
logStream = new ofstream(logFileName,ios::out);
(*logStream) <<"Testing "<<CUTname<<" started at: H: "
<<(int)t.ti_hour<< " M:"<<(int)t.ti_min<< " S: "<<(int)t.ti_sec<<endl;
}
virtual ~Tester(){  // sumarize results in logStream and close it
sumarize();

}
virtual void runAllSuites(){
runFunctionalSuite();
runStructuralSuite();
runInteractionSuite();
}
virtual void runFunctionalSuite()   =0; // abstract methods
virtual void runStructuralSuite()   =0;
virtual void runInteractionSuite()  =0;
virtual CUT *newCUT ()       =0;
virtual CUT *newCUT(const CUT &)        =0;
virtual CUT *getOUT(){return OUTPtr;}
virtual void disposeOUT(){ // finish use of current OUT
if(! OUTPtr) {delete OUTPtr; OUTPtr=NULL;}
}

int getPassed() const {return passed;}
int getFailed() const {return failed;}
int getTBD() const {return tbd;}
int getTotal() const {return passed+failed+tbd;}
protected:
virtual int runBaseLineSuite() =0;
virtual int CUTinvariantHolds()=0;
void setOUT(CUT *OUTPtr){this->OUTPtr=OUTPtr;} // used by factory
methods
// 3 methods to log test cases
void logTestCaseStart(char *testID){
this->testID=testID;  testSubCaseNumber=0;
(*logStream)<<++testCaseNumber<<" Start test case: "<<testID<<endl;
}
void logTestResult(TestResult result);
void logTestCaseEnd(){

(*logStream)<<testCaseNumber<<" End"<<endl;
}
// 3 methods to log test subcases
void logTestSubCaseStart(char *subCaseID=NULL){
this->subCaseID=subCaseID;
(*logStream)<<"\t"<<testCaseNumber<<"."<<++testSubCaseNumber<<" Start
test subcase: ";
if(!(subCaseID==NULL))(*logStream)<<subCaseID;
(*logStream)<<endl;
}
```

```
// use  logTestResult(TestResult result) for loging the result of the
subcase

void logTestSubCaseEnd(){
(*logStream)<<"\t"<<testCaseNumber<<"."<<testSubCaseNumber<<" End";
//if(!(subCaseID==NULL))(*logStream)<<subCaseID;
(*logStream)<<endl;
}

void logComment(char *comment){(*logStream)<<"\t*"<<comment<<endl;}

TestResult passORfail(int condition){// utility for not TBD
if(condition && CUTinvariantHolds()) return Pass;
else return Fail;
}
void sumarize(){// sumarize results in logStream and close it; use by
~Tester()
(*logStream)
<<endl<<"Summary of results:"<<endl
<<"Total:"<<getTotal()<<endl
<<"Passed: " <<getPassed()<<endl
<<"Failed: " <<getFailed()<<endl
<<"To be determined (TBD): "<<getTBD()<<endl;
logStream->close();
}
private:
char *CUTname;
CUT  *OUTPtr;
char *logFileName;
ofstream *logStream;
int passed, failed, tbd;
int testCaseNumber,testSubCaseNumber;
char *testID, *subCaseID;
struct time  t;  // time Tester has been created
};
// functions containing switch can not be in-line expanded
template <class CUT>
void Tester<CUT>::logTestResult(TestResult result){
(*logStream)<<"  RESULT: ";
switch (result){
case Fail: ++failed; (*logStream)<<"FAIL "; break;
case Pass: ++passed; (*logStream)<<"PASS "; break;
case TBD: ++tbd; (*logStream)<<"TBD (To be determined, see the next
comment) "; break;
default:(*logStream)<<"BAD RESULT ("<<int(result)<<")"; break;
}
(*logStream)<<endl;
}

//file fifotest.h
#include "Tester.h"
#include "fifo.h"
#include "elements.h"
class FifoTester: public Tester<Fifo>{
public:
FifoTester(char *logFileName)
:Tester<Fifo>("Fifo",logFileName){
```

```
//set, get, =, etc. Exit on FAIL!
logTestCaseStart("BaseLineSuite test case");
//set, get, =, etc. Exit on FAIL!
setOUT(new Fifo(10));
int success=
runBaseLineSuite();
logTestCaseEnd();
if (!success){sumarize(); exit(EXIT_FAILURE);}
}
virtual void runFunctionalSuite(){
//  construction test
testConstructors();
testFifoStrategy();

}
virtual void runStructuralSuite(){

}
virtual void runInteractionSuite(){}
virtual Fifo *newCUT (){return new Fifo();}
virtual Fifo *newCUT(const Fifo &obj){return new Fifo(obj);}

protected:
virtual int runBaseLineSuite(){ // verify that accessor are consistent
Fifo *out=getOUT();
// a subcase
logTestSubCaseStart("Checking nrmax");
int success=(out->getNrmax()==10);
logTestResult(passORfail(success));
logTestSubCaseEnd();
// end of subcase
return success;
}

virtual int CUTinvariantHolds(){
const Fifo OUT=*getOUT();
const nrmax=OUT.getNrmax();
const first=OUT.getFirst();
const free=OUT.getFree();
int result=(1<=nrmax)
&&(first<= (nrmax-1))
&&(free<= (nrmax-1))
&&(first>=0)
&&(free>=0);
if(! result) logComment("Invariant does not hold");
return result;
}
void testConstructors(){
Fifo *OUT;
     logTestCaseStart("Constructors");
// subcase, no args constructor
logTestSubCaseStart("Fifo()");
setOUT(newCUT());
OUT=getOUT();

logTestResult(passORfail(OUT->getNrmax()==1));
disposeOUT();
```

```
logTestSubCaseEnd();

// subcase, constructor with args
logTestSubCaseStart("Fifo(nrmax)");
setOUT(newCUT(10));
OUT=getOUT();

logTestResult(passORfail(OUT->getNrmax()==10));
disposeOUT();
logTestSubCaseEnd();

     logTestCaseEnd();
}
void testFifoStrategy(){
    setOUT(newCUT(2));
    Fifo *OUT=getOUT();
    logTestCaseStart("FIFO Strategy");
// subcase 1
logTestSubCaseStart("It is really a FIFO?");
OUT->put(new Person("Tudor"));
OUT->put(new Car("B-39-TDR"));
(OUT->get())->display();
OUT->remove();
(OUT->get())->display();
logTestResult(TBD);
logComment("The output should be: Tudor B-39-TDR");
logTestSubCaseEnd();
// subcase 2
logTestSubCaseStart("It should have a single element now");
logTestResult(passORfail(OUT->getNrelem()==1));
logTestSubCaseEnd();
// subcase 3
logTestSubCaseStart("and should be void now");
OUT->remove();
logTestResult(passORfail(OUT->isVoid()));
logTestSubCaseEnd();
    logTestCaseEnd();

}

};

//elements.cpp
#include "elements.h"
Person::Person(char *name):name(name){};
void Person::display(){cout<<name<<endl;}
void Person::process(){
       cout<<"A Person: ";
       display();
}
Car::Car(char *name):name(name){};
void Car::display(){cout<<name<<endl;}
void Car::process(){
cout<<"A Car:";
display();
}
```

```
//fifo.cpp
#include "fifo.h"
Fifo::Fifo(int nrmax):nrmax(nrmax){
first=free=nrelem=0;
support = new AbstractElem* [nrmax];
}
void Fifo::put(AbstractElem * pe){
support[free]=pe; free= ++free % nrmax; nrelem++;
}
AbstractElem* Fifo::get(){
return support[first];
}
void Fifo::remove(){
first= ++first % nrmax;
--nrelem;
}

// file testfifo.cpp
#include "FifoTest.h"
void main(){
FifoTester ft("FifoTestResults.txt");
ft.runAllSuites();
}

// file fifotest.txt
Testing Fifo started at: H: 10 M:27 S: 38
1 Start test case: BaseLineSuite test case
1.1 Start test subcase: Checking nrmax
  RESULT: PASS
1.1 End
1 End
2 Start test case: Constructors
2.1 Start test subcase: Fifo()
  RESULT: PASS
2.1 End
2.2 Start test subcase: Fifo(nrmax)
  RESULT: PASS
2.2 End
2 End
3 Start test case: FIFO Strategy
3.1 Start test subcase: It is really a FIFO?
  RESULT: TBD (To be determined, see the next comment)
*The output should be: Tudor B-39-TDR
3.1 End
3.2 Start test subcase: It should have a single element now
  RESULT: PASS
3.2 End
3.3 Start test subcase: and should be void now
  RESULT: PASS
3.3 End
3 End

Summary of results:
Total:6
Passed: 5
Failed: 0
```

To be determined (TBD): 1

Testarea asistată a metodelor

Tehnica de creare a unor programe de test (test driver) prin care să se verifice corectitudinea implementărilor poate fi supervizată prin utilizarea unor pachete specializate în asistarea activităţii de testare. Există multe pachete din această categorie care sunt distribuite gratis, precum JUnit (http://download.sourceforge.net/junit/.) sau ESCJava (Extended Static Checker for Java, http:// www.research.compaq.com/SRC/esc/).

În cele ce urmează vom prezenta câteva noţiuni elementare despre testarea asistată de JUnit. Instalarea pachetului este simplă, fiind de fapt o operaţie de dezarhivare a fişierului junit.zip descărcat de la adresa indicată. Prin dezarhivare se instalează o arhivă Java numită junit.jar care conţine toate mecanismele de asistare a activităţii de testare. Se adaugă apoi arhiva la valoarea variabilei CLASSPATH, printr-o comandă de genul

```
set classpath=%classpath%;INSTALL_DIR\junit.jar
```

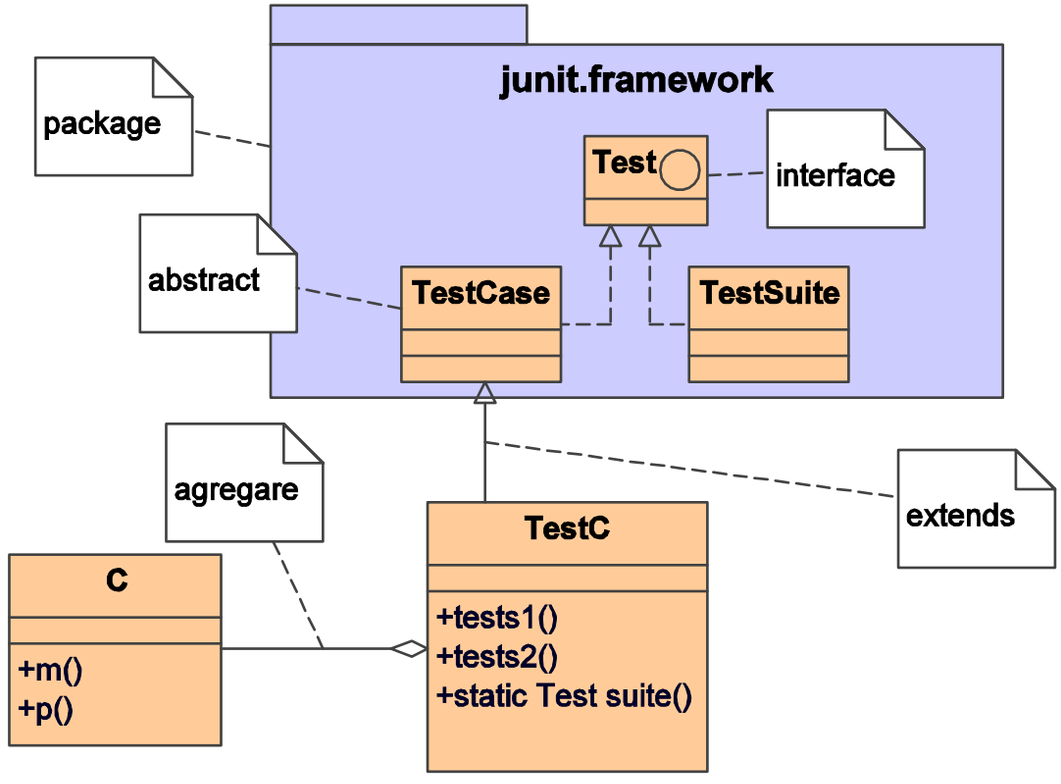
Există un mediu de testare cu interfaţă grafică ce se lansează prin comanda:

```
java junit.awtui.TestRunner
```

Dacă mediul nu are acces la clasa TestC, înseamnă că va trebui să adăugaţi la valoarea variabilei CLASSPATH şi directorul în care se află clasa de test.

Arhiva junit.jar conţine printre altele clasele Test, TestCase şi TestSuite. Metodele ce urmează să fie testate trebuie să aparţină unei subclase a clasei TestCase. Ele pot fi testate în secvenţe (test suite) foarte variate: aceste secvenţe sunt obiecte ale clasei TestSuite.

Să considerăm cazul unei clase (client) C care va folosi serviciile clasei junit.Test pentru a-şi testa metodele m() şi p(), o dată în secvenţa m();p() şi a doua oară în secvenţa p();m(). Pentru aceasta se va crea TestC, subclasă a clasei TestCase cu două metode tests1(){m();p();} şi tests2() {p();m();} ce corespund celor două secvenţe de test avute in vedere. Numele metodelor ce corespund secvenţelor de test vor conţine totdeauna prefixul „test”. Secvenţele de test sunt create prin specializarea metodei public static Test suite() din clasa TestCase. Diagrama de colaborare este prezentată în figura următoare:



Secvenţele de test (test1 şi test2 în cazul de mai înainte) pot conţine o mare varietate de *aserţiuni* prin care se confruntă rezultatele obţinute cu cele aşteptate şi se pot afişa mesaje de avertizare. Aceste aserţiuni sunt de fapt metode ale clasei junit.framework.Assert. Numele lor este sugestiv pentru verificările ăpe care le realizează: assertEquals, assertFalse, assertTrue, assertNull, assertNotNull, assertSame, assertNotSame.

Exemplu. Să presupunem că se doreşte testarea metodelor clasei CoadaCirculara. Obiectele clasei sunt cozi care au un nume şi o capacitate maximă iar elementele lor sunt dispuse într-un vector circular. Trebuie remarcat că în această clasă este specializată metoda Object.equals astfel încât două cozi sunt considerate egale în condiţii mai slabe: este suficient să aibă acelaşi conţinut, indiferent de numele lor sau de capacitatea maximă.

```
public class CoadaCirculara{
 public CoadaCirculara(int n, String s){maxDim=n; nume=s; v= new
Object[maxDim];}
 public boolean esteGoala(){return numarElemente==0;}
 public boolean estePlina(){return numarElemente==maxDim;}
 public void adauga(Object ob){
  numarElemente++;
  ultim= (ultim+1)%maxDim;
  v[ultim]=ob;
 }
```

```
 public void elimina(){
  numarElemente--;
  prim=(prim+1)%maxDim;
 }

 public Object element(){
  return v[prim];
 }

 // specializarea metodei Object.equals
 // doua cozi sunt egale daca au aceleasi elemente
 // chiar daca ele au dimensiuni maxime  sau nume diferite
  public boolean equals(Object obiectOarecare) {
    if (obiectOarecare instanceof CoadaCirculara) {
      CoadaCirculara cc= (CoadaCirculara)obiectOarecare;
      boolean b=true;
      b=b && (cc.numarElemente==numarElemente);
      int i=prim;
      int nr=numarElemente;
      while(b && (nr !=0)){
        b=b && cc.v[i].equals(v[i]);
        nr--; i=(i+1)%maxDim;

      }

      return b;
    }
    return false;
  }
 public String nume(){return nume;}
 private Object v[];
 private int maxDim; // numarul maxim de elemente
 private int prim=0;//pozitia primului element din lista
 private int ultim=-1; //pozitia  elementului  adaugat ultima data
 //lista are elementele in vectorul v, de la prim la ultim,
 //in sensul acelor de ceasrnic
 private int numarElemente=0; // numarul de elemente din coada
 private String nume;// numele cozii
}
```

Pentru testarea asistată este creată clasa TestCoadaCirculara (subclasă a clasei junit.framework.TestCase) cu o singură secvenţă de test, numită testeCombinate. În acest test sunt create două cozi c1 şi c2. Se testează dacă sunt într-adevăr goale imediat după creare, apoi se adaugă elemente, se verifică dacă sunt egale sau diferite la anumite momente etc.
Secvenţa de test este creată prin metoda public static Test suite(). Rezultatele testului se afişează prin interfaţa grafică a mediului de testare JUnit. În cazul considerat, toate verificările făcute prin aserţiuni au corespuns cu rezultatele aşteptate.

