

*Programare orientata-
obiect (POO)
utilizand C++*

Gheorghe Grigoras
Dorel Lucanu

Bibliografie

- **H. Schildt**: C++ manual complet, Teora, 2000
- **D. Kaler, M.J. Tobler, J. Valter**: C++, Teora, 2000
- **Bjarne Stroustrup**: The C++ Programming Language, Addison-Wesley, 3rd edition, 1997
- **Martin Fowler**. UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition). Addison-Wesley Professional, 3 edition, 2003

Manuale electronice

- **Bruce Eckel** : Thinking in C++, 2nd Edition
- ******* : Online C++ tutorial
- Donald Bell. UML basics: An introduction to the Unified Modeling Language
- SGI Standard Template Library Programmer's Guide

Organizare

- pagina curs:
 - ! se vor face (cel putin) saptamanal schimbari
 - <http://portal.info.uaic.ro/curs/poo/default.aspx>
- laboratoarele
 - prima parte - 3 teme individuale
 - a doua parte – 1 proiect individual
- examinare
 - pe parcurs: 1 test (sapt. 16)
 - 50% din nota finala
 - laborator – 2 evaluari a temelor (sapt. 8, 15)
 - 50% din nota finala
- conditii de promovabilitate
 - 50% din punctajul total maxim
 - 40% din punctajul maxim la fiecare proba (lab., test)

Organizare cursuri

- 19.02 Clase in C++
- 26.02 Parametrizare + STL(I)
- 05.03 Ierarhii in C++
- 12.03 Polimorfism
- 19.03 Principii OO (I)
- 26.03 Principii OO (II)
- 02.04 Proiectare (I)
- 08.04-13.04 Verificare
- 16.04 Proiectare (II)
- 23.04 Exceptii
- 30.04 Proiectare (III)
- 07.05 STL (II) (recuperare)
- 14.05 Proiectare (IV)
- 21.05 Programare avansata in C++
- 28.05 Modelare
- 03-08 Test scris

Teme laborator

- Tema 1
 - accent pe modelare (organizarea claselor, dinamica obiectelor, descrierea acestora in C++)
- Tema 2
 - accent pe proiectare (sabloane)

Curs 1

- Primul pas in POO
 - principii de baza
 - concepte
 - Clase si obiecte
 - attribute, metode, stari
 - declararea claselor si obiectelor in C++
 - utilizarea obiectelor in C++
 - constructori, destructori
 - exemple
 - dezvoltarea unei aplicatii utilizand patternul model-view-controller (cu interfata text)

Principii de baza ale POO

- **abstractizare**

- pastrarea aspectelor importante (generale) si ignorarea detaliilor nesemnificative (specifice)

- **incapsulare**

- ascunderea implementarii fata de client; clientul depinde doar interfata

- **modularizare**

- impartirea unui sistem complex in parti (module) manevrabile

- **ierarhizare**

- clasificarea pe nivele de abstractizare

- **polimorfism**

- aceeaasi forma sintactica poate avea diferite intelesuri in functie de contextul de utilizare

Concepte POO

obiecte

clase

attribute

metode

mesaje

supraincarcare

suprascriere

legare dinamica

interfata

relatii dintre clase

generalizare

specializare

mostenire

compozitie

asociere

pachet

subsistem

modelare

Obiecte - informal

- un obiect reprezinta o entitate care poate fi fizica, conceptuala sau software
 - entitate fizica
 - tren, angajat CFR (?)
 - entitate conceptuala
 - proces chimic, rationament matematic
 - entitate software
 - structura de date, program

Obiecte – definitie mai formala

- un **obiect** este o abstractie, concept sau orice alt lucru ce are un inteles strans legat de domeniul problemei
- un **obiect** este caracterizat de:
 - **identitate**
 - nume, ...
 - **stare**
 - exprimata prin **attribute**
 - valorile atributelor la un moment dat definesc o stare
 - **comportarea**
 - data de multimea **metode** (servicii, operatii publice)

Un obiect are identitate

- fiecare obiect are identitate unica, chiar daca starea sa este identica cu a altui obiect