```
import junit.framework.*;
public class TestCoadaCirculara extends TestCase{
```

```
  public void testeCombinate(){
    CoadaCirculara c1=new CoadaCirculara(10, "Coada 1");
    CoadaCirculara c2=new CoadaCirculara(2, "Coada 2");
    //cozile create sunt goale, deci egale
    // asa cum se afirma in asertiunile urmatoare
    assertTrue(c1.esteGoala() && c2.esteGoala());
    assertEquals(c1,c1);
    assertNotSame(c1,c2);
    assertEquals(c1,c2);
    assertEquals(c1.nume(), "Coada 1");
    assertEquals(c2,c2);

    assertNotSame(new Integer (10),new Integer (10));
    c1.adauga(new Integer (10));
    c2.adauga(new Integer (10));
    // cozile sunt inca egale,
    // desi nu a fost adaugat acelasi obiect
    // ci obiecte egale
    assertEquals(c1,c2);


    c1.adauga("sir");
    c2.adauga("sir");
    // acum c2 este plina, dar c1 nu
    assertEquals("Ar fi trebuit sa fie egale!", c1,c2);
    assertTrue(c2.estePlina());
    assertFalse(c1.estePlina());

    c1.elimina();
    // cozile nu mai sunt egale
    assertFalse("Ar fi trebuit sa nu fie egale!",c1.equals(c2));
    c2.elimina();
    // acum sunt egale si primul element este "sir"
    assertEquals(c1,c2);
    assertEquals(c1.element(),"sir");

    c2.elimina();
    // c2 este goala
    assertTrue(c2.esteGoala());
  }
  public static Test suite(){
    return new TestSuite(TestCoadaCirculara.class);
  }
}
```

Interfeţe. În limbajul Java o clasă poate moşteni caracteristici de la cel mult o clasă de bază (moştenire simplă). Modelarea sistemelor de obiecte reale poate fi anevoioasă sub această restricţie, dar există mecanisme complementare care pot suplini absenţa conceptului de moştenire multiplă.

Să considerăm cazul unei clase server S utilizată de o clasă client C, ca in figura de mai jos. Colaborarea dintre clasa C şi orice altă clasă derivată din S se face numai prin intermediul caracteristicilor de interfaţă ale clasei server S, adică cele

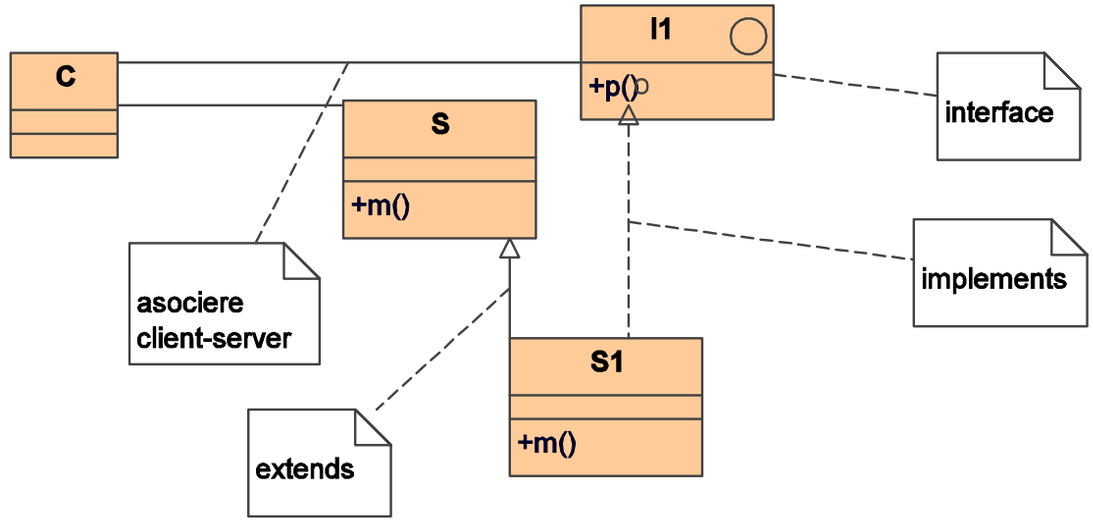
declarate cu specificatorul “public”. În cazul considerat, C are acces la oricare din implementările metodei m() din clasele server S şi S_1. Există situaţii când în clasa C se doreşte “extinderea” interfeţei moştenite de la S prin adăugarea unor facilităţi suplimentare de prelucrare a obiectelor din clasele derivate din serverul S. În limbajul Java există o construcţie sintactică asemănătoare unei clase abstracte, numită “interfaţă”,care se introduce cu ajutorul cuvântului cheie **`interface,`** ca în exemplul următor:

```
interface I1{
void p();
}
```

Prin această construcţie se stabilesc noi elemente de comunicare client-server (metoda p() în cazul considerat), prin urmare utilizarea aceluiaşi termen pentru cele două situaţii (interfaţa declarată a serverului şi elemente suplimentare de interfaţă) nu este deloc abuzivă.

Pentru ca o clasă derivată din S să-şi extindă interfaţa este necesar ca ea să implementeze una sau mai multe construcţii de tip **`interface`** (cuvântul cheie **`implements`**). Prin aceasta se înţelege că se implementează toate metodele interfeţei, fără excepţie. Trebuie observat că o clasă, deşi poate avea cel mult o clasă de bază, poate implementa mai multe interfeţe, prin aceasta eliminând dificultăţile de proiectare datorate moştenirii simple. În interfeţe diferite pot exista metode cu signaturi identice iar clasa va furniza o singură implementare; aceasta va fi asociată cu ambele metode din cele două implementări.

La rândul ei, pentru a putea beneficia de serviciile claselor server care implementează o interfaţă I1, in clasa client C trebuie să existe variabile de tip I1. Prin urmare, o interfaţă defineşte un tip de date: variabila de tip I1 poate referi obiecte din orice clasă care implementează pe I1, ceea ce oferă posibilitatea de a scrie programe cu mare grad de generalitate.



Exemplu. Considerăm în rolul clasei server S clasa numită Persoana: iar în rolul clasei S_1 clasa PersoanaSpecializata, după cum urmează:

```
public class Persoana{
  public Persoana(String nume, int varsta){
    this.nume=nume; this.varsta=varsta;
  }
  public Persoana(){}
  protected String nume="XXXX";
  protected int varsta=0;
}
```

Servicile clasei Persoana sunt minimale şi au fost extinse în clasa PersoanaSpecializată prin itermediul interfeţelor Actualizare şi Afişare.

```
public interface Actualizare{
  public void actualizareNume(String nume);
  public void actualizareVarsta(int varsta);
}

interface Afisare{
  public void afisare();
  public void actualizareVarsta(int varsta);
}
```

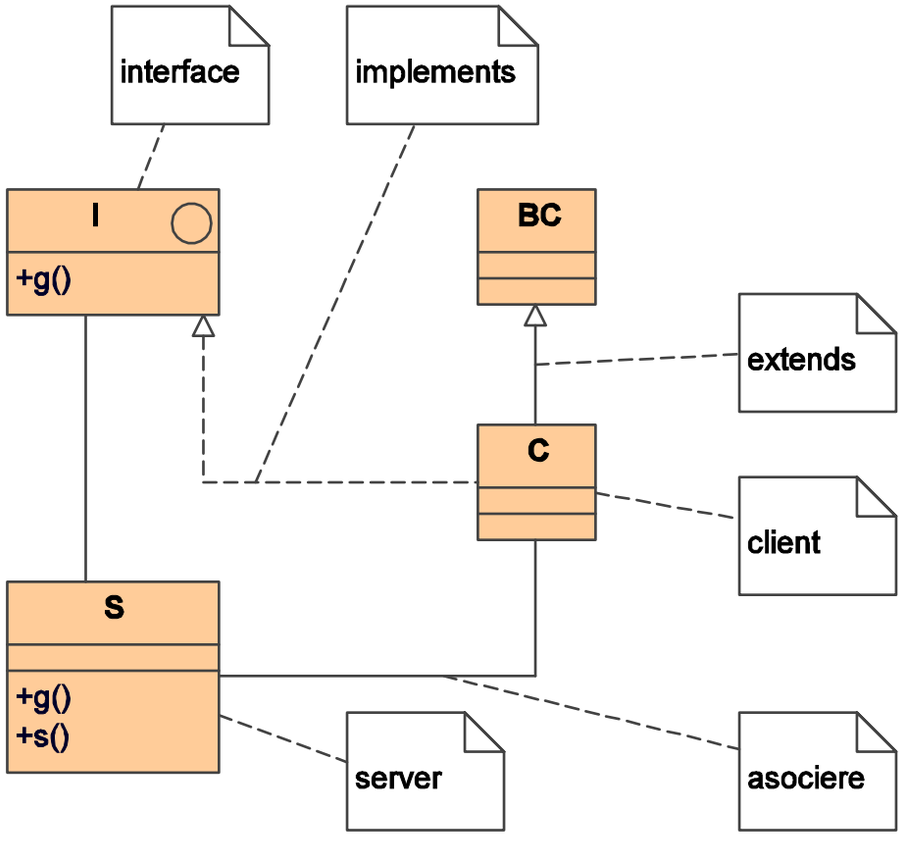
```
class PersoanaSpecializata extends Persoana
implements Actualizare, Afisare{
  public void actualizareNume(String nume){this.nume=nume;}
  public void actualizareVarsta(int varsta){this.varsta=varsta;}
  public void afisare(){
    System.out.println("nume= "+nume);
    System.out.println("varsta= " + varsta);

  }
}
```

Clasa Client utilizează serviciile serverului PersoanaSpecializată, după următoarea schemă:

```
public class Client{
  public static void main(String args[]){
    PersoanaSpecializata p=new PersoanaSpecializata();
    Actualizare act=p;
    Afisare afis=p;
    act.actualizareNume("Balanescu");
    act.actualizareVarsta(58);
    afis.afisare();
  }
}
```

În mod intenţionat metoda `public void actualizareVarsta(int varsta)` apare în ambele interfeţe: trebuie remarcat că acest lucru nu conduce la ambiguitate, deoarece clasa `PersoanaSpecializata` nu poate furniza decât o singură implementare pentru această metodă.

Există şi alte situaţii în care conceptul de interfaţă îşi dovedeşte aplicabilitatea. Să considerăm cazul unei clase server S care furnizează două tipuri de servicii clienţilor: unele cu caracter general implementate complet de clasa server (metoda g() în această ilustrare) şi altele mai specifice, care depind de tipul de client cu care serverul colaborează (metoda s()). Implementarea serviciilor din ultima categorie cade prin urmare în sarcina clientului. În cazul în care clasa client este subclasă a serverului, aceste servicii pot fi implementate prin specializarea metodelor moştenite de la server. În situaţia în care clasa client este deja subclasă a unei alte clase diferite de server (BC), atunci soluţia nu poate fi adoptată deoarece în limbajul Java moştenirea multiplă nu este posibilă. În acest caz se utilizează o interfaţă I care descrie signatura metodei specifice s(). Serverul S este client al aceastei interfeţe iar interfaţa este implementată de clientul C, ca în figura următoare.



Acest tip de colaborare este frecvent în cazul interfeţelor grafice (ca în cazul arhitecturilor Model View Controller) unde evenimentele sunt instalate astfel încât să provoace apelul unor metode ce aparţin unor interfeţe implementate de utilizatori (clienţi).

Exemplu. În cele ce urmează, serverul Legitimatie (în rolul lui S) este specializat în scrierea legitimaţiilor pentru profesori universitari (clasa Profesor este în rolul clientului C). Serviciul specific este scrierea antetului universităţii din care profesorul face parte (metoda afisareAntet() a interfeţei Antet); această interfaţă este implementată în clasa Profesor, care este deja subclasă a clasei Persoana.

```
public  class Legitimatie{
  public static void scrie(Antet antet, Object o){
    antet.afisareAntet();
    ((Profesor)o).afisare();
    System.out.println("Legitimatia a fost eliberata la data:......");
    System.out.println("_____________________________________________");
  }
}
public class Persoana{
  public Persoana(String nume){this.nume=nume;}
```

```
  public Persoana(){}
  public void afisare(){System.out.println(nume);}
  private String nume="XXXX";
}
public class Profesor extends Persoana implements Antet{
  public Profesor(String nume, String universitate){
    super(nume);
    this.universitate=universitate;
  }
  public void afisare(){
    System.out.println("Nume profesor: ");
    super.afisare();
  }
  public void eliberareLegitimatie(){
    Legitimatie.scrie(this, this);
  }
  public void afisareAntet(){
    System.out.println("****"+ universitate + "****");
  }
  private String universitate="YYYY";
}

public  class Legitimatie{
  public static void scrie(Antet antet, Object o){
    antet.afisareAntet();
    ((Profesor)o).afisare();
    System.out.println("Legitimatia a fost eliberata la data:......");
    System.out.println("_____________________________________________");
  }
}
```

Arhitectura Model View Controller

În arhitectura Model View Controller (MVC) aplicaţia propriu-zisă (obiectul { Model}) este proiectată independent de contextul în care va fi utilizată. Prin urmare, obiectul model poate fi asociat ulterior cu interfeţe utilzator (obiecte { View}) de cele mai diverse tipuri. Pentru a realize interactţiunea dintre elementele de interfaţa şi aplicaţie sunt utilizate obiecte de control ({ Controller}).

Exemplu In această foarte simplificată ilustare a elementelor de arhitectură MVC, aplicaţia este alcătuită dintr-un obiect m din clasa Model, asupra caruia se poate acţiona pentru a mări cu o unitate atributul m.x (prin metoda increment()) sau pentru a inspecta valoarea acestui atribut (metoda get_x()).
 Interfaţa pe care o asociem modelului este de tip graphic şi este obiect v al clasei View. Elementul de interacţiune pe care îl avem în vedere în acest exemplu este obiectul buton v.b al clasei Button. La fiecare acţionare a sa este incrementată valoarea atributului m.x iar noua valoare este afişată în campul v.tf.

Interceptarea şi prelucrarea evenimentelor generate prin acţionarea butonului b sunt realizate prin intermediul unui obiect de control c al clasei Controller (această clasă implementează metoda void actionPerformed(ActionEvent) a interfaţei ActionListener).

Relaţia dintre clasele din acest exemplu este prezentată în Figura MVCClase. Colaborarea dintre obiectul model, obiectele de interfaţă şi obiectul de control este ilustrată în Figura MVCColaborare. Prin mesajul b.addActionListener(c) trimis către obiectul buton b se face operaţia de { instalare} a evenimentelor. După instalare, acţionarea butonului b este { notificată} prin mesajul actionPerformed(…) trimis obiectului de control. Ca răspuns, obiectul de control trimite mesaje către modelul m pentru modificarea şi afişarea atributului m.x.

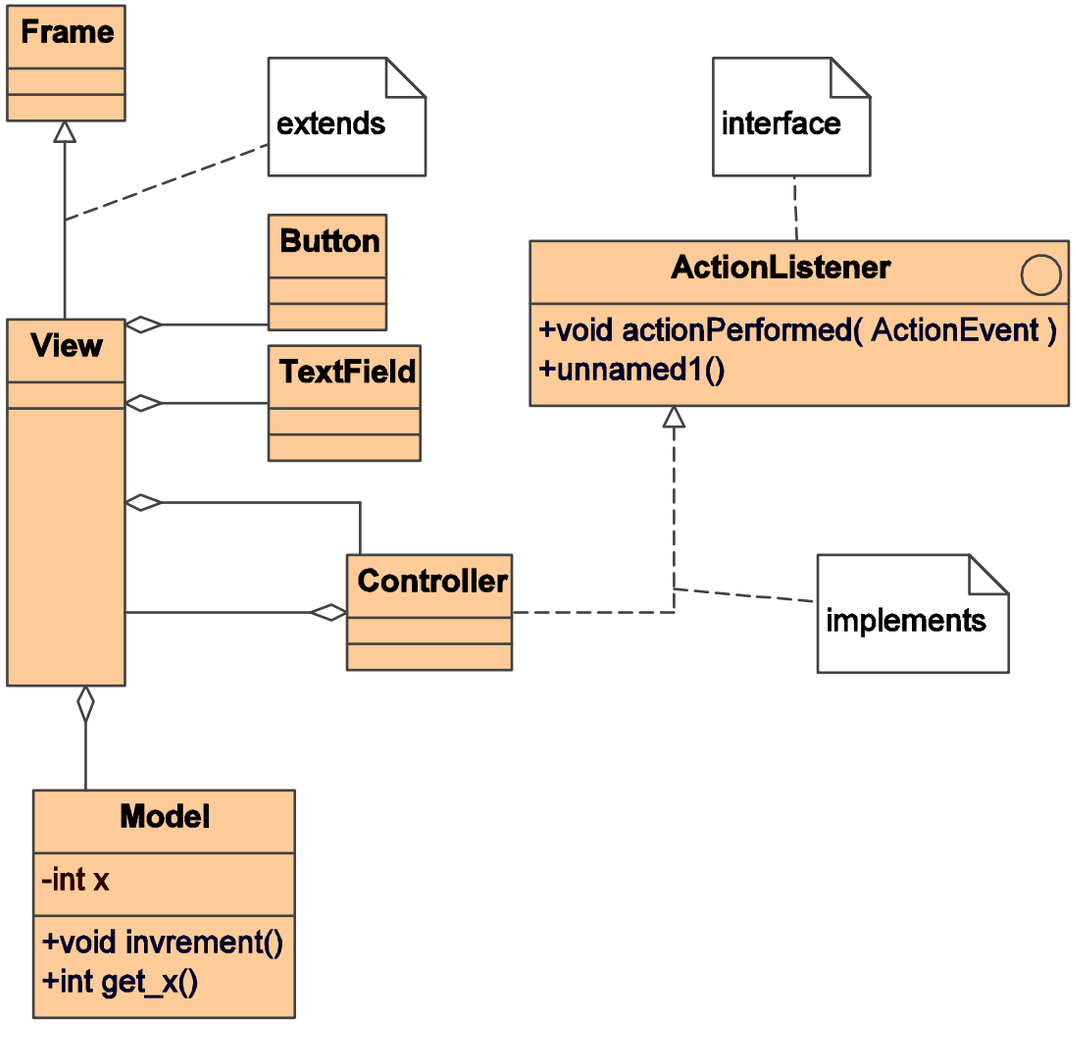


Figura MVCClase. Relaţia dintre clase

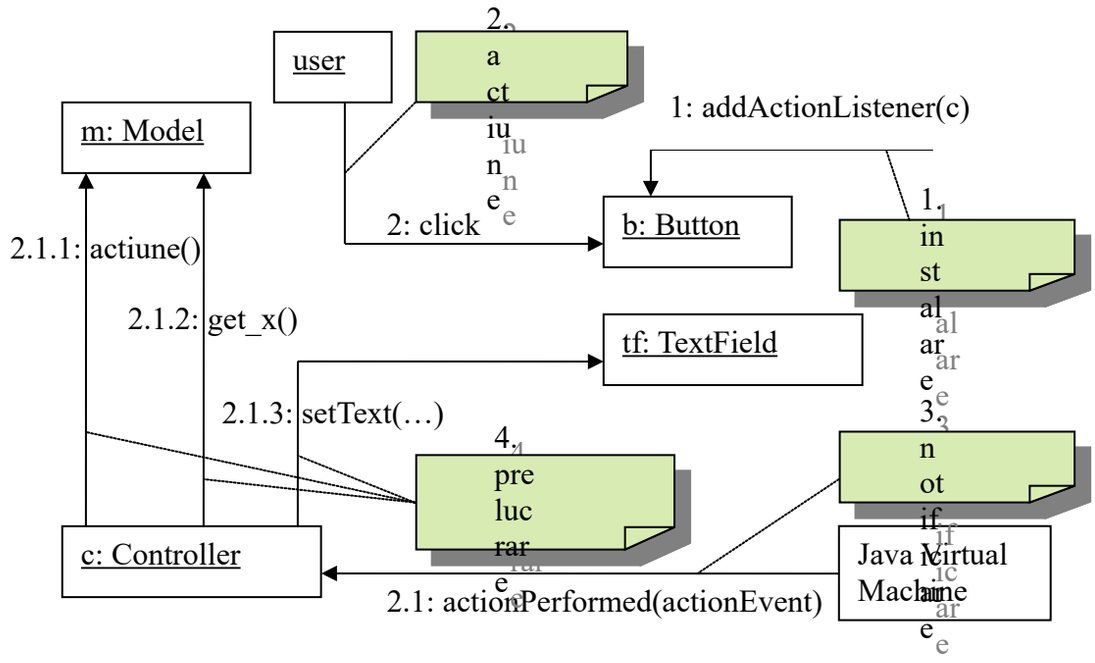


Figura MVCColaborare. Diagrama de colaborare între obiectele MVC