Un obiect are stare

- starea unui obiect este una dintre conditiile in care acesta poate exista
- starea se poate schimba de-a lungul timpului
- este data de valorile atributelor + legaturi (instante ale relatiilor)

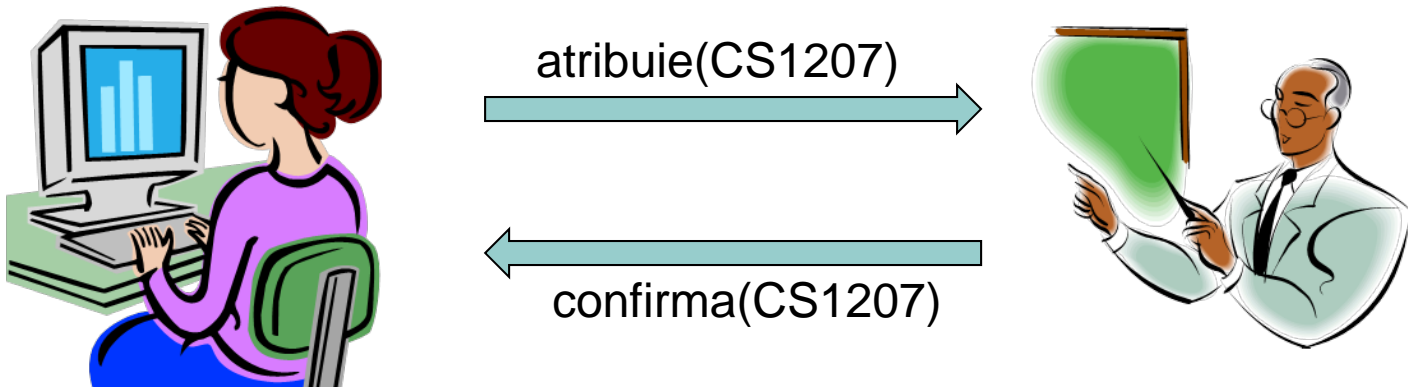


nume
nr. marca
functia
status

Ion Ion
2143576
conferentiar
preda POO

Un obiect are comportament

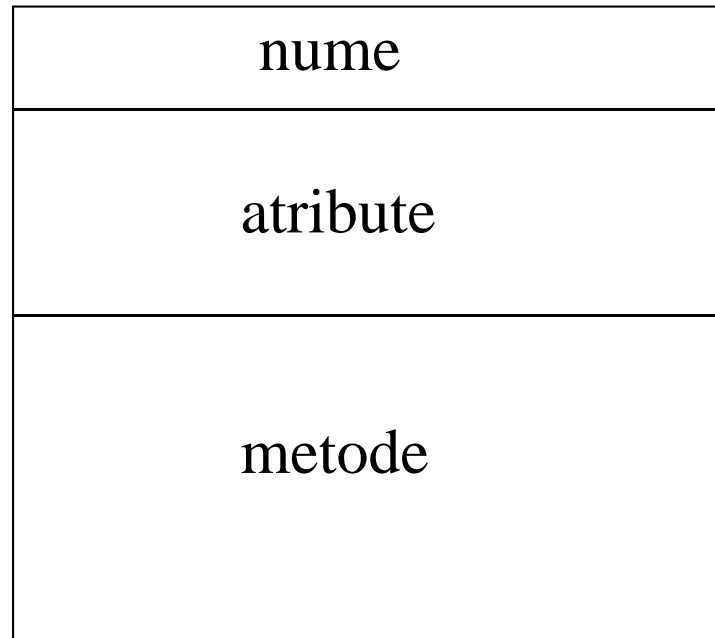
- determina cum un obiect actioneaza si reactioneaza la cererile altor obiecte



Clase

- O **clasa** descrie unul sau mai multe obiecte ce pot fi precizate printr-un set uniform de atribute si metode.
 - un obiect este o **instanta** a clasei
- o clasa este o **abstractizare**
 - pastreaza trasaturile (atribute, metode) esentiale
 - ignora ce nu este esential

Clase - reprezentare grafica



Clasa Cont (versiunea 01)

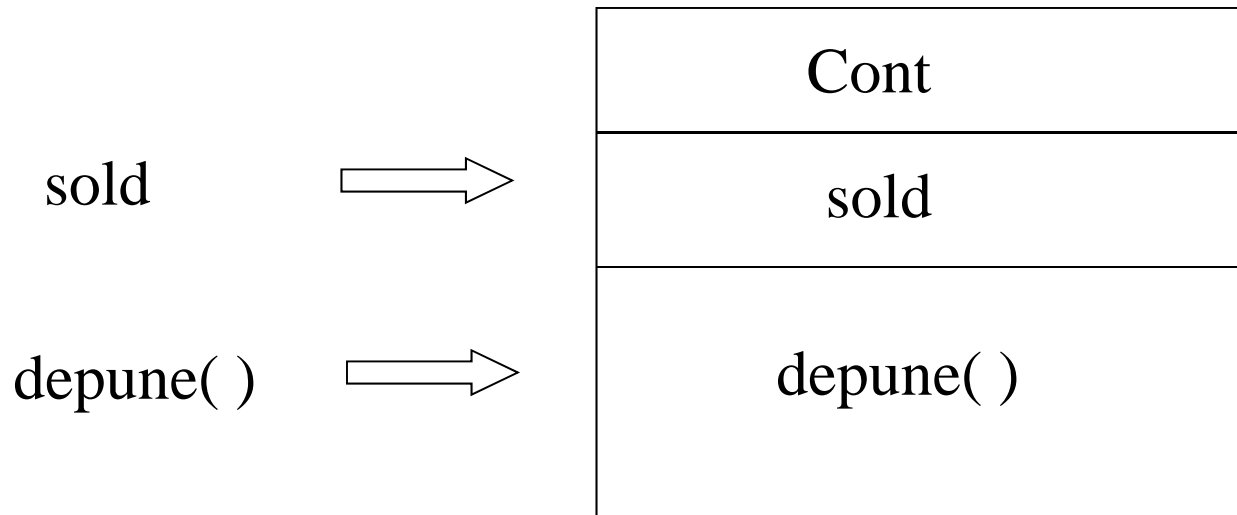
- Specificarea clasei
 - Un cont bancar:
 - are un titular, sold, o rata a dobinzii, numar de cont si
 - se pot efectua operatii de depunere, extragere, actualizare sold
- Extragerea atributelor:
 - titular, sold, rata a dobinzii, numar de cont
- Extragerea metodelor
 - depunere, extragere, actualizare sold
- Completarea specificatiei (analiza)
 - actualizare sold ⇒ data ultimei operatii

Clase (continuare)

- o clasa joaca roluri diferite in cadrul unei aplicatii software
 - **furnizor** de servicii pentru clasele client
 - **client** pentru clasele ale caror servicii sunt necesare in descrierea propriei comportari
- definitia unei clase presupune:
 - combinarea datelor cu operatii
 - ascunderea informatiei
 - clasele client nu acces la implementarea metodelor

Combinarea datelor cu operatiile asupra lor

- datele si operatiile asupra lor sint incluse in aceeaasi unitate sintactica
 - datele si procesele devin legate



⇒ C++: clase, structuri

Ascunderea informatiei

- modul de structurare a datelor nu este cunoscut
- actiunile asupra datelor sint realizate numai prin operatiile (metodele) clasei
- si anumite actiuni pot fi ascunse: e.g., validare a contului
- clientul clasei Cont:

~~`cont.sold += suma`~~

`cont.depune (suma)`

OK!