```
// Model View Controller

import java.awt.*;
import java.awt.event.*;
class Model{
private int x=0;
public Model(){};
public void increment(){x++;}
public int get_x(){return x;}
}
public class View extends Frame{
private Button b;
protected Model m;
private Controller c;
protected TextField tf;
public static void main(String args[]){
Frame v= new View();
}
public View(){
setTitle("Exemplu Model-View-Controller");

b= new Button("Actiune");
add("North",b);

m=new Model();

c=new Controller(this);
b.addActionListener(c);

tf=new TextField(10);
```

```
add("Center",tf);

setSize(100,250);
setVisible(true);
}
}

class Controller implements ActionListener{
private View vw;
public Controller(View v){
vw=v;
}
public void actionPerformed(ActionEvent e){
vw.m.increment();
                vw.tf.setText(String.valueOf(vw.m.get_x()));
}
}
```

Arhitectura Observer-Observable

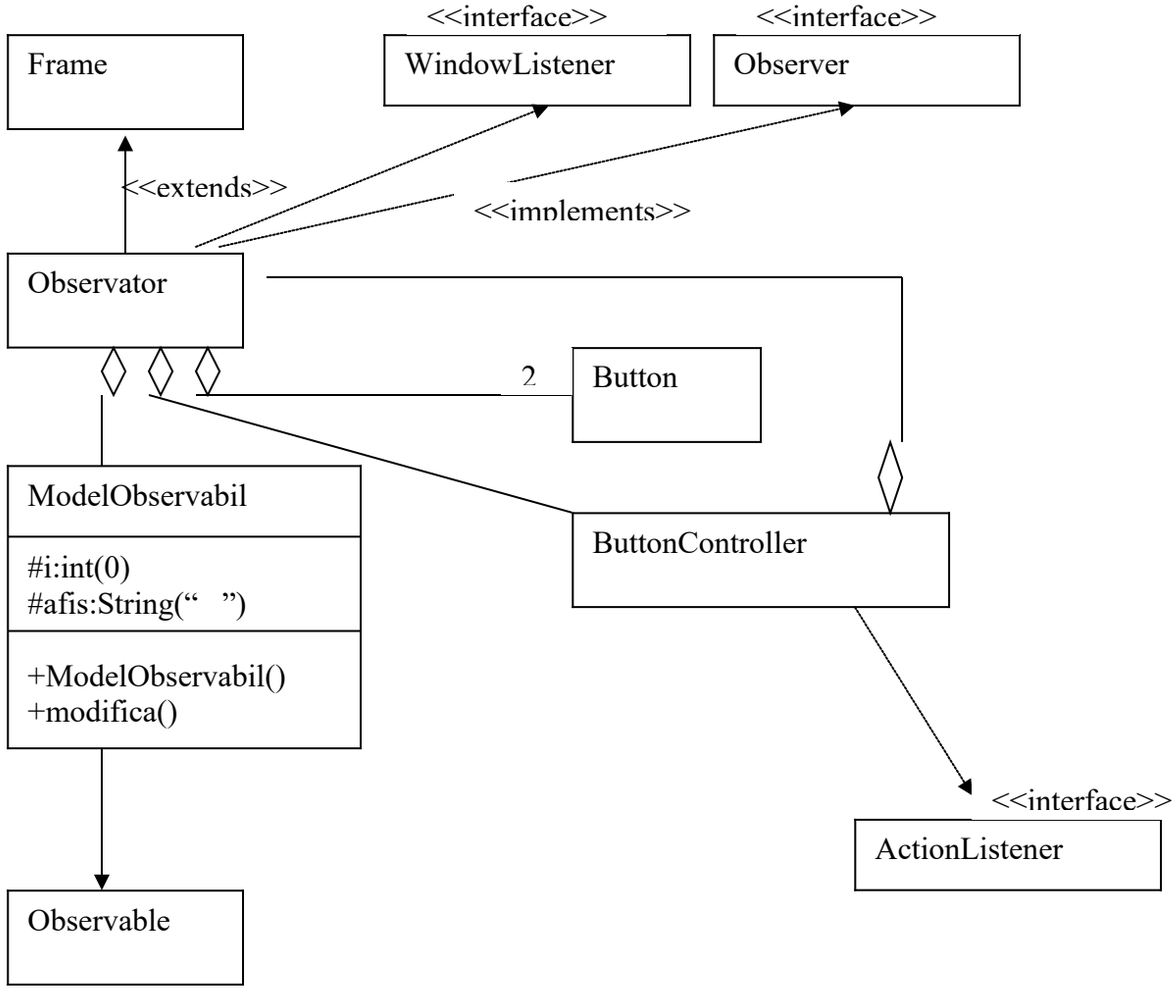


Diagrama de clase

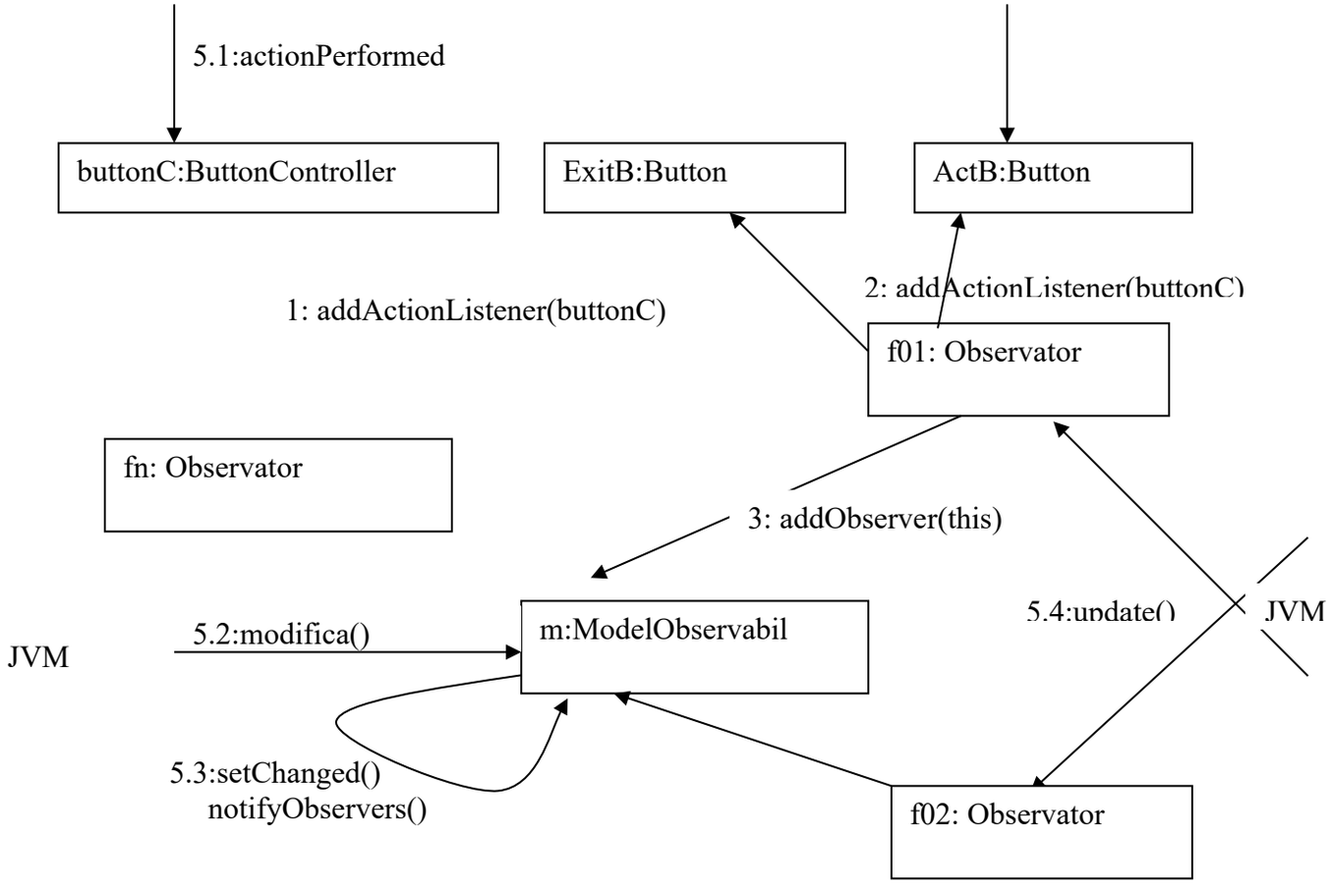


Diagrama de colaborare intre obiecte

```
//Trei ferestre si un model observabil m
// Doar fo1 si fo2 sunt observatori ai lui m
// fn nu este observator
// Apasarea butonului modifica din oricare cele trei ferestre
// incrementeaza cu 1 un atribut al modelului
// Dar numai fo1 si fo2 sesizeaza acest lucru.


import java.util.Observer;
import java.util.Observable;
import java.awt.*;
import java.awt.event.*;
public class Observator   extends Frame
implements WindowListener, Observer{
static protected ModelObservabil m=new ModelObservabil();
protected Button exitB, actB;
protected TextField field;
private ButtonController buttonC;
public static void main(String args[]){
Frame fn=new Observator(0);// nu observa pe m

Frame f01=new Observator(1);// observa pe m
Frame f02=new Observator(1);// observa pe m
}

public Observator(int i){
if (i==1)m.addObserver(this);
Panel p= new Panel();
buttonC=new ButtonController(this);
exitB= new Button("Exit");
exitB.addActionListener(buttonC);
actB= new Button("modifica");
actB.addActionListener(buttonC);
field= new TextField("MyTextField");
field.setEditable(true);
p.add(exitB);
p.add(actB);
add(field);
add("North",p);
addWindowListener(this);
```

```
setSize(160,110);
setVisible(true);

}
public void windowClosed(WindowEvent e){}
public void windowOpened(WindowEvent e){}
public void windowIconified(WindowEvent e){}
public void windowDeiconified(WindowEvent e){}
public void windowActivated(WindowEvent e){}
public void windowDeactivated(WindowEvent e){}
public void windowClosing(WindowEvent e){dispose();
//System.exit(0);
}

public void update(Observable observ, Object ob){
field.setText(m.afis);

}

}

class ButtonController implements ActionListener{
Observator view;
public ButtonController(Observator win){
view=win;
}
public void actionPerformed (ActionEvent e){
Object source=e.getSource();
if (source == view.exitB) System.exit(0);
else {
view.m.modifica();
};

}


}

class ModelObservabil extends Observable{
protected int i=0;
protected String afis="     ";
public ModelObservabil(){

setChanged();
notifyObservers();
}
public void modifica(){
i++;
afis= " "+ i;
setChanged();
notifyObservers();
}
}
```

Clase interne

O clasă internă este o clasă definită în interiorul altei clase (numită clasă externă), cu scopul de a îngrădi accesul clienţilor la anumite informaţii din obiectele clasei interne şi de a simplifica mecanismul de acces al clasei interne la caracteristicile clasei externe.
Exemplu. În cele ce urmează, sunt prelucrate depozite bancare prin intermediul metodelor de depunere şi extragere. Un depozit bancar este identificat printr-un număr de cont. Suplimentar, depozitul are un atribut prin care este identificată ultima tranzacţie la care a fost supus depozitul (depunere sau extragere) şi suma de bani implicată. Dacă se doreşte ca potenţialii clienţi ai clasei de depozite bancare să nu aibă acces la clasa de tranzacţii, această clasă se declară clasă internă (inaccesibilă) a clasei de depozite, ca în programul de mai jos. Este remarcabil modul direct în care este accesat numărul de cont în metoda public String toString() a clasei Tranzactie. *Dacă Tranzactie nu ar fi clasă internă, ar fi fost necesar un atribut suplimentar în această clasă şi transmiterea valorii acestui atribut ca argument al constructorului Tranzactie*.

```
/** Clasa de depozite bancare*/
public class DepozitBancar {
    private int numarDeCont; // numarul de identificare al depozitului
    private int valoareDepozit=0;    // valoarea curenta a depozitului
     private Tranzactie ultimaTranzactie;
                 // ultima  operatie asupra depozitului
    public DepozitBancar(int numarDeCont){
      this.numarDeCont=numarDeCont;
    }
    /** Depunere suma  */
    public void depune(int suma) {
        valoareDepozit += suma;
        ultimaTranzactie = new Tranzactie(suma, "depunere");
    }

    /** Extragere  suma  */
    public void extrage(int suma) {
        valoareDepozit -= suma;
        ultimaTranzactie= new Tranzactie(suma, "extragere");
    }

    /** metode de acces*/
    public int getNumarDeCont() { return numarDeCont; }
    public int getValoaredepozit() { return valoareDepozit; }
    public String getUltimaTranzactie() { return
ultimaTranzactie.toString(); }

    /** Clasa interna de tranzactii */
    private class Tranzactie {
        int suma;          // suma din tranzactie
        String tranzactie;  // poate fi  "depunere" sau "extragere"

        /** Constructor:  */
        public Tranzactie(int n, String t) {
           suma= n; tranzactie= t;
```

```
        }

        /** informatie explicita despre tranzactie */
        public String toString(){
          // a se remarca accesul direct la atributul numarDeCont
          return numarDeCont + ": " + tranzactie + ": " + suma;
        }
    }
}
```

Exemplu. Continuăm exemplul anterior prin considerarea unor obiecte care conţin liste de depozite bancare. Aceste liste sunt parcurse, in ordine inversă adăugării, prin intermediul unui obiect iterator din clasa internă IteratorInvers. Se poate observa încă o dată accesul simplificat la atrbutele clasei externe Banca. Clienţii clasei Banca nu pot crea obiecte ale clasei interne IteratorInvers; pentru a folosi obiecte iteratoare, clienţii utilizează interfaţa Iterator şi metoda creeazaIteratorInvers() a clasei Banca. În cazul prezentat aici, clasa Main joacă rolul de client al claselor DepozitBancar şi Banca. Se observă cum cele două obiecte iteratoare iterInvers1 şi iterInvers2 parcurg lista independent unul de celalalt.
Exista şi obiecte iteratoare din clasa interna IteratorDirect, care parcurg lista începând cu indicele 0 şi oprindu-se pe indicele `numarDeDepozite`. În aplicaţie, după ce iteratorul iterDirect staţionează pe ultimul element cu numarul de cont 10231, se adaugă un cont nou în listă. Deoarece metodele iteratorului au acces direct la atributul numarDeDepozite, metoda hasNext() lucrează pe valoarea actualizată a acestui atribut şi sesizează corect că mai sunt elemente de parcurs. Comparaţi cu situaţia în care clasa IteratorDirect nu este clasa interna a clasei Banca; în acest caz atributul numarDeDepozite va trebui transmis prin valoare ca argument la crearea iteratorului (utilizând o metoda de genul getNumarDeDepozite()) şi schimbarea lui nu mai este sesizată.

```
// fisier Banca.java
import java.util.*;

/** Un obiect  Banca este o lista de depozite bancare */
/** Lista de depozite este parcursa de la sfarsit catre inceput printr-
un iterator dat de metoda Iterator creeazaIteratorInvers()  */
/** si este parcursa de la inceput catre sfarsit printr-un iterator dat
de metoda Iterator creeazaIteratorDirect()  */
public class Banca {
   private DepozitBancar[] listaDeDepozite;
   private int numarMaximDeDepozite;
private int numarDeDepozite; //listaDeDepozite[0..numarDeDepozite-1]
   public Banca(int numarMaximDeDepozite){
       this.numarMaximDeDepozite=numarMaximDeDepozite;
       listaDeDepozite=new DepozitBancar[numarMaximDeDepozite];
   }
   public void addDepozitBancar(DepozitBancar db){
       listaDeDepozite[numarDeDepozite++]=db;
   }
     /** Clasa interna de iterare (in ordine inversa) */
```

```
    private class IteratorInvers implements Iterator {
       private int n= numarDeDepozite; // listaDeDepozite[0..n - 1]

        /** = "nu mai exista elemente de iterat" */
       public boolean hasNext() { return n > 0; }

        /** = urmatorul depozit ce urmeaza in ordinea de iterare */
       public Object next() {
          if (hasNext()) n= n - 1;
          // altfel, iteratorul ramane pozitionat
          // pe ultimul element iterat
          return listaDeDepozite[n];
       }

        /** remove nu este implementata*/
        public void remove() {}
    }

  /** Aceasta metoda a clasei Banca furnizeaza clientilor
        iteratoare de parcurgere de la sfarsit la inceput.
        Este utilizata o clasa interna
  */
  public Iterator creeazaIteratorInvers() {
       return new IteratorInvers();
  }

    /** Clasa interna de iterare (in ordine directa) */
  private class IteratorDirect implements Iterator {
        private int n= 0; // listaDeDepozite[0..n - 1]
        /** = "nu mai exista elemente de iterat" */
        public boolean hasNext() { return n < numarDeDepozite; }

        /** = urmatorul depozit ce urmeaza in ordinea de iterare */

        public Object next() {
          if (hasNext()) n= n + 1;
          // iteratorul ramane pozitionat pe ultimul element iterat
          return listaDeDepozite[n-1];
        }

        /** remove nu este implementata*/
        public void remove() {}
  }

  /** Aceasta metoda a clasei Banca furnizeaza clientilor
        iteratoare de parcurgere de la inceput la sfarsit.
        Este utilizata o clasa interna
  */
  public Iterator creeazaIteratorDirect(){
     return new IteratorDirect();
  }
}

// fisier Main.java
import java.util.*;
public class Main{
  public static void main(String args[]){
```

```
    DepozitBancar b0, b1;
    b0=new DepozitBancar(10230);
    b1=new DepozitBancar(10231);
    b0.depune(100);
    b1.depune(200); b1.extrage(75);
    // o banca mica, cu doua depozite
    Banca banca=new Banca(3);
    banca.addDepozitBancar(b0);
    banca.addDepozitBancar(b1);
    // doua obiecte iteratoare
    Iterator iterInvers1, iterInvers2, iterDirect;
    iterInvers1=banca.creeazaIteratorInvers();
    iterInvers2=banca.creeazaIteratorInvers();
    iterDirect=banca.creeazaIteratorDirect();
    // cu orimul iterator parcurgem complet
    System.out.println(
       ((DepozitBancar)(iterInvers1.next())).getUltimaTranzactie()
    );

    System.out.println(
       ((DepozitBancar)(iterInvers1.next())).getUltimaTranzactie()
    );
    // here the first iterator is on the first element of banca
    System.out.println(
       ((DepozitBancar)(iterInvers1.next())).getUltimaTranzactie()
    );
    // while the second is still at the last element
        System.out.println(
       ((DepozitBancar)(iterInvers2.next())).getUltimaTranzactie()
    );
     // iteratorul direct
    System.out.println(
       ((DepozitBancar)(iterDirect.next())).getUltimaTranzactie()
    );

    System.out.println(
       ((DepozitBancar)(iterDirect.next())).getUltimaTranzactie()
    );
    // aici iteratorul a ramas pe ultimul element
    System.out.println(
       ((DepozitBancar)(iterDirect.next())).getUltimaTranzactie()
    );
    DepozitBancar b2=new DepozitBancar(10232);
    b2.depune(2000);
    banca.addDepozitBancar(b2);
    // dar dupa adaugarea altuia, iteratorul trece mai departe
    System.out.println(
       ((DepozitBancar)(iterDirect.next())).getUltimaTranzactie()
    );
  }
}
/* resultate
*/
```

Prin executarea programului anterior se obţin următoarele rezultate:

```
10231: extragere: 75
10230: depunere: 100
10230: depunere: 100
10231: extragere: 75
10230: depunere: 100
10231: extragere: 75
10231: extragere: 75
10232: depunere: 2000
```

Clase locale. În cazul când o clasă server este utilizată doar într-o singură metodă, este natural să plasăm descrierea clasei chiar în interiorul metodei. O astfel de clasă se numeşte *clasă locală* sau *clasă internă locală*. Metoda în care este definită clasa locală este prin urmare un client al acestei clase. Pot exista şi alţi clienţi, dacă metoda respectivă creează în contextul de apelare obiecte ce aparţin clasei locale (referite ca obiecte dintr-o superclasă a clasei locale sau ca o interfaţă implementată de clasa locală). De pildă, în exemplul care a prezentat modele de iteratori pentru clasa Banca se poate observa că de fapt singurul loc în care a fost utilizată direct clasa internă IteratorDirect este metoda `public Iterator creeazaIteratorDirect()`. Aceeaşi observaţie se poate face şi despre cazul clasei interne IteratorInvers şi al metodei `public Iterator creeazaIteratorInvers()`. Apare naturală utilizarea acestor clase drept clase locale. Singura modificare se face în descrierea clasei Banca (fişierul Banca.java), după cum urmează:

```
import java.util.*;
/** Un obiect  Banca este o lista de depozite bancare */
/** Lista de depozite este parcursa de la sfarsit catre inceput printr-
un iterator dat de metoda Iterator creeazaIteratorInvers()  */
/** si este parcursa de la inceput catre sfarsit printr-un iterator dat
de metoda Iterator creeazaIteratorDirect()  */

public class Banca {
   // ca in exemplul anterior…
  /** Aceasta metoda a clasei Banca furnizeaza clientilor
        iteratoare de parcurgere de la sfarsit la inceput.
        Este utilizata o clasa locala
  */
  public Iterator creeazaIteratorInvers() {
  /** Clasa locala de iterare (in ordine inversa) */
     class IteratorInvers implements Iterator {
       // aici este o descriere a clasei  ca in exemplul anterior…
     }
      return new IteratorInvers();
  }

/** Aceasta metoda a clasei Banca furnizeaza clientilor
        iteratoare de parcurgere de la inceput la sfarsit.
        Este utilizata o clasa locala
  */
  public  Iterator creeazaIteratorDirect() {
      /** Clasa de iterare (in ordine directa) */
      class IteratorDirect implements Iterator {
```

```
        // aici este o corpul clasei  ca in exemplul anterior…
       }
      return new IteratorDirect();
   }
}
```

Clase anonime. Dacă o clasă locală este utilizată într-un singur loc din metodă, atunci nu este necesar ca ea să aibă un nume. Este cazul celor două clase locale din exemplul anterior, utilizate doar acolo unde este returnat obiectul iterator. În acest caz corpul clasei poate fi scris direct în locul (unic) de utilizare. Procedeul trebuie aplicat însă cu precauţie: structura sintactică devine stufoasă dacă numărul claselor anonime dintr-o metodă este mare.
În următoarea adaptare a fişierului Banca.java există doar cîte o clasă anonimă în fiecare din cele două metode.

```
// fisierul banca.java
import java.util.*;

/** Un obiect  Banca este … */
/** Lista de depozite este parcursa …  */
/** si este parcursa …  */


public class Banca {
   private DepozitBancar[] listaDeDepozite;
   private int numarMaximDeDepozite;
   private int numarDeDepozite; //listaDeDepozite[0..numarDeDepozite-1]
   public Banca(int numarMaximDeDepozite){
       this.numarMaximDeDepozite=numarMaximDeDepozite;
       listaDeDepozite=new DepozitBancar[numarMaximDeDepozite];
   }
   public void addDepozitBancar(DepozitBancar db){
       listaDeDepozite[numarDeDepozite++]=db;
   }


   /** Aceasta metoda a clasei Banca furnizeaza clientilor
         iteratoare de parcurgere de la sfarsit la inceput.
         Este utilizata o clasa anonima
   */
   public Iterator creeazaIteratorInvers() {
       return new Iterator(){ // clasa anonima
         // corpul clasei IteratorInvers, ca in exemplele anterioare
       }
   }

   /** Aceasta metoda a clasei Banca furnizeaza clientilor
         iteratoare de parcurgere de la inceput la sfarsit.
         Este utilizata o clasa anonima
   */
   public  Iterator creeazaIteratorDirect() {
       return new Iterator(){ // alta clasa anonima
          // corpul clasei IteratorDirect, ca in exemplele anterioare
```

```
            }
        }
    }
```

Fire de executare cu resurse partajate .

Variabile partajate, fire de executare nesincronizate

În exemplul următor, firele de executare au in comun variabila value, care inițial are valoarea 0. Fiecare fir adaugă de 100 de ori cate o unitate la valoarea value. Deoarece ele fac operația value=value +1 cu viteze diferite și nu se sincronizează, valoarea finala a variabilei nu este 200.
Orice fir asteapta inainte de a incarca valoarea value un timp sleepTimeLoad.
In acest interval este posibil ca alt fir sa modifice valoarea value.
Firul asteapta si inainte de a memora valoarea incrementata (sleepTimeStore).
Devine posibil ca firul mai rapid sa incarce pentru incrementare o valoare memorata
de firul mai lent si sa reia incrementarea de la o valoare mai mica decat cea pe care
urma s-o prelucreze.
In exemplul prezentat, t1 incrementase value de la 0 la 6; prin interventia firului
mai lent t2 value este repus la valoarea 2 si anuland o parte din actiunile lui t1:

```
t1 enters to modify the value 0
                              t2 enters to modify the value 0
t1 before modifying value
t1 enters to modify the value 1
t1 before modifying value
t1 enters to modify the value 2
t1 before modifying value
t1 enters to modify the value 3
                              t2 before modifying value
                              t2 someone interfered and changed  value!
                              t2 Sorry, I'm going to put my value
instead
                              t2 enters to modify the value 1
t1 before modifying value
t1 someone interfered and changed  value!
t1 Sorry, I'm going to put my value instead
t1 enters to modify the value 4
t1 before modifying value
t1 enters to modify the value 5
t1 before modifying value
                              t2 before modifying value
                              t2 someone interfered and changed  value!
                              t2 Sorry, I'm going to put my value
instead
t1 enters to modify the value 2
                              t2 enters to modify the value 2
t1 before modifying value
t1 enters to modify the value 3
t1 before modifying value
t1 enters to modify the value 4
```

```
class Concurrent extends Thread{
```

```
public Concurrent(int sleepTimeLoad,int sleepTimeStore, String name){
this.sleepTimeLoad=sleepTimeLoad;
this.sleepTimeStore=sleepTimeStore;
this.name=name;
}
public static int value()  {return value; }

private void incValue(){
int temp;
// incrementare value

try{
sleep(sleepTimeLoad);
} catch (InterruptedException e){}

temp=value;
System.out.println(name+" enters to modify the "+"value "+ temp);
temp++;

try{
sleep(sleepTimeStore);
} catch (InterruptedException e){}
System.out.println(name+" before modifying value ");
if(value !=(temp-1)){
System.out.println(name+" someone interfered and changed  value! ");
System.out.println(name+" Sorry, I'm going to put my value instead ");

}
value=temp;

// sfarsit incrementare
}

public void run(){// value++
for (int i=0; i<N; i++) incValue();
}

public  static void main(String[] args){
Concurrent
t1= new Concurrent(10,5,"t1"),
t2= new Concurrent(0,100,"                                t2");
t1.start();
t2.start();
while(t1.isAlive() || t2.isAlive()){}
System.out.println("NonSynchronizedConcurrent.value="+
Concurrent.value());
}

private  static int value  =0;
private  static int N   =100;
private  int sleepTimeLoad   =0;
private  int sleepTimeStore    =0;
private  String name    ="No Name Thread";

}
```

Sincronizare prin obiecte semafoare

În cele ce urmează, firele de executare îşi sincronizează acţiunile de incrementare ale variabilei partajate value. Sincronizarea se face prin procedeul de excludere mutuală la esecutarea acţiunilor din metoda incValue (orice fir de executare are acces exclusiv la variabila value pe tot timpul operatiei de incrementare).

Pentru a marca faptul ca metoda incValue este deja în proprietaea unui fir de executare la un moment dat se utilizează un obiect de sincronizare care are rolul unui semafor. El este ataşat unui bloc de instrucţiuni. Semaforul devine roşu la intrarea în bloc şi verde la ieşirea din acesta.

Un fir nu poate intra într-un bloc de excludere reciproca decat daca semaforul asociat este verde.

Exemplul din paragraful anterior se modifică doar în următoarele locuri:

Se adaugă în clasa Concurrent atributul:

```
private  static Object semaphore=new Object();
```

Toate instrucţiunile din metoda incValue, cuprinse între liniile de comentariu

```
// incrementare value
```

şi

```
// sfarsit incrementare
```

se includ în următorul bloc de excludere reciprocă:

```
synchronized(semaphore){
        // incrementare value
    //…
        // sfarsit incrementare
}
```

Variabila value este incrementată de 200 de ori, fiecare fir intervenind la anumite momente, în funcţie de viteza sa de executare. Un fir nu poate interveni însă asupra variabilei value decât dacă celălalt a terminat acţiunea de incrementare.

Iată o posibilă interfernţă a actiunilor sincronizate ale celor două fire:

```
 t1 enters to modify the value 0
t1 before modifying value
                        t2 enters to modify the value 1
                        t2 before modifying value
t1 enters to modify the value 2
t1 before modifying value
                        t2 enters to modify the value 3
                        t2 before modifying value
t1 enters to modify the value 4
t1 before modifying value
…
t1 enters to modify the value 191
t1 before modifying value
          t2 enters to modify the value 192
…
                          t2 enters to modify the value 199
                        t2 before modifying value
```

SynchronizedConcurrent.value=200

Observaţie. Obiectul semafor trebuie să fie atribut static al clasei Concurrent. În absenţa specificatorului static, fiecare fir are propriul său semafor şi sincronizarea nu se produce.

Sincronizare prin obiecte monitoare
WriterReader doar cu excludere reciproca Writer-Writer

În clasa de monitorizare Buffer, pentru excluderea reciprocă a operaţiilor de scriere se utilizează cheia obiectului current. Metoda put() are signatura:

```
public  synchronized  void put(String c, int threadID)
```

Având signatura

```
public    void get(int threadID)
```

metoda get nu are zone de excludere reciprocă şi o operaţie de citire se poate suprapune cu orice altă operaţie, fie ea de scriere sau de citire.

Sunt posibile secvenţe de forma:

rs2 **ws2** rs3 rf2 **wf2** rs1 rs5 rs4 rf1 rf5 **ws1** rf4 rf3 **wf1** rs1 **ws4** rs2 etc.

Se observă cum firele Writer 2 şi Reader 3 încep o operaţie de scriere (**ws2**) şi respectiv citire (rs3) înainte ca firul Reader 2 să termine operaţia de citire începută prin rs2. Este exclusă însă suprapunerea a două operaţii de scriere; mecanismul de excludere reciprocă nu admite secvenţe de forma

…ws1 … ws2 … wf1 …

în care un obiect Writer incepe scrierea înainte ca cel anterior să o fi terminat.

```
// director java/concurrent/task/excludere reciproca doar w-w
class Buffer{
public  synchronized
// obiectul Buffer este inchis
// pentru excludere reciproca put-put
// astfel, se asigura integritatea mesajelor (1)
void put(String c, int threadID) throws InterruptedException{
System.out.println("  ws " + threadID);

// pentru a da posibilitatea si altor fire sa intervina
for (int i=0; i<100000; i++){
String s= new String("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
}

//System.out.print(keep[current]);
System.out.println("  wf "+ threadID);
}
public
// non synchronized

// obiectul Buffer nu este inchis
// Operaţiile get-get, get-put se pot suprapune
void get(int threadID)throws InterruptedException{
System.out.println("rs "+ threadID);

// pentru a da posibilitatea si altor fire sa intervina
for (int i=0; i<100000; i++){
String s= new String("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
}

System.out.println("rf "+ threadID);
}

}
```

Clasa obiectelor consumatoare

```
class Reader extends Thread{
public void run(){
   while(true){
try{
sleep(sleepTime);
b.get(threadID);
}catch(InterruptedException e){
System.out.println("b.get: InterruptedException");
}
System.out.println();
   }
}
public Reader(Buffer b, int sleepTime){
this.b=b;
this.sleepTime=sleepTime;
threadID=++numberOfReaders;
}
private Buffer b;
int sleepTime=10;
private static int numberOfReaders=0;
private int threadID;
}
```

Clasa obiectelor producătoare

```
class Writer  extends Thread{
public void run(){
   int messageID=0;
   while(true){
messageID++;
try{
sleep(sleepTime);
b.put("Message "+ threadID+ "." +messageID + " from thread "+threadID,
threadID);
}catch(InterruptedException e){
System.out.println("b.put: InterruptedException");
}
   }
}
Writer(Buffer b, int sleepTime) {
this.b=b;
this.sleepTime=sleepTime;
threadID=++numberOfWriters;

}
private static int numberOfWriters=0;
private Buffer b;
private int threadID;
private int sleepTime=10;

}
```

Clasa test driver

```
class WriterReader{
public static void main(String[] args ){
Buffer buffer= new Buffer();
```

```
for (int i=0; i<5; i++){
Writer w=new Writer(buffer, 100+i*100); w.start();
Reader r=new Reader(buffer, 10+ i*10); r.start();
}

}
}
```

WriterReader cu excludere reciproca Writer-Writer, Writer-Reader, Reader-Reader

Dacă în clasa de monitorizare Buffer metodele put() şi get() utilizează pentru sincronizare acelaşi obiect (obiectul curent al metodei), atunci toate toate operaţiile se exclud reciproc. Prin urmare, cele două metode au signatura

```
public  synchronized  void get(int threadID)
```

şi respectiv

```
public  synchronized  void put(String c, int threadID)
```

.

Sunt posibile doar secvenţe de forma:
ws4 wf4 rs1 rf1 rs2 rf2 ws1 wf1 rs1 rf1 etc.

WriterReader doar cu excludere reciproca Writer-Writer, Writer-Reader

Sunt posibile secvenţe de forma:
rs2 rs1 rs3 rf2 rf3 rs4 rf5 rf1 rf4 **ws2 wf2** rs1 rs2 rf2 rf1 **ws1 wf1** etc.
în care o operaţie de scriere nu poate fi iniţiată dacă este deja în derulare o altă operaţie de citire sau de scriere.

```
class Buffer{
public  synchronized
// obiectul Buffer este inchis
// pentru excludere reciproca put-put
// astfel, se asigura integritatea mesajelor (1)
void put(String c, int threadID) throws InterruptedException{
while(numberOfReaders != 0) wait();
writerActive=true;
System.out.println("  ws " + threadID);

// pentru a da posibilitatea si altor fire sa intervina
for (int i=0; i<100000; i++){
String s= new String("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
}

//System.out.print(keep[current]);
System.out.println("  wf "+ threadID);
writerActive=false;
notifyAll();
}
public
// special synchronized

// obiectul Buffer nu este inchis
// pentru excludere reciproca get-get, get-put

void get(int threadID)throws InterruptedException{
// excludere put-get prin wait()
```

```
synchronized(this){while (writerActive)wait();}
synchronized (this){numberOfReaders++;}
//excludere cu testul din put() sau cu alte fire Reader!
synchronized(semaphore) { // pentru integritatea afisarii
System.out.println("rs "+ threadID);
}

// daca nici-un fir Writer nu este activ
// mai multe fire Reader pot ajunge aici simultan!
// ele sunt lasate sa lucreze nesincronizat!,
// presupunand ca nu fac decat operatii de citire

// pentru a da posibilitatea si altor fire sa intervina
for (int i=0; i<100000; i++){
String s= new String("xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx");
}

synchronized(semaphore){System.out.println("rf "+ threadID);}
synchronized(this) {numberOfReaders--;}
//excludere cu testul din put() sau cu alte fire Reader!
synchronized(this){notifyAll();}
}
private static int numberOfReaders=0;
private static Object semaphore=new Object();
boolean writerActive=false;

}
```

Transmiterea mesajelor

Transmiterea de mesaje :

1. cu păstrarea integrităţii;
2. fara pierderea mesajelor
3. un mesaj e preluat de un singur destinatar
4. un destinatar nu preia acelaşi mesaj de mai multe ori

Clasa de monitorizare Buffer

..\..\programe\Buffer.java

```
class Buffer{
public  synchronized
// obiectul Buffer este inchis
// pentru excludere reciproca put-put, put-get
// astfel, se asigura integritatea mesajelor (1)
void put(String c) throws InterruptedException{
while (count==maxSize) wait();   //evita pierderea mesajelor (2)
//firul e trecut in asteptare la acest obiect Buffer
//obiectul Buffer este deschis,
//alte fire pot utiliza put sau get
// now, count != maxSize
keep[free]=c;
count=count +1;          //
free=(free +1)% maxSize;
notifyAll();              //sincronizare; toate firele in asteptare
// la acest obiect Buffer trec in starea Ready
}
public  synchronized
// obiectul Buffer este inchis
// pentru excludere reciproca get-get, get-put
// get-get: un mesaj e preluat de un singur destinatar (3)
// get-put: se asigura integritatea mesajelor (1)
void get()throws InterruptedException{
while (count==0) wait();    //un destinatar nu preia un
//mesaj de mai multe ori (4)
//firul e trecut in asteptare la acest obiect Buffer
//obiectul Buffer este deschis,
//alte fire pot utiliza get sau put
// now, count <> 0