Incapsulare

Incapsulare = Combinare + Ascundere

Combinarea: avantaje

- definește clar ce structuri de date sunt manevrate și care sunt operațiile legale asupra lor
- adaugă programului modularitate
- scade riscul ca datele să fie alterate de programele "slabe"
- facilitează ascunderea informației

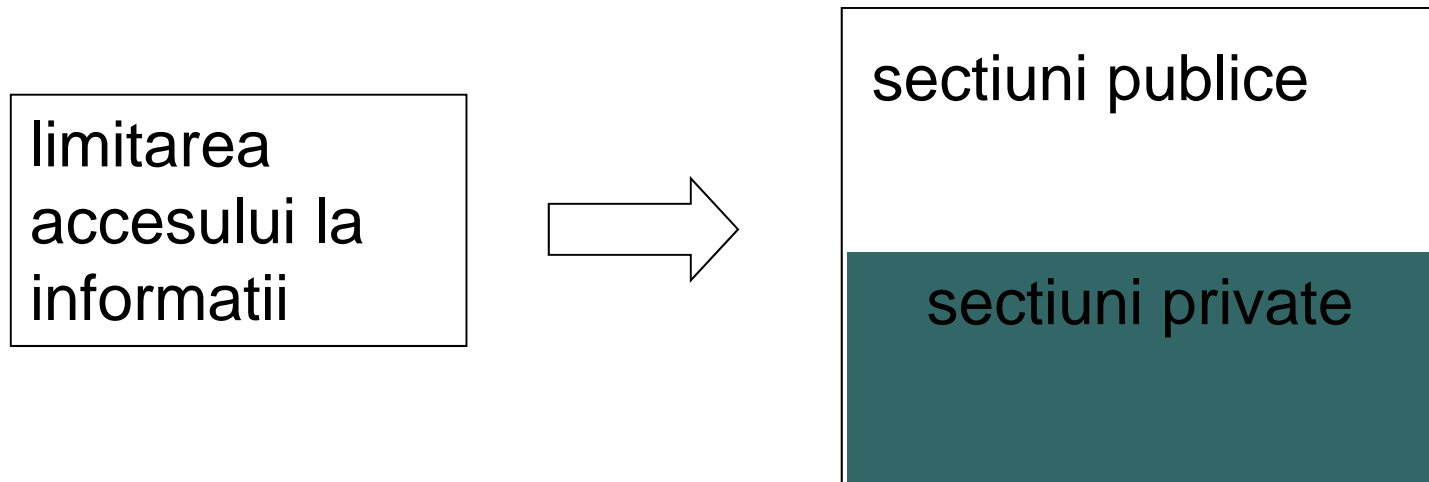
Ascunderea informatiei: avantaje

- programe mai sigure si fiabile
- elibereaza clasele client de grija cum sint manevrate datele
- previne aparitia erorilor produse de clasele client ce manevreaza datele utizind cod "slab"

Incapsulare (Combinare + Ascundere): avantaje

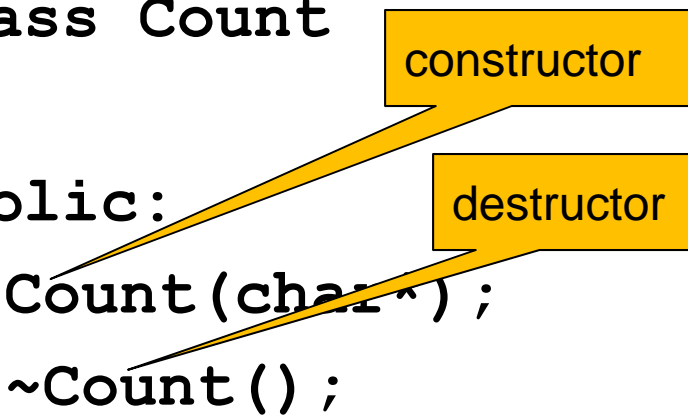
- combinare + ascunderea informatiei = protejarea datelor
- previne aparitia erorilor prin limitarea accesului la date
- asigura portabilitatea programelor
- faciliteaza utilizarea exceptiilor
- interfata unei clase = operatiile cu care o clasa utilizator poate manevra datele

Structurarea nivelului de acces la informatie



Count.h

```
class Count
{
public:
    Count(char*);
    ~Count();
    void
    deposit(double);
    void draw(double);
    double balance();
```



```
private:
    string *owner;
    string countNo;
    double sold;
    assignCountNo();
    releaseCountNo();
};
```

Count.cpp

```
Count::Count(char
*s)
{
    owner =
        new string(s);
    sold = 0;
    assignCountNo();
}
```

create object nou

```
Count::~~Count()
{
    delete owner;
    releaseCountNo();
}
```

distrugere obiect

```
void Count::deposit(double
ammount)
```

```
{
    sold += ammount;
}
```

```
double Count::balance()
```

```
{
    return sold;
}
```

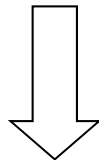
. . .