System.out.print(keep[current]);

count=count -1;
current=(current+1)%maxSize;
notifyAll();              //sincronizare; toate firele in asteptare
// la acest obiect Buffer trec in starea Ready
}
public Buffer(int maxSize){
this.maxSize=maxSize;
keep=new String [maxSize];
}
public int maxSize(){return maxSize;}
private int maxSize=1;
private String[] keep;
private int free=0;   // pozitia libera in care poate fi pus un mesaj nou
```

```
        private int current=0;        // pozitia in care se afla primul mesaj inca nepreluat
private int count=0;      //  numarul mesajelor nepreluate
}
```

Clasa obiectelor producătoare

..\..\programe\Writer.java

```
class Writer  extends Thread{
public void run(){
   int messageID=0;
   while(true){
messageID++;
try{
sleep(sleepTime);
b.put("Message "+ threadID+ "." +messageID + " from thread "+threadID);
}catch(InterruptedException e){
System.out.println("b.put: InterruptedException");
}
  }
}
Writer(Buffer b, int sleepTime) {
this.b=b;
this.sleepTime=sleepTime;
threadID=++numberOfWriters;

}
private static int numberOfWriters=0;
private Buffer b;
private int threadID;
private int sleepTime=10;

}
```

Clasa obiectelor consumatoare

..\..\programe\Reader.java

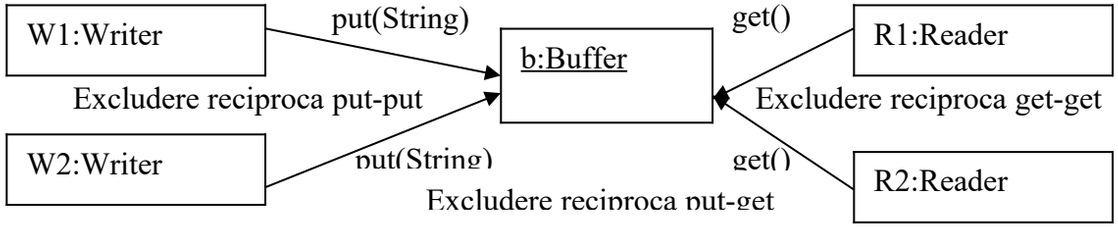
```
class Reader extends Thread{
public void run(){
   while(true){
try{
sleep(sleepTime);
b.get();
}catch(InterruptedException e){
System.out.println("b.get: InterruptedException");
}
System.out.println();
   }
}
public Reader(Buffer b, int sleepTime){
this.b=b;
this.sleepTime=sleepTime;
}
private Buffer b;
int sleepTime=10;
}
```

Clasa driver

..\..\programe\WriterReader.java

```
class WriterReader{
public static void main(String[] args ){
Buffer buffer= new Buffer(20);
for (int i=0; i<5; i++){
Writer w=new Writer(buffer, 100+i*100); w.start();
Reader r=new Reader(buffer, 10+ i*10); r.start();
}

}
}
```

Observaţia 1. Prin clauza `synchronized` metodele devin zone de exculdere reciprocă.



Numai unul din firele ce au transmis mesaje synchronized la acelaşi obiect monitor ocupă monitorul pe tot parcursul executării metodei (se spune că firul este proprietarul monitorului). Când metoda synchronized se termină, firul pierde controlul asupra obiectului monitor. Firul mai poate pierde controlul monitorului şi în cazul când i se transmite mesajul wait().

Observaţia 2. Dacă i se transmite mesajul `wait()`, monitorul trece firul de executare care are control asupra sa în starea de aşteptare `wait`. Fiecare obiect monitor are ataşată o mulţime proprie de fire în aşteptare. Dacă i se transmite mesajul `notify()` şi mulţimea firelor în aşteptare nu este vidă atunci un fir oarecare din această mulţime preia controlul asupra obiectului monitor. Prin urmare, este executată metoda care a transmis mesajul `wait()`, începând cu instrucţiunea care urmează invocării acestei metode.

Observaţia 3. În cazul programului de test (driver) anterior, sunt afişate linii de forma:
Message 3.1 from thread 3
Message 1.1 from thread 1
Message 2.1 from thread 2
Message 1.2 from thread 1
etc.

Ordinea de afişare depinde de contextul de executare, Clauza `synchronized` asigură transmiterea si afişarea liniilor fără pierderi de caractere din linie.
Instrucţiunile de sincronizare care utilizează mesajele `wait(), notify()` şi variabila de control `contor` au următorul efect:

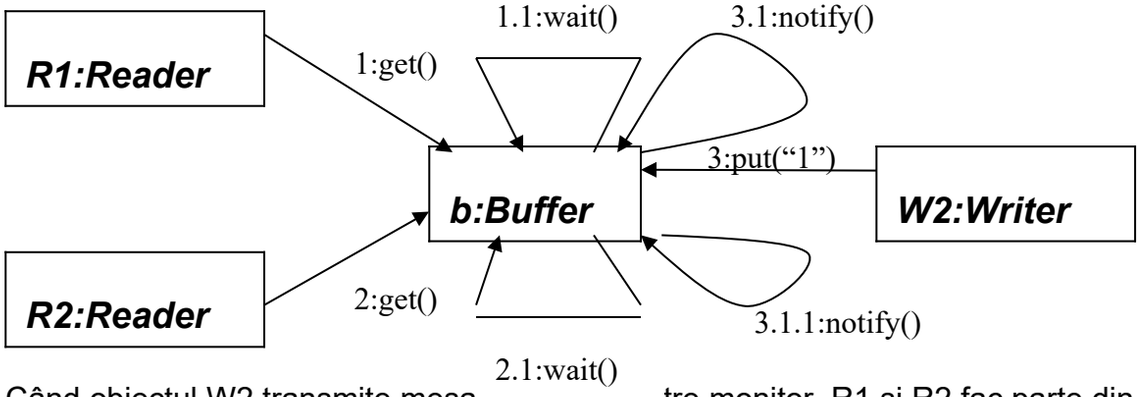
- împiedică afişarea repetată a unui mesaj provenind de la acelaşi fir de scriere

  (prin instrucţiunea `while (count==0) wait();`)
- nici-un mesaj transmis nu este pierdut
  (prin instrucţiunea `while (count==maxSize) wait();`)

Observaţia 4. Aparent, acelaşi efect de sincronizare se poate obţine prin înlocuind instrucţiunile de sincronizare
while `(count==1) wait();`
şi
while `(count==0) wait();`
prin
if `(count==1) wait();`
respectiv
if `(count==0) wait();`

În realitate, mecanismul de sincronizare astfel modificat, deşi asigură încă afişarea integrală a caracterelor dintr-o linie, nu mai este capabil să evite afişarea multiplă a unei linii
Un posibil rezultat ar putea fi de exemplu:
Message 2.1 from thread 2
Message 2.1 from thread 2
etc.

O colaborare ipotetică intre obiecte, care să justifice acest rezultat, este următoarea:

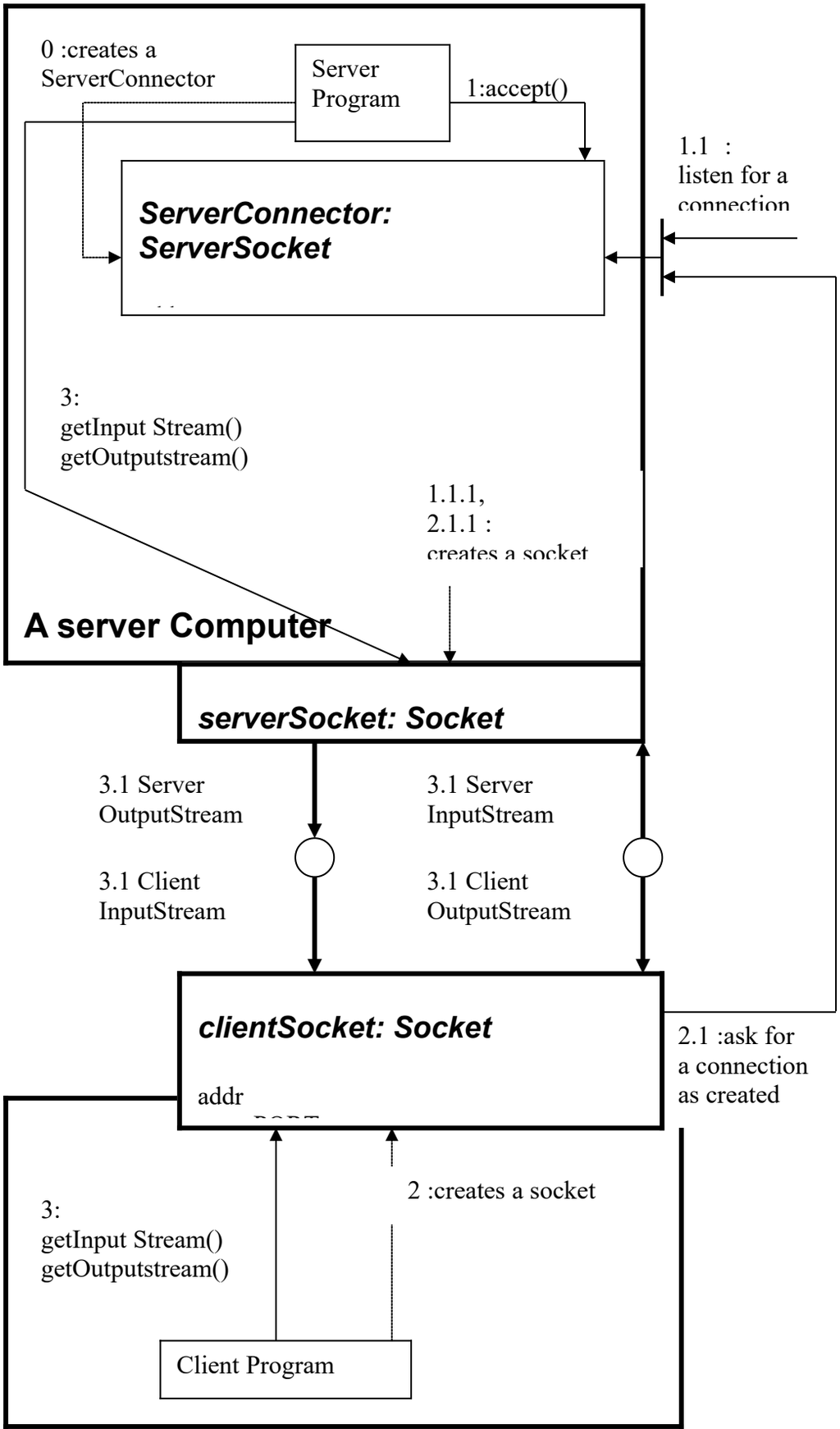


Când obiectul W2 transmite mesaj [illegible] tre monitor, R1 şi R2 fac parte din mulţimea firelor în aşteptare la acest monitor.

Mesajul 3.1:notify() transmis din metoda put() reda controlul firului R1. Executarea metodei get() se reia cu instrucţiunea ce urmează expresiei wait() care a trimis firul în aşteptare. Prin urmare se afişează prima linie. Afisarea liniei se face într-o zona de excludere mutuală, deci alte fire de executare nu pot interveni ca să altereze conţinutul liniei. Trebuie notat că valoarea variabilei de control count este pusă la valoarea 0, pentru a semnala faptul că linia a fost deja afişată.
Mesajul 3.1.1: notify() este trimis după această actualizare a variabilei count din metoda get(). Unul din firele din mulţimea de aşteptare a monitorului b, spre exemplu R2, preia controlul asupra monitorului: executarea metodei get începe însă după instrucţiunea de test if(count==0)wait(). Schimbarea variabilei de control nu mai este sesizată si linia mai este afişată incă o dată.

Observaţia 5. În absenţa clauzei synchronized, metodele get şi put nu mai sunt zone de excludere reciprocă. În acest caz este afectată şi integritatea liniilor, fiind posibilă afişarea unor linii de forma:
Message 2.157 from thread 3
unde numărul firului (3) nu mai corespunde cu prefixul numărului de mesaj (2).

Comunicare între calculatoare utilizând dispozitive “socket”



1. Cazul un obiect server şi un obiect client

Socket Connector and Server Socket

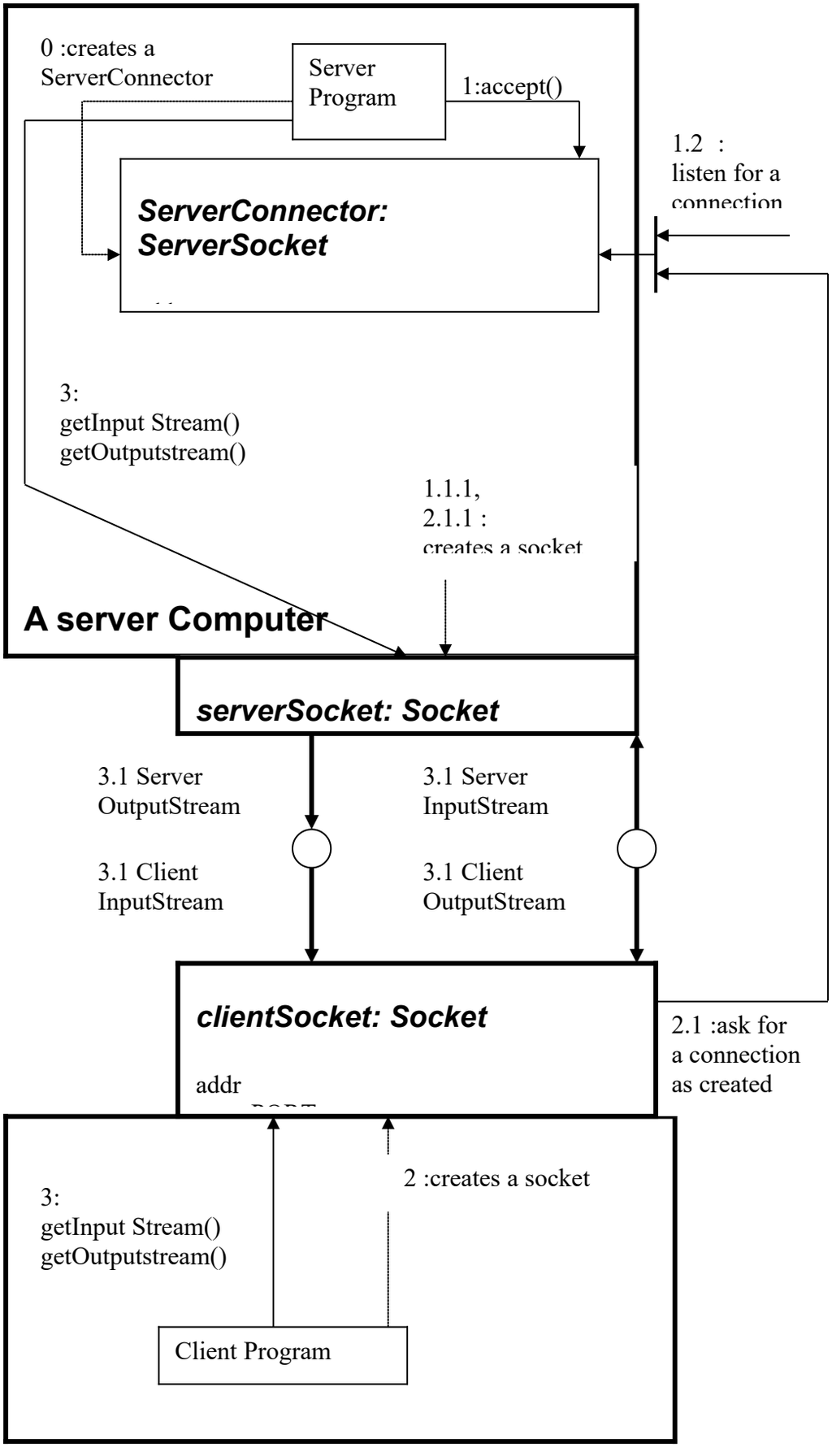
```
import java.io.*;
import java.net.*;
public class Server{
public static final int PORT=8080;
public static void main(String[] args)throws IOException{
ServerSocket serverConnector=new ServerSocket(PORT);
System.out.println("Created ServerSocket: " + serverConnector);

try{
   System.out.println("Waiting for a connection! ");
   Socket serverSocket=serverConnector.accept(); // blocks until a
connection occurs
   System.out.println("Connection accepted: " + serverSocket);
   try{

BufferedReader in=
  new  BufferedReader(new
InputStreamReader(serverSocket.getInputStream()));
BufferedWriter bw=
  new  BufferedWriter(new
OutputStreamWriter(serverSocket.getOutputStream()));
PrintWriter out=
  new PrintWriter(bw,true);
        int correctpsw=0;
        out.println("Your Password, please: " );
        for (int i=0; i<3 ; i++){
           out.println("Enter Password, please: " );
           String str=in.readLine();
           if( str.equals("t18c25ad")){correctpsw=1;
out.println("Password Accepted!"); break;}
           else{
              out.println("Wrong Password; "+ (2-i)+" attempts" );
      if(i==2)out.println("Good Bye, try later!");
           }
        }
     while(correctpsw==1){
          String str=in.readLine();
  if(str.equals("END")) break;
  System.out.println("Echoing: "+str);
  out.println("Echoing: "+str);
}
   }
   finally{
System.out.println("Closed client socket: " + serverSocket);
serverSocket.close();
   }
}
finally{
System.out.println("Close ServerConnector: " + serverConnector);
serverConnector.close();
}
}
```

```
}
```

Client Socket

```
import java.net.*;
import java.io.*;
public class Client{
public static void main(String[] args) throws IOException{
InetAddress addr= InetAddress.getByName (null);
System.out.println("InetAddress: " + addr);
Socket clientSocket=new Socket(addr, JabberServer.PORT);
System.out.println("Created a client socket:" + clientSocket);
String str;
try{
BufferedReader in=
  new  BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
BufferedWriter bw=
  new  BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()));
PrintWriter out=
  new PrintWriter(bw,true);
out.println("18c25ad");


for(int i=0; i<10; i++){
out.println("howdy " + i);
str= in.readLine();
System.out.println(str);
}
out.println("END");
}
finally{
System.out.println("The   client socket:" + clientSocket +" is going to
be closed");
clientSocket.close();
}
}
}
```

1. Cazul un obiect server şi mai multe obiecte client



Socket Connector and Server Socket

```
import java.io.*;
import java.net.*;
class ServeOneClient extends Thread{
private Socket serverSocket;
private BufferedReader in;
private BufferedWriter bw;
private PrintWriter out;
public ServeOneClient(Socket serverSocket) throws IOException{
   this.serverSocket=serverSocket;
   in=new BufferedReader(new
InputStreamReader(serverSocket.getInputStream()));
   bw=new BufferedWriter(new
OutputStreamWriter(serverSocket.getOutputStream()));
   out=new PrintWriter(bw,true);
   start();
}
public void run(){
   try{
while(true){
   String str=in.readLine();
   if(str.equals("END")) break;
   System.out.println("Echoing: " + str);
   out.println("Echoing: " + str);
}
System.out.println("Closing ...");
   }
   catch (IOException e){}
   finally{
try{
   serverSocket.close();
}
catch(IOException e){}
   }
}

}
public class MultiClientServer{
static final int PORT=8080;
public static void main(String[] args) throws IOException{
ServerSocket socketConnector=new ServerSocket(PORT);
System.out.println("socketConnector Server Started");
try{
   while (true){
     Socket serverSocket=socketConnector.accept();
try{
   new ServeOneClient(serverSocket);
}
catch(IOException e){}
   }
}
finally{socketConnector.close();}
}
}
```

Client Socket

```
import java.net.*;
import java.io.*;
public class OneClient{
static final int MAX_THREADS=10;
public static void main(String[] args)throws IOException,
InterruptedException{
InetAddress addr= InetAddress.getByName(null);
 new ClientThread(addr);
Thread.currentThread().sleep(100);
}
}
class ClientThread extends Thread{
private Socket clientSocket;
private BufferedReader in;
private BufferedWriter bw;
private PrintWriter out;
public ClientThread(InetAddress addr){
   System.out.println("Making client ");
   try{clientSocket=new Socket(addr, MultiClientServer.PORT);  }
   catch(IOException e){}
   try{
in=new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
bw=new BufferedWriter(new
OutputStreamWriter(clientSocket.getOutputStream()));
out=new PrintWriter(bw,true);
start();
   }
   catch(IOException e){
try{clientSocket.close();}
catch(IOException e2){}
   }
}
public void run(){
try{
for(int i=0; i<25; i++){
out.println(" "+ i);
String str=in.readLine();
System.out.println(str);
try{
sleep(1000);
}catch(InterruptedException ie){}
}
out.println("END");
}
catch(IOException e){
}
finally{
try{ clientSocket.close();
}catch(IOException e){}
//threadcount--;
}
}
}
```

Tehnici evoluate de programare în Java. Diagrame de stare

În acest capitol sunt prezentate elementele definitorii ale diagramelor de stare, definindu-se tranziţiile, evenimentele speciale, superstările, pseudo-stările iniţiale şi finale, modul de utilizare (folosind tabelele de stare), precum şi tehnici de implementare a acestora: prin instrucţiuni *switch-case* seriale sau prin şablonul *State*.

Noţiuni fundamentale

UML posedă o mulţime destul de bogată de notaţii folosite pentru a descrie maşinile cu stări finite (FSMs), în continuare prezentându-se cele mai importante dintre acestea. Maşinile cu stări finite reprezintă unelte extrem de folositoare în dezvoltarea diverselor aplicaţii software. De obicei, sunt folosite în crearea interfeţelor grafice (GUIs), a protocoalelor de comunicaţie, cât şi a oricărui sistem bazat pe evenimente.

În figura 1.1 este prezentată o diagramă de stare (STDs – *State Transition Diagram*) care descrie o maşină cu stări finite ce controlează modul în care un utilizator se conectează la un sistem. Dreptunghiurile cu colţurile rotunjite reprezintă *stări*. Numele fiecărei stări se află în compartimentul său superior. În partea de jos a dreptunghiului se află *acţiunile speciale*, care indică ceea ce trebuie făcut atunci când se intră sau se iese din starea respectivă. De exemplu, atunci când se intră în starea Prompting for Login, se invocă acţiunea showLoginScreen. Când se iese din această stare se invocă acţiunea hideLoginScreen.

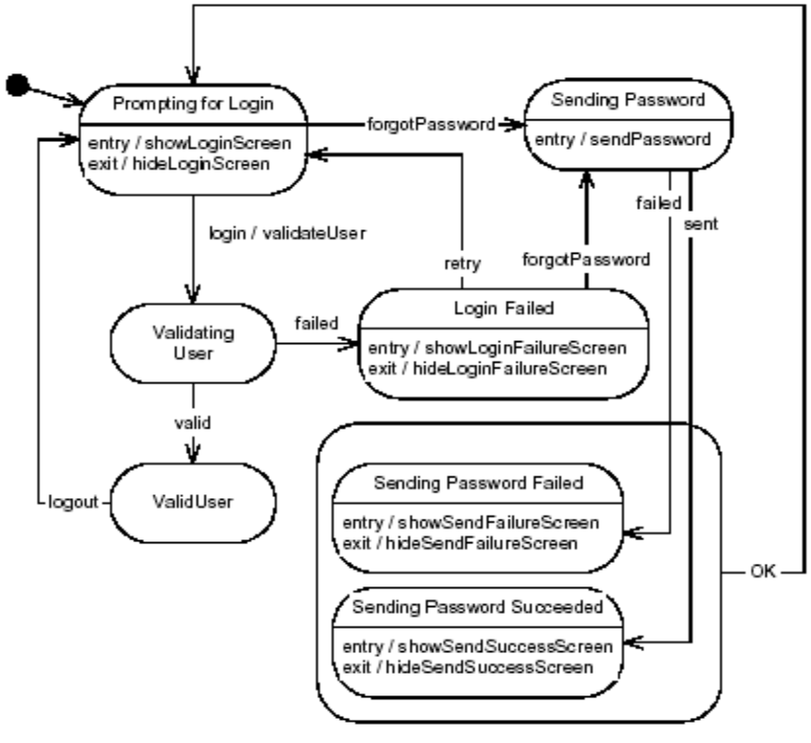


Figura 1.1 Diagrama de stare a unei FSM ce controlează modul de conectare la un sistem

Săgeţile dintre stări sunt numite *tranziţii*. Fiecare este etichetată cu numele evenimentului care declanşează tranziţia. Această etichetă poate conţine şi acţiunea care trebuie executată atunci când are loc tranziţia. De exemplu, dacă sistemul se află în starea Prompting for Login şi are loc un eveniment login, se va trece în starea Validating User şi se va invoca metoda validateUser.
Cercul negru din colţul stânga-sus al diagramei este numit *pseudo stare iniţială*. O maşină cu stări finite îşi începe activitatea printr-o tranziţie din această stare iniţială. Deci, în exemplul nostru, FSM-ul porneşte printr-o tranziţie în starea Prompting for Login.
Stările Sending Password Failed şi Sending Password Succeeded sunt incluse într-o *super stare*, deoarece ambele reacţionează la evenimentul OK printr-o tranziţie în starea Prompting for Login. Întrucât nu s-a dorit utilizarea a două săgeţi identice, s-a introdus convenţia de a folosi o superstare.

Evenimente speciale

Compartimentul inferior al unei stări conţine perechi eveniment/acţiune. Evenimentele entry şi exit sunt standard dar, aşa cum se poate vedea din figura 1.2, se pot folosi şi alte evenimente, în funcţie de necesităţi. Dacă unul din aceste evenimente speciale apare în timp ce maşina cu stări finite se află în acea stare, atunci este invocată acţiunea corespunzătoare.

Figura 1.2 Diagramă de stare cu evenimente speciale

Înainte de apariţia UML, pentru a reprezenta un eveniment special se folosea o săgeată care forma un ciclu pe starea respectivă, aşa cum se poate vedea în figura 1.3. Însă, în UML, acest lucru are un alt înţeles. Atunci când se iese dintr-o stare, orice tranziţie invocă acţiunea corespunzătoare evenimetului exit (dacă există). Similar, când se intră într-o nouă stare, se invocă acţiunea ce corespunde evenimentului entry (dacă există). Deci, în UML, o tranziţie reflexivă precum cea din figura 3, invocă nu numai acţiunea myAction, ci şi acţiunile corespunzătoare evenimentelor exit şi entry.

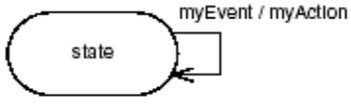


Figura 1.3 Tranziţie reflexivă

Super stări

Aşa cum se poate vedea în figura 1.1, super stările sunt folositoare atunci când mai multe stări, la apariţia aceluiaşi eveniment, răspund în acelaşi mod. Se poate trasa o super stare în jurul stărilor care se comportă

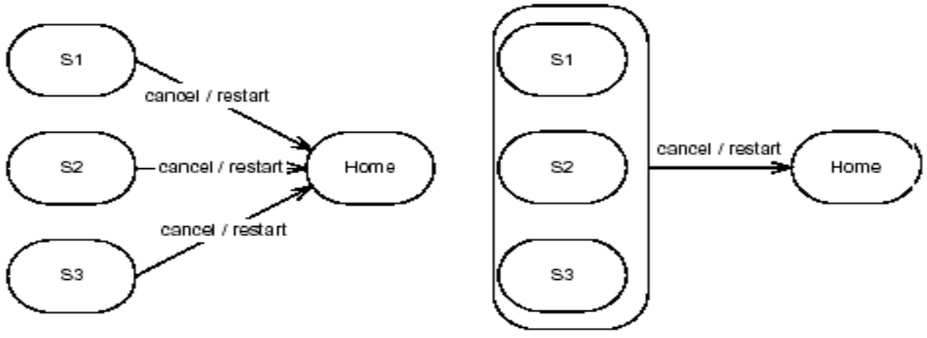


Figura 1.4

identic la apariţia unui anumit eveniment, putându-se înlocui astfel tranziţiile care pornesc de la fiecare stare cu o singură tranziţie care porneşte de la super stare. Aşadar, cele două diagrame din figura 1.4 sunt echivalente.

Tranziţiile de la super stări pot fi suprascrise prin trasarea tranziţiilor explicite de la substări. Astfel, în figura 1.5, tranziţia pause de la S3 suprascrie tranziţia implicită pause corespunzătoare super stării Cancelable. În acest sens, o super stare poate fi comparată cu o clasă de bază. Substările suprascriu (override) tranziţiile super stării în acelaşi mod în care clasele derivate pot suprascrie metodele clasei de bază corespunzătoare. Totuşi, relaţia dintre super stări şi substări nu este echivalentă cu moştenirea.

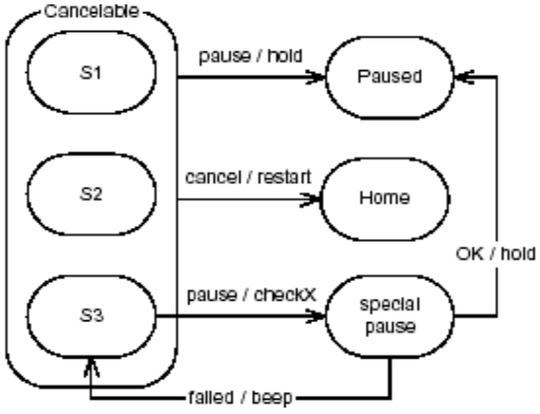


Figura 1.5 Suprascrierea tranziţiilor super stărilor

La fel cum stările obişnuite pot avea evenimente speciale, şi super stările pot deţine astfel de evenimente. În figura 1.6 este prezentată o maşină cu stări finite în care există evenimente exit şi entry atât în super stări cât şi în substări. Atunci când FSM-ul trece din starea Some State în starea Sub, mai întâi este invocată acţiunea enterSuper, urmată de acţiunea enterSub. În acelaşi mod, dacă are loc tranziţia din starea Sub2 în starea Some State, prima dată este invocată exitSub2 şi apoi exitSuper. Oricum, întrucât tranziţia e2 din starea Sub în starea Sub2 nu iese din super stare, se invocă doar acţiunile exitSub şi enterSub2.

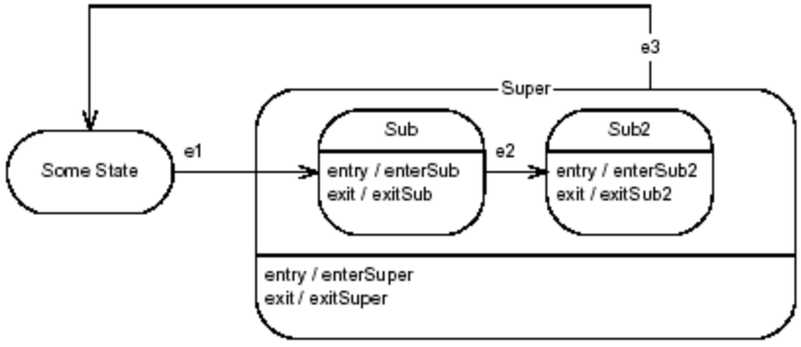


Figura 1.6 Invocarea ierarhică a acţiunilor corespunzătoare evenimentelor entry şi exit

Pseudo stări iniţiale şi finale

În figura 1.7 sunt prezentate două pseudo stări care sunt utilizate frecvent în UML. O maşină cu stări finite

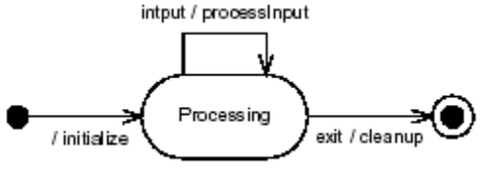


Figura 1.7 Pseudo stări: iniţială şi finală

îşi începe existenţa în *procesul de tranziţie* din pseudo starea iniţială. Tranziţia care porneşte din pseudo starea iniţială nu poate avea un eveniment, întrucât evenimentul este crearea maşinii cu stări finite. În schimb, ea poate avea o acţiune. Aceasta va fi prima acţiune invocată după crearea FSM-ului.

Similar, existenţa unei maşini cu stări finite se încheie în *procesul de tranziţie* în pseudo starea finală. De fapt, în această psudo stare finală nu se ajunge niciodată. Dacă trebuie realizată o acţiune în timpul tranziţiei către starea finală, atunci aceasta va fi ultima acţiune invocată de FSM.

Utilizarea diagramelor de stare

Diagramele de stare sunt extrem de folositoare pentru determinarea stărilor subsistemelor al căror comportament este destul de bine cunoscut. Pe de altă parte, majoritatea sistemelor care pot fi reprezentate prin diagrame de stare nu au comportamente ce pot fi cunoscute în avans. În plus, conduita celor mai multe sisteme se schimbă, evoluează în timp. Diagramele nu reprezintă o metodă favorabilă pentru sistemele care trebuie să se schimbe în mod frecvent. Problemele legate de aşezare şi spaţiu se manifestă chiar şi în cazul conţinutului diagramelor. Acest lucru îi împiedică uneori pe designeri să facă modificările necesare pentru un anumit proiect. Spectrul refacerii diagramei îi determină să nu mai adauge anunite stări necesare, obligându-i să folosească o soluţie particulară care să nu modifice “layout- ul” diagramei.

Pe de altă parte, textul reprezintă un mediu în care se pot face foarte uşor modificări. Problemele legate de aşezare aproape că nu există, existând întotdeauna spaţiu pentru adăugarea de noi linii. De aceea, pentru sistemele care evoluează în timp, se folosesc tabelele de stare (STTs – *State Transition Tables*). În figura 2.1 este prezentată diagrama de stare (STD) corespunzătoare unei uşi de metrou, care poate fi uşor reprezentată sub forma unui tabel de stare (STT), aşa cum se poate vedea în figura 2.2.

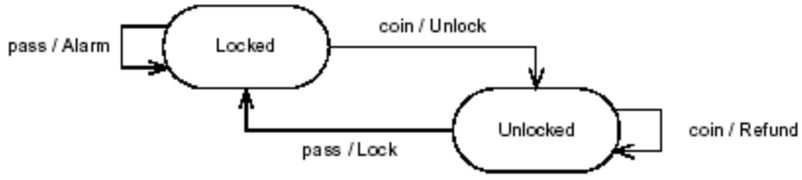


Figura 2.1 Diagrama de stare corespunzătoare unei uşi de metrou

Current State	Event	New State	Action
Locked	coin	Unlocked	Unlock
Locked	pass	Locked	Alarm
Unlocked	coin	Unlocked	Refund
Unlocked	pass	Locked	Lock

Figura 2.2 Tabelul de stare corespunzător unei uşi de metrou

Tabelul de stare conţine patru coloane. Fiecare rând al tabelului reprezintă o tranziţie, conţinând cele două capete ale săgeţii, evenimentul şi acţiunea corespunzătoare ce formează eticheta tranziţiei. Acest tabel se citeşte folosind următorul model: "Dacă sistemul se află în starea Locked şi apare un eveniment coin, atunci se trece în starea Unlocked şi se invocă funcţia Unlock.
Acest tabel poate fi uşor convertit într-un fişier text:

```
Locked    coin   Unlocked  Unlock
Locked    pass    Locked    Alarm
Unlocked  coin   Unlocked  Refund
Unlocked  pass    Locked    Lock
```

Aceste şaisprezece cuvinte conţin întreaga logică a acestei maşini cu stări finite. Există compilatoare care citesc acest fişier text şi generează cod care implementează logica respectivă.

Implementarea maşinilor cu stări finite

Există mai multe tehnici pentru a implementa o maşină cu stări finite. Una din cele mai folosite metode constă în folosirea mai multor instrucţiuni switch seriale (nested). În codul sursă 1 este prezentată implementarea maşinii cu stări finite din figura 2.1.

Cod sursă 1

```
class Turnstile{
```

```
final static int Locked = 0;
final static int Unlocked = 1;
final int Pass = 2;
final int Coin = 3;
int s = Unlocked;

public void Lock(){
System.out.println("se executa metoda Lock()" +
                   "\n" + "se trece in starea Locked");
}
public void Unlock(){
System.out.println("se executa metoda Unlock()" +
                   "\n" + "se trece in starea Unlocked");
}
public void Refund(){
System.out.println("se executa metoda Refund()" +
                                    "\n" + "se ramane in starea
Unlocked");
}
public void Alarm(){
System.out.println("se executa metoda Alarm()" +
                   "\n" + "se ramane in starea Locked");
}

void Transition(int e){
// static int s = Unlocked;
switch(s){
case Locked:
switch(e){
case Coin:
s = Unlocked;
Unlock();
break;
case Pass:
Alarm();
break;
}
break;
case Unlocked:
switch(e){
case Coin:
Refund();
break;
case Pass:
s = Locked;
Lock();
break;
}
break;
}
}

};

class MyApp{
public static void main(String[] args){
Turnstile stare = new Turnstile();
```

```
stare.Transition(stare.Pass);
stare.Transition(stare.Pass);
stare.Transition(stare.Coin);
stare.Transition(stare.Coin);
}
};
```

Ieşirea corespunzătoare acestui program este următoarea:

```
se execută metoda Lock( )
se trece în starea Locked
se execută metoda Alarm( )
se rămâne în starea Locked
se execută metoda Unlock( )
se trece în starea Unlocked
se execută metoda Refund( )
se rămâne în starea Unlocked
```

Deşi este destul de folosită, această metodă nu este cea mai eficientă. Pe măsură ce maşina cu stări finite implementată folosind această tehnică se dezvoltă, instrucţiunile switch seriale devin din ce în ce mai greu de folosit şi de citit de către programator. Codul poate ajunge foarte mare, multe porţiuni fiind identice.

O alternativă a acestei metode constă în folosirea şablonului de proiectare State.

Un şablon de proiectare denumeşte, abstractizează şi identifică aspectele cheie ale unei structuri comune de proiectare pe care o face utilă pentru crearea unui design orientat spre obiecte, reutilizabil. Şablonul de proiectare identifică clasele şi instanţele participante, rolurile şi colaborările lor, precum şi distribuţia responsabilităţilor. Fiecare şablon de proiectare se concentrează asupra unei anumite probleme de proiectare orientată spre obiecte. El descrie când se aplică, dacă poate fi aplicat în cazul unor alte restricţii de proiectare şi consecinţele şi compromisurile utilizării lui.

Şablonul de proiectare State (numit şi obiect pentru stări) permite unui obiect să-şi modifice comportamentul când starea sa internă se schimbă. Astfel, obiectul va părea că-şi schimbă clasa.

Şablonul State este folosit în oricare dintre următoarele cazuri:

- Comportarea unui obiect depinde de starea sa şi, la execuţie, obiectul trebuie să-şi modifice comportamentul în funcţie de respectiva stare.

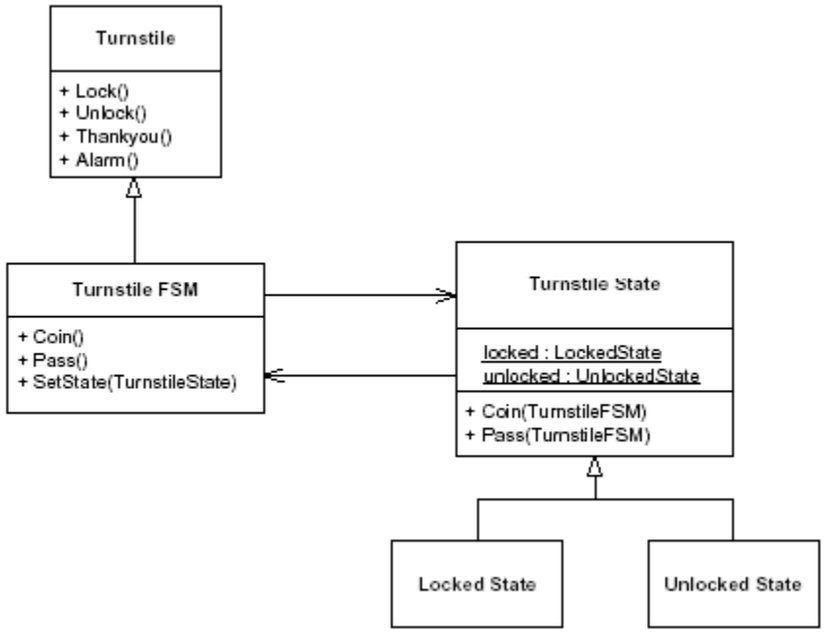


Figure 3.1 TurnstileFSM modelat cu şablonul State

- Operaţiile au instrucţiuni condiţionale de mari dimensiuni, cu multe părţi, care depind de starea obiectului. Această stare este de obicei reprezentată prin una sau mai multe constante enumerate. Deseori, mai multe operaţii vor conţine aceeaşi structură condiţională. Şablonul `State` plasează fiecare ramură a structurii condiţionale într-o clasă separată. Acest lucru va permite tratarea stării unui obiect ca un obiect cu drepturi depline, care poate varia independent de alte obiecte.

Folosind acest şablon de proiectare, maşina cu stări finite din figura 2.1 poate fi transpusă în diagramă de clase din figura 3.1.

Implementarea acestei diagrame este prezentată în codul sursă următor:

Codul sursă 2

```
class Turnstile{

public void Lock(){
System.out.println("se executa metoda Lock()" +
            "\n" + "se trece in starea Locked");
}
public void Unlock(){
System.out.println("se executa metoda Unlock()" +
            "\n" + "se trece in starea Unlocked");
}
public void Refund(){
System.out.println("se executa metoda Refund()" +
                  "\n" + "se ramane in starea Unlocked");
}
public void Alarm(){
System.out.println("se executa metoda Alarm()" +
```

```
                "\n" + "se ramane in starea Locked");
}
};


class TurnstileFSM extends Turnstile{
public TurnstileState itsState;

public void setState(TurnstileState s){
itsState = s;
}
public void Coin(){
itsState.Coin(this);
}
public void Pass(){
itsState.Pass(this);
}
};

abstract class TurnstileState{
public static LockedState lockedState;
public static UnlockedState unlockedState;

public abstract void Coin(TurnstileFSM t);
public abstract void Pass(TurnstileFSM t);
};

class LockedState extends TurnstileState{
public LockedState(){
}
public void Coin(TurnstileFSM t){
t.setState(unlockedState);
t.Unlock();
}
public void Pass(TurnstileFSM t){
t.Alarm();
}
};

class UnlockedState extends TurnstileState{
public UnlockedState(){
}
public void Coin(TurnstileFSM t){
t.Refund();
}
public void Pass(TurnstileFSM t){
t.setState(lockedState);
t.Lock();
}
};

class MyApp{
public static void main(String[] args){
TurnstileFSM turnstileFSM = new TurnstileFSM();
TurnstileState turnstileState = new UnlockedState();
turnstileState.lockedState = new LockedState();
turnstileState.unlockedState = new UnlockedState();
turnstileFSM.setState(turnstileState);
turnstileFSM.Pass();
turnstileFSM.Pass();
turnstileFSM.Coin();
turnstileFSM.Coin();
```

```
}
};
```

Ieşirea corespunzătoare acestui program este aceeaşi cu a celui din codul sursă 1.

Dacă starea iniţială este Unlocked, atunci TurnstileFSM, după apelul funcţiei setState( ), va conţine o referinţă către clasa UnlockedState, derivată din clasa TurnstileState. Dacă apare un eveniment Pass, se apelează metoda Pass( ) a clasei TurnstileFSM. Aceasta deleagă (apelează) metoda Pass( ) a clasei abstracte TurnstileState, care va apela metoda cu acelaşi nume din UnlockedState. Obiectul transmis ca parametru metodei Pass( ) este chiar obiectul curent (cel asupra căruia se va face modificarea). Metoda Pass( ) din UnlockedState va face operaţiile necesare asupra obiectului current.

Acest design are mai multe avantaje. În primul rând, comportamentul este separat de logica sistemului. Aceasta din urmă este conţinută în ierarhia de clase TurnstileState, în timp ce comportamentul este cuprins în ierarhia ce are la bază clasa Turnstile. Dacă se doreşte schimbarea logicii, dar păstrarea comportamentului, ori se schimbă clasa TurnstileState, ori se creează o nouă clasă derivată din aceasta. Pe de altă parte, dacă se doreşte modificarea comportamentului fără a modifica logica, se poate deriva clasa TurnstileFSM şi suprascrie metodele acesteia.

Concluzii

Maşinile cu stări finite reprezintă un concept destul de important pentru dezvoltarea de sisteme software. UML, prin diagramele de stare, oferă o modalitate eficientă de vizualizare a acestor FSM-uri. Cu toate acestea, pentru a dezvolta şi menţine un FSM, este mai convenabil să se folosească, în locul unei diagrame, un limbaj textual.

Exerciţii

1. Explicaţi rezultatele afişate de următorul program:

```
#include <iostream.h>
int g=0, x=0;
class C{
public:
void m(){x=500;   g=100;   }
void afisare(){ cout<<"x= "<<x<<" g= "<<g; }
private:
int x;
};

void m(){x=250; g=50;  }
void main(){
     C c;
     cout<<endl;

     c.m();
     c.afisare(); cout<<endl;

     m();
     c.afisare(); cout<< endl;

}
```

2. Explicaţi rezultatele afişate de următorul program:

```
#include <iostream.h>
void interschimb(float& i, float& j ) {
float t=i;
i=j;
j=t;
}
void interschimb(float& i, int& j ) {
float t=i;
i=j;
j=t;
}

int x=100;
float f1=1.0, f2=2.0;
void main(){
interschimb(f1,f2);
      cout<< endl<< " f1= "<<f1<<" f2= "<<f2;

      interschimb(f1,x);
      cout<< endl<< " f1= "<<f1<<" x= "<<x;

      interschimb(x,f2);
      cout<< endl<< " x= "<<x<<" f2= "<<f2;
}
```

3.

a. Supraincarcati operatorul de insertie << si explicaţi rezultatele afişate de următorul program:

```
#include <iostream.h>
#include <string.h>
class Persoana{
public:
Persoana(char* n){nume=n;};
void x(){
for (int i=0; nume[i]!= '\0';i++)nume[i]='x';
}
private:
char *nume;
};
void main(){
char *t="Tudor";
char *a="Andrei";
Persoana pt(t), pa(a);
cout<<pt<<endl<<pa<<endl;
pt=pa;
pt.x();
cout<<pt<<endl<<pa<<endl;
}
```

b. Supraincarcati operatorul de atribuire astefel încât acelaşi program să afişeze:

```
Tudor
Andrei
xxxxx
Andrei
```

4. Descrieţi diferenţa dintre legarea timpurie (statică) şi legarea târzie (dinamică) a metodelor

5. Explicati rezultatele afişate la executarea următorului program:

```
#include <iostream.h>
class S{
public:
static  S* create(){
      if(i==0){
refS=new S(); i++;
      }
      return refS;
}
void setName(char *s){name=s;}
char* getName(){return name;}
        static int geti(){return i;}
private:
static S* refS;
static int i;
S(){};
char *name;

};
int S::i=0;
```

```
S* S::refS;
void main(){
    S *o1, *o2;
    o1=S::create();
    o2=S::create();
    cout<<S::geti()<<endl;
    o1->setName("Matematica");
    o2->setName("Informatica");
    cout<<o1->getName()<<endl;
    cout<<o2->getName()<<endl;

}
```

6. Numere raţionale

Definiti clasa NumereRationale, ale carei obiecte sunt numere rationale. Definiţi metode şi operatori astfel încât executarea programului următor:

```
#include “rational.h”
void main(){
NumereRationale x(10, -20);
cout<< x << “ ; “ << 3 * x << “ ; “ << x  * 2 << “ ; “<< x * x<< endl;
}
```

să afişeze linia:

-1/2 ; -3/2 ; -1 ; 1/4

7. Persoana-Student

Fie urmatoarele 3 fisiere:

1) Fisierul persoana.h

```
class Persoana{
private:
char *nume;
public:
Persoana(char *n);
void afisare();
};
```

2) Fisierul student.h

```
class Student:public Persoana{
private:
char *universitate;
public:
Student(char *n, char *univ);
void afisare();
};
```

3) Fisierul aplicatie.cpp

```
void main(){
Persoana p("Tudor");
Student s("Tudor", "Universitatea din
Pitesti");
Persoana *ppers;
Student *pstud;
pstud=&p;
ppers=&p;
ppers->afisare();
ppers=&s;
ppers->afisare();
}
```

Se cere sa:

a) Modificati cele trei fisiere astfel incit nici unul sa nu aiba erori de compilare.

b) Construiti fisierele persoana.cpp si student.cpp pentru a implementa cele doua clase

c) Indicati ce fisiere trebuie incluse in proiect pentru ca executarea programului sa fie posibila

d) Explicati rezultatele afisate prin executarea programului

e) Modificati fisierele persoana.h si student .h astfel incat programul sa afiseze urmatoarele linii:

Tudor
Tudor, Universitatea din Pitesti

f) Explicati legarea statica si legarea dinamica (tarzie)
a metodelor unei clase.

g) Supradefiniti operatorul << astfel incat liniile de la punctul e)
sa fie obtinute prin executarea instructiunilor:
cout << p<<endl;
cout << s;

---

8. Explicati erorile semnalate la compilarea urmatorului program

```
class Punct{
public:
Punct(){x=0; y=0;}
Punct(int xx=0){x=xx;y=0;}
Punct(int xx, int yy=0){x=xx; y=yy;}
private:
int x,y;
};
```

```
void main(){
Punct p;
Punct p1(1);
Punct p2(1,2);
}
```

9. a Explicati erorile la executarea programului urmator
.b Modificati constructorul de copiere pentru a elimina aceste erori

```
#include <iostream.h>
class X{
public:
X(int i=0){p=new int; if(p) *p=i;}
X(const X &r){p=r.p;}
~X(){if(p){delete p; p=0;}}
void show(){cout<<*p<<endl;}
private:
int *p;
};
```

```
void main(){
X *o1,*o2;
o1=new X(1);
o2=new X(*o1);
o1->show(); delete o1;
o2->show(); delete o2;
}
```

10. a In programul urmator, supradefiniti operatorul << astfel incat cout<<i sa afiseze valoarea atributului i.x
b Precizati si explicati rezultatele afisate la executarea programului astfel obtinut

```
#include <iostream.h>
class C{
public:
  C(int i=0){x=i;}
  C& operator++(){++x; return *this;}
  C  operator--(){--x; return *this;}
private:
int x;
};
```

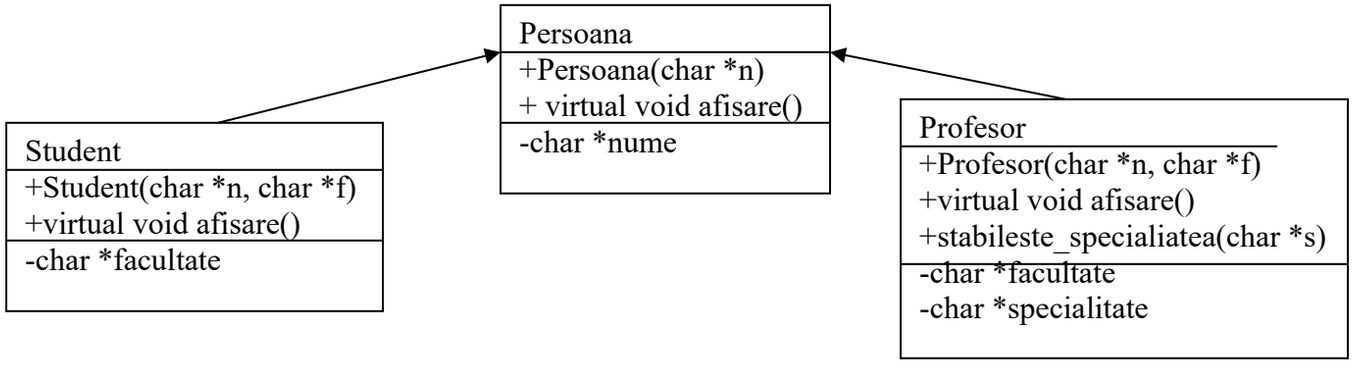
```
void main(){
  C i;
  cout<<i<<endl;
  cout<<++(++i)<<endl<<i<<endl;
  cout<<--(--i)<<endl<<i<<endl;
}
```

// 4.
11. a. Inlocuiti . . . in clasa Stack, astfel incat metodele push si pop sa asigure tratarea exceptiilor. Numarul maxim de elemente din vectorul supporteste dat de expresia suport.length.

b. Scrieti o aplicatie in care sa tratati exceptiile lansate de push si pop.

```
class Stack{
int varf;
        Object suport[];
        void push(Object x). . .{. . .}
        Object pop(). . .{. . .}
        void init(int s){
          varf=0;
          support=new Object[s];
        }
        Stack(int s){. . .}
}
```

12. Declarati si implementati clasele din urmatoarea diagrama UML (Unified Modeling Language)



13. Fie următorul program C++:

```
#include <iostream.h>
class A{
public:
void st(){cout<<"metoda A::st()"<<endl;}
virtual void vrt(){cout<<"metoda A::vrt()"<<endl;}
void stafis(){cout<<"metoda A::stafis()"<<endl; st(); vrt(); }
virtual void vrtafis(){
                    cout<<"metoda A::vrtafis()"<<endl; st(); vrt();
    }
};
class B: public A{
public:
void st(){cout<<"metoda B::st()"<<endl;}
virtual void vrt(){cout<<"metoda B::vrt()"<<endl;}
void stafis(){cout<<"metoda B::stafis()"<<endl; st(); vrt(); }
virtual void vrtafis(){
          cout<<"metoda B::vrtafis()"<<endl; st(); vrt();
    }
};
void main(){
A a, *p;
B b;

cout<<"Obiectul a"<<endl;
p=&a;
p->st();
p->vrt();
p->stafis();
p->vrtafis();

cout<<"Obiectul b"<<endl;
p=&b;
p->st();
p->vrt();
p->stafis();
p->vrtafis();
```

}

Ce se afişează prin executarea sa? Explicaţi fiecare linie afişată.
R.

Obiectul a	Obiectul b
metoda A::st();	metoda A::st();
metoda A::vrt();	metoda B::vrt();
metoda A::stafis();	metoda A::stafis();
metoda A::st();	metoda A::st();
metoda A::vrt();	metoda B::vrt();
metoda A::vrtafis();	metoda B::vrtafis();
metoda A::st();	metoda B::st();
metoda A::vrt();	metoda B::vrt();

14. Matrice

Fie urmatoarea schita a clasei Matrice,
ale carei obiecte sunt matrice de numere reale:

```
class Matrice{
private:
int m; // nr. linii
int n; // nr. coloane
float *p; // zona celor m*n elemente
// constructori
// operatori
// metode
};
```

Se cere:
sa completati definitia clasei si sa o implementati in asa fel incat urmatorul program sa produca efectele sugerate in comentarii:

```
 void main(){
Matrice a(2,3); //toate elementele nule
cin>>a;
// a va primi valoarea {1.1, 1.2, 1.3, 2.1, 2.2, 2.3}
Matrice b(a);
// b va fi initializat cu valoarea lui a
cin>>a;
// a = {0.11, 0.12, 0.13, 0.21, 0.22, 0.23}
cout<<a;
// afiseaza {0.11, 0.12, 0.13, 0.21, 0.22, 0.23}
cout<<endl;
cout<<b;
// afiseaza  {1.1, 1.2, 1.3, 2.1, 2.2, 2.3}
b=a;
a=0;   // toate elementele nule
cout<<a;
// afiseaza {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
cout<<b;
// afiseaza {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}
 }
```

15. Liste generice dublu inlantuite

Sa se completeze definitiile si sa se implementeze
urmatoarele clase, ale caror obiecte sunt
liste generice dublu inlantuite:

```
template <class T> class EDL{
private:
T info;
EDL *urmator;
EDL *anterior;
public:
EDL();
EDL(T c);
EDL<T> *da_urmator();
EDL<T> *da_anterior();
void da_info(T &c);
// operatorii <<, >>

};
template <class T> class LDL: public
EDL<T>{
private:
EDL<T> *inceput_lista, *sfarsit_lista;
public:
LDL();
void memo(T c);// introduce la inceputul listei
c
void scoate(T c);// scoate primul element c
void afisare();
};
```

```
void main(){
LDL <double> ld;
double c;
EDL<double> *p;
ld.memo(1.1);
ld.memo(2.2);
ld.memo(3.3);
ld.afisare();
ld.scoate(2.2);
ld.afisare();

}
```

16. Sir caractere

Fie urmatoarea schita a clasei CharString,
ale carei obiecte sunt siruri de caractere:

```
class CharString{
private:
char *p;
int lungime;
// constructori
// operatori
};
```

Se cere:
sa completati definitia clasei i sa o implementati in asa fel incat urmatorul program sa produca efectele sugerate in comentarii:

```
void main(){
CharString s("Examen"), t("POO");
CharStringr; // sir nul
r= " la ";
cout<<("Examen la " + t)<<endl; // afiseaza "Examen la POO"
cout<<(s+r+t)<< endl;// afiseaza "Examen la POO"
cout<<(s+" la POO")<<endl;// afiseaza "Examen la POO"
char *cs;
CharString a("Anul 2 ");
cs=a;
cout<<cs; // afiseaza "Anul 2 "
if(cs<=a) cout <<"cs <= a"<< endl; // afiseaza "cs <= a"
}
```

Utilizati functiile strcmp, strcat etc.

17. Completati specificarea clasei C si implementati metodele si operatorii astfel incat prin executarea programului sa se obtina rezultatele indicate prin comentarii.