Demo.cpp

```
void main()  
{  
    Count count("Ionescu");  
    count.deposit(2000);  
}
```

main() joaca rol de client pentru Count

```
count.sold += 5000;
```



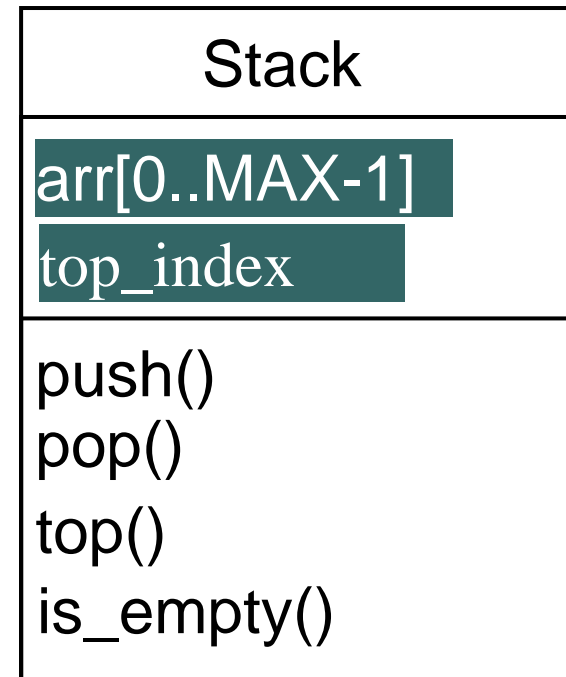
error C2248: 'sold' : cannot access private member

Tipuri de date abstracte si obiecte

- **tip de data abstract** = o descriere a unui tip de data independent de reprezentarea datelor si implementarea operatiilor
- **O clasa este o implementare a unui tip de date abstract.** Ea defineste attribute si metode care implementeaza structura de date respectiv operatiile tipului de date abstract.

Stiva

- tipul de data abstract Stiva
 - ❑ entitati de tip data: liste LIFO
 - ❑ operatii
 - ⇒ push()
 - ⇒ pop()
 - ⇒ top()
 - ⇒ is_empty()



Stiva.h

```
class Stack
{
public:
    Stack();
    ~Stack();
    void push(char);
    void pop();
    char top();
    bool is_empty();
private:
    char arr[MAX_STACK];
    int top_index;
};
```

Stiva.cpp

```
void Stack::push(char c)
{
    if (top_index == MAX_STACK-1)
        throw "Depasire superioara.";
    arr[++top_index] = c;
}

char Stack::top()
{
    if (top_index < 0)
        throw "Depasire inferioara.";
    return arr[top_index];
}
```

Stiva_demo

```
Stack s;  
char c = 'a';  
  
try {  
    while (true) {  
        s.push(c++);  
        cout << s.top() << endl;  
    }  
}  
catch (char *mes_err) {  
    cout << mes_err << endl;  
}
```



a
b
c
d
e
f
g
h
i
j
Depasire superioara.
Press any key to ...

Utilizarea de clase

- exista multe biblioteci de clase
 - STL
 - MFC
 - etc
- pentru utilizare, trebuie cunoscuta doar interfata (elementele publice)
- nu intereseaza implementarea
- programul care utilizeaza clasa este independent de implementarea clasei
- Exemple:
 - string
 - iostream

clasa string

- pentru a utiliza clasa **string** trebuie inclus fisierul antet (header):

```
#include <string>
```

- clasa string face parte din biblioteca STL (Standard template Library)
- aceasta biblioteca defineste toate numele in spatiul de nume std, pentru evitarea de conflicte
- pentru inceput e bine sa includeti declaratia:

```
using namespace std;
```

clasa string – exemple de constructori

- **string ()**
- creeaza sirul vid ("")
string s1;
- **string (other_string)**
- creeaza un sir identic cu alt sir
string s2("Curs POO");
string s3(s2);
- **string (count, character)**
- creeaza un sir ce contine un caracter repetat de un numar de ori
string s4(8, '=');

clasa string – ex. functii membre constante

nu modifica sirul

- **const char * c_str()**
 - intoarce un sir reprezentat ca in C egal cu continutul obiectului curent
- **unsigned int length()**
 - intoarce lungimea sirului
-
- **unsigned int size()**
 - la fel ca **length**
-
- **bool empty()**
 - inraorec **true** daca sirul este vid, **false** altfel

clasa string – exemple functii membre

- **void swap (other_string)**
- interschimba continuturile a doua siruri
- **string & append (other_string)**
- adauga sirul parametru la sfarsit
- **string & insert (position, other_string)**
- insereaza sirul parametru la pozitia data
- **string & erase (position, count)**
- elimina un subsir de lungime data de la o pozitie data
- **unsigned int find(other_string, position)**
- intoarce pozitia primei aparitii a subsirului dat ca parametru; cautarea incepe de la pozitia data ca param.

clasa string - aplicatie

- separarea unui sir in cuvinte

```
string s("Acesta este un sir demonstrativ  
        pentru cursul de POO.");  
while (s[0] == ' ') s.erase(0,1);  
int p;  
while (! s.empty()) {  
    p = s.find(" ");  
    if (p < 0) p = s.length();  
    cout << s.substr(0, p) << endl;  
    s.erase(0, p);  
    while (s[0] == ' ') s.erase(0,1);  
}
```

Acesta
este
un
sir
demonstrativ
pentru
cursul
de
POO.

Intrari/iesiri

- Flux (stream) = tip de date ce descrie la nivel abstract fisiere “destepte”
- C++ : pachetul de clase `iostream` este responsabil cu fluxurile de intrare/iesire
 - Fluxuri de intrare: `istream`
 - Operatorul `>>`
`input_stream >> l_value_expr`
 - Obiectul `cin` modeleaza fis. std. de intrare
 - Fluxuri de iesire: `ostream`
 - Operatorul `<<`
`output_stream << r_value_expr`
 - Obiectul `cout` fisierul standard de iesire

Intrari/iesiri

- introducerea unui numar

```
int nr;  
cout << "Numar>";  
cin >> nr;
```

- introducerea unui sir C

```
string s;  
cout << "Sir>";  
cin >> s;
```

- prototipul operatorilor

```
istream& operator>>(int& n)  
istream& operator>>(char* s)  
...
```


Intrari/iesiri (continua)

- Declararea fisierelor ca fluxuri (ex. c2cpp10.cpp)

```
#include <fstream>
```

```
ifstream f_inp("c2cpp10.cpp");
```

```
ofstream f_out("c2cpp10.cp1");
```

ifstream \equiv **input file stream**

ofstream \equiv **output file stream**

Copierea unui flux – varianta 1

```
if (f_out && f_inp)
    while (f_inp >> c)
        f_out << c;
```

- nu-i prea OK

c2cpp10.cpp	c2cpp10.cp1
<pre>#include <iostream.h> #include <fstream.h> int main(void) { // copierea unui ...</pre>	<pre>#include<iostream.h>#include<f stream.h>intmain(void){//copie reaunuifisier ...</pre>

Copierea unui flux – varianta 2

```
ifstream f_inp("c2cpp10.cpp");  
ofstream f_out("c2cpp10.cp2");  
if (f_out && f_inp)  
    while (f_inp.get(c))  
        f_out << c;
```

⇒ mai fidela

c2cpp10.cpp	c2cpp10.cp2
<pre>#include <iostream.h> #include <fstream.h> int main(void) { // copierea unui ...</pre>	<pre>#include <iostream.h> #include <fstream.h> int main(void) { // copierea unui ...</pre>