```
#include <iostream.h>
class C{
public:
void set_x(int x){
     *px=x;
   }
void set_y(int y){
     *py=y;
   }
private:
int *px;
int *py;
};
```

```
void main(){
C m, n(1,2);
cout<<m<<endl;        // x=0; y=0;
cout<<n<<endl;        // x=1; y=2;
m=n*100;
cout<<n*100<<endl; // x=100;y=200;
cout<<n<<endl;        //x=1; y=2;
m=n;
n.set_x(100);n.set_y(200);
cout<<m<<endl;        //x=100; y=2
}
```

R.

Pentru rezolvare vor fi luate în considerare următoarele elemente:

1. constructor cu valori implicite pentru parametri
2. supraincarcare operator de insertie
3. supraincarcare operator * cu rezultat intors prin valoare si fara modificarea obiectului curent
4. supraincarcare operator= prin referinta pentru px si prin valoare pentru py.

```
#include <iostream.h>
class C{
public:
C(int x=0, int y=0){ // 1
            px=new int; py=new int; *px=x; *py=y;
      }
void set_x(int x){*px=x;}
void set_y(int y){*py=y;}
C& operator=(C&);     // 4
C  operator*(int x);    // 3
private:
int *px;
int *py;
friend ostream& operator<<(ostream&, C&); // 2
};

void main(){
C m, n(1,2);
cout<<m<<endl;        // x=0; y=0;
cout<<n<<endl;     //x=1; y=2;
m=n*100;
cout<<n*100<<endl; // x=100; y=200;
cout<<n<<endl;     //x=1; y=2;
m=n;
n.set_x(100);n.set_y(200);
cout<<m<<endl;          //x=100; y=2
}

ostream& operator<<(ostream& o, C& z){ // 2
```

```
o<< *(z.px)<<";";
o<< *(z.py)<<";";
return o;
}
C& C::operator=(C& c){ // 4
px=c.px;
*py=*(c.py);
return *this;

}
C  C::operator*(int i){return C(*px*i,*py*i);} // 3
```

18. Indicaţi ce rezultate afişează programul următor.
Urmăriţi conversiile explicite din expresii. Justificaţi răspunsul explicând mecanismul de legare statică şi de legare dinamică.

```
#include <iostream.h>
#include <conio.h>
class B{
public:
void st(){
     cout<< "B::st()"<<endl;
   }
virtual void v(){
     cout<< "B::v()"<<endl;
   }
};
class D: public B{
public:
void st(){
     cout<< "D::st()"<<endl;
   }
virtual void v(){
     cout<< "D::v()"<<endl;
   }
};
```

```
void main(){
   B b, *pb;
D d, *pd;
pb=&b;
pd=&d;

   clrscr();
cout<<"1"<<endl;
pb->st();
pb->v();

cout<< "2"<<endl;
((D*)pb)->v();
  ((D*)pb)->st();
((B*)pd)->v();
((B*)pd)->st();

cout<<"3"<<endl;
pb=&d;
pb->st();
pb->v();
((D*)pb)->v();
((D*)pb)->st();

cout<<"4"<<endl;
d.B::st();
d.B::v();
}
```

R.

1

B::st()	legare statică, tipul expresiei este B
B::v()	legare dinamică

2

B::v()	legare dinamică, conversia ignorată, tipul obiectului este B
D::st()	conversie la D*, apoi legare statică (tipul expresiei este D)
D::v()	legare dinamică, tipul obiectului current este D
B::st()	conversie la B*, apoi legare statică (tipul expresiei este B)

3

B::st() legare statică, tipul expresiei este B
D::v() legare dinamică, tipul obiectului current este D
D::v() legare dinamică, tipul obiectului current este D
D::st() conversie la D*, apoi legare statică (tipul expresiei este D)

4.
B::st() invocare explicită a metodei moştenite
B::v() invocare explicită a metodei moştenite

Se consideră clasa următoare:

```
class C{
public:
C(int n=1, int v[]);
 void set(int i, int val){
   pi[i]=val;
 }
// alte metode si operatori
private:
int dim;
int *pi;
};
```

şi programul:

```
void main(){
int a[]={1,2,3}, b[]={10,20};
C x(3,a),y(2, b);
// prima afisare
cout<<x<<endl; //1; 2; 3;
a[0]=10;

// a doua afisare
cout<<x<<endl; //1; 2; 3;

// a treia afisare
cout<<(x=y)<<endl;
y.set(0,1000);

// a patra afisare
cout<<x<<endl;

cout<<(x=x)<<endl;
}
```

1. Implementaţi operatorul de inserţie, astfel încat // prima afisare să producă rezultatele din comentariu.
2. Implementaţi constructorul astfel încât // a doua afisare să producă rezultatele din comentariu. Discutaţi semantica prin referinţă şi semantica prin valoare.
3. Presupunem că operatorul de atribuire din // a treia afisare este cel predefinit. Precizaţi ce se afişează la // a treia afisare şi la // a patra afisare. Explicaţi rolul operatorului de atribuire la obţinerea acestor rezultate.
4. Fie următoarea schiţă de supraîncărcare a operatorului de atribuire:

   ```
   C& C::operator=(C& x){
         delete[] pi;
         // completari
   }
   ```

   Completaţi implementarea operatorului de atribuire astfel încât // a treia afisare şi // a patra afisare să producă aceleaşi rezultate.

5. Modificaţi implementarea de la punctual 3 astfel încât expresiile cout<<x şi cout<<(x=x) sa aibă acelaşi efect (afişarea valorilor lui x)

R.

1. Se declară operatorul funcţie prietenă a clasei C

```
friend  ostream& operator<<(ostream &o, const C&);
```

şi se implementeză:

```
ostream& operator<<(ostream &o, const C &m){
   for(int j=0; j<m.dim; j++)o<< m.pi[j]<<" ";
   return o;
}
```

2. Se implementează constructorul cu semantica prin valoare: se face o copie a vectorului v:

```
C::C(int n, int v[]){
dim=n;
pi= new int[dim];
          for(int j=0; j<dim; j++)pi[j]=v[j];
}
```

Un constructor cu semantica prin referinţă (pi=v în locul ultimelor două instrucţiuni) ar face ca // a doua afisare să producă rezultatele 10; 2; 3; deoarece vectorul v devine resursă partajată a obiectului x şi a funcţiei main().

3. Datorită semanticii prin referinţă a operatorului de atribuire predefinit, după atribuire vectorul b devine resursă partajată a obiectelor x şi y. Prin urmare // a treia afisare produce 1; 2; iar // a patra afisare 1000; 2;
4. Se supraîncarcă operatorul de atribuire cu semantică prin valoare; schiţa sugerată în enunţ eliberează resursele obiectului destinaţie.

```
C& C::operator=(C& x){
 delete[] pi;
      dim=x.dim;
pi=new int[dim];
for(int j=0; j<dim; j++)pi[j]=x.pi[j];
 return *this;
}
```

5. Se verifică dacă este sau nu vorba de o autoatribuire (this==&x). Dacă da, se execută doar return *this.

17. Fie urmatorul program Java:

```
// Model View Controller

import java.awt.*;
import java.awt.event.*;
class Model{
private int x=0;
public Model(){};
public void increment(){x++;}
	public void decrement(){x--;}
public int get_x(){return x;}
}
public class View extends Frame{
protected Button binc;
	protected Button bdec;
protected Model m;
private Controller c;
protected TextField tf;
public static void main(String args[]){
Frame f= new View();
}
public View(){
setTitle("Exemplu Model-View-Controller");

binc= new Button("A");
add("North",binc);

bdec= new Button("B");
add("South",bdec);

m=new Model();

c=new Controller(this);
binc.addActionListener(c);
bdec.addActionListener(c);


tf=new TextField(10);
add("Center",tf);

setSize(100,250);
setVisible(true);
}
}

class Controller implements ActionListener{
private View vw;
public Controller(View v){
vw=v;
}
public void actionPerformed(ActionEvent e){
		Button source=(Button)e.getSource();
if (source==vw.binc) vw.m.increment();
		else if(source==vw.bdec) vw.m.decrement();
```

```
            vw.tf.setText(String.valueOf(vw.m.get_x()));
}
}
```

Se cere:

a) Descrieti dispunerea in fereastra a componentelor si efectul actionarii butoanelor A si B.
b) Descrieti procedeul Model-View-Controller
c) Modificati programul,folosind interfata WindowListener si metoda sa public void windowClosing(WindowEvent e), astfel incat actionarea butonului x din coltul din dreapta sus sa inchida fereastra.

R.

```
// Model View Controller
import java.awt.*;
import java.awt.event.*;
class Model{ …
}
public class View extends Frame implements WindowListener{
…
      public void windowClosed(WindowEvent e){}
      public void windowOpened(WindowEvent e){}
      public void windowDeiconified(WindowEvent e){}
      public void windowIconified(WindowEvent e){}
      public void windowActivated(WindowEvent e){}
      public void windowDeactivated(WindowEvent e){}
      public void windowClosing(WindowEvent e){
            System.exit(0);
      }
}

class Controller implements ActionListener{ …
}
```

18. Fie următorul program C++:

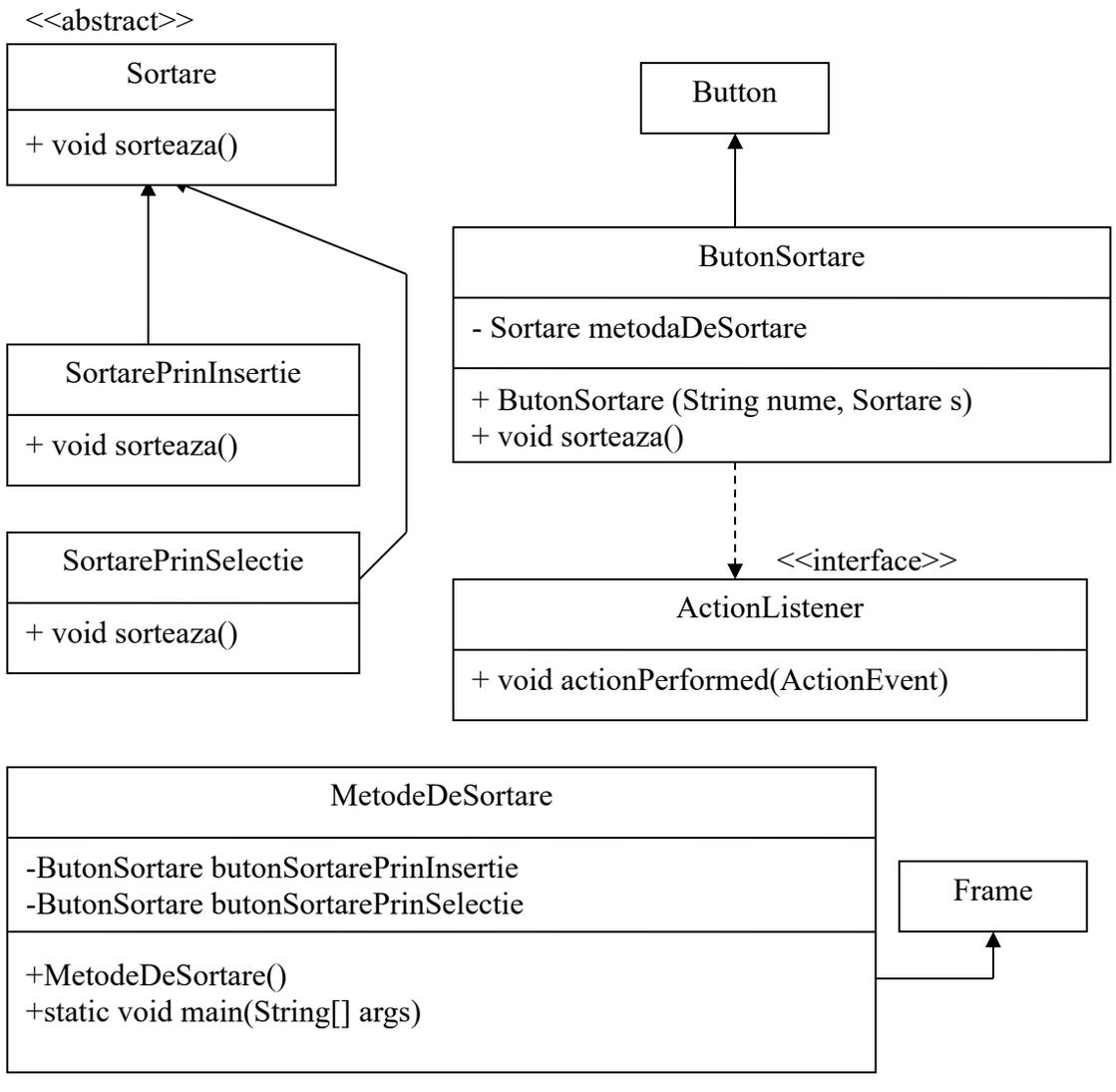
```
#include <iostream.h>
class A{
public:
void st(){cout<<"metoda A::st()"<<endl;}
virtual void vrt(){cout<<"metoda A::vrt()"<<endl;}
void stafis(){cout<<"metoda A::stafis()"<<endl; st(); vrt(); }
virtual void vrtafis(){cout<<"metoda A::vrtafis()"<<endl; st(); vrt(); }
};
class B: public A{
public:
void st(){cout<<"metoda B::st()"<<endl;}
virtual void vrt(){cout<<"metoda B::vrt()"<<endl;}
void stafis(){cout<<"metoda B::stafis()"<<endl; st(); vrt(); }
virtual void vrtafis(){cout<<"metoda B::vrtafis()"<<endl; st(); vrt(); }
};
void main(){
A a, *p;
B b;

cout<<"Obiectul a"<<endl;
p=&a;
```

```
p->st();
p->vrt();
p->stafis();
p->vrtafis();

cout<<"Obiectul a"<<endl;
p=&b;
p->st();
p->vrt();
p->stafis();
p->vrtafis();

}
```

Ce se afişează prin executarea sa? Explicaţi fiecare linie afişată.

Fie următoarea diagramă de clase UML, în care butoanele dintr-o fereastră activează o metoda de sortare asociată.

1. Implementati acest proiect în limbajul Java. Nu se cere implementarea algoritmilor de sortare (prin inserţie sau prin selecţie), prin activarea butoanelor vor fi doar afişate pe monitor mesajele **Sortare prin insertie** sau **Sortare prin selectie.**
2. Desenati diagrama de colaborare între butonul *butonSortarePrinInsertie*
3. şi obiectul atribut al său *metodaDeSortare*
4. Explicaţi rolul mecanismului de legare dinamică a metodelor şi a conceptului de clasă abstractă în acest proiect.



R.

```
import java.awt.*;
import java.awt.event.*;
abstract class Sortare{
public abstract void sorteaza();
}

class SortarePrinInsertie extends Sortare{
public void sorteaza(){System.out.println("Sortare prin insertie");}
}

class SortarePrinSelectie extends Sortare{
public void sorteaza(){System.out.println("Sortare prin selectie");}
}

class ButonSortare extends Button implements ActionListener{
public ButonSortare(String nume,Sortare s){super(nume);
metodaDeSortare=s;}
```

```
private Sortare metodaDeSortare;

public void actionPerformed(ActionEvent e){metodaDeSortare.sorteaza();}
}

public class MetodeDeSortare extends Frame{
private ButonSortare butonSortarePrinInsertie;
private ButonSortare butonSortarePrinSelectie;
public MetodeDeSortare(){

butonSortarePrinInsertie=
new ButonSortare("Insertie",new SortarePrinInsertie());
butonSortarePrinInsertie.addActionListener(butonSortarePrinInsertie);

butonSortarePrinSelectie=
new ButonSortare("Selectie",new SortarePrinSelectie());
butonSortarePrinSelectie.addActionListener(butonSortarePrinSelectie);

add("North",butonSortarePrinInsertie);
add("South",butonSortarePrinSelectie);
show();
}
public static void main(String[] arg){
MetodeDeSortare m=new MetodeDeSortare();
}
}
```

Bibliografie.

1. Herbert Schildt: *C++, manual complet*. Teora, 1997
2. Octavian Catrina, Iuliana Cojocaru: *Turbo C++,* Teora, 1993
3. Tudor Bălănescu: *Metodologii avansate de programare orientată pe obiecte*, Editura Fundaţiei România de Mâine (în curs de apariţie)
4. H.M. Deitel, P.J. Deitel: *C++, How to program,* Prentice Hall, 1998
5. Ipate, Florentin Eugen *Modelare Orientată pe Obiecte cu UML*, Editura Universitţii din Piteşti, 2001
6. Martin, Robert Cecil *UML for Java Programmers*, Prentice-Hall, 2002
7. John D. McGregor, David A. Sykes: A practical guide to testing object-oriented software, Addison- Wesley, 2001.
8. Hans- Gerhard Gross, Component- Based Software testing with UML, Springer-Verlag, 2005