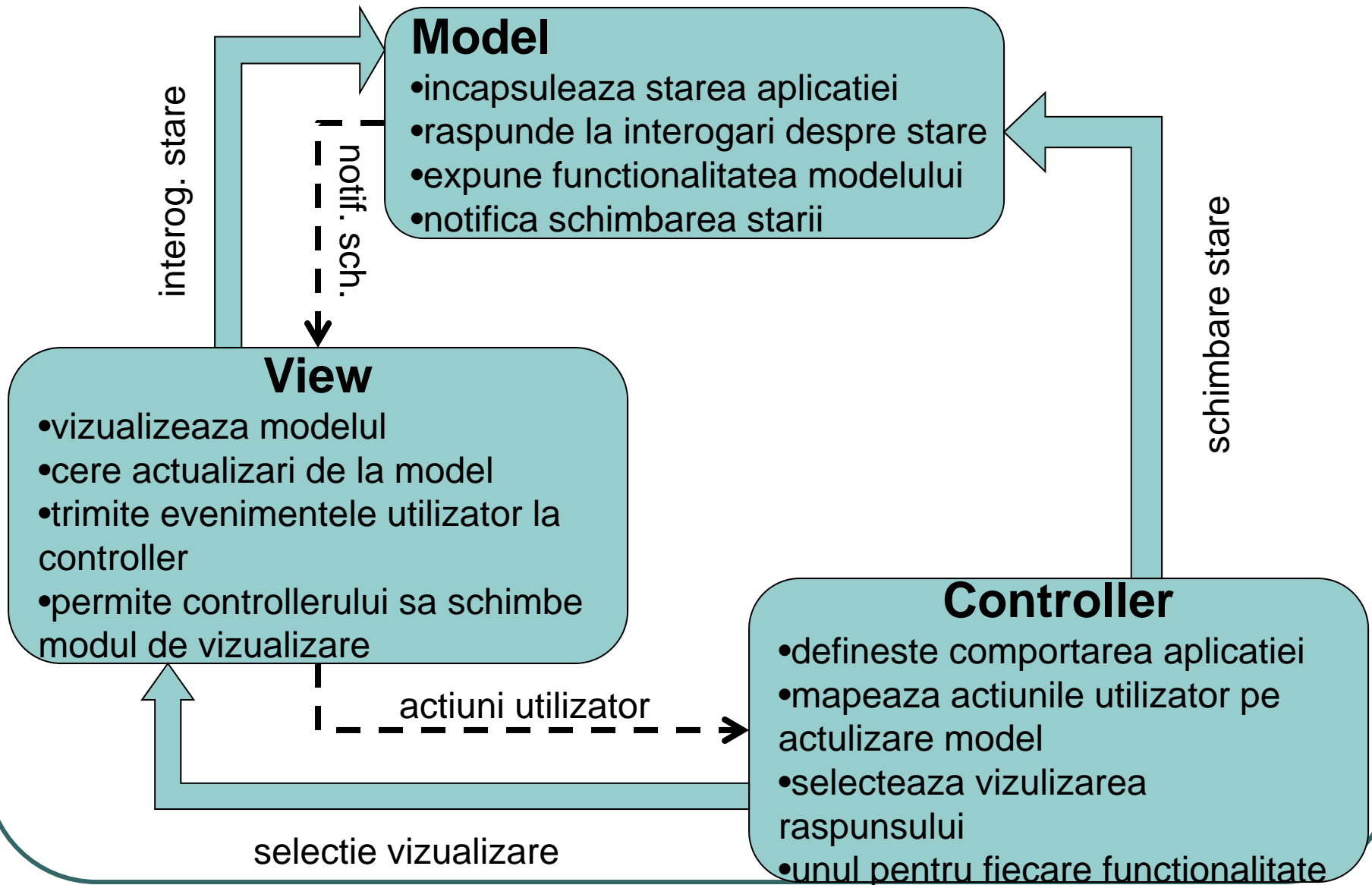
Cum construim o aplicatie OO?

- constructia unei aplicatii OO este similara cu cea a unei case: daca nu are o structura solida se darama usor
- ca si in cazul proiectarii cladirilor (urbanisticii), *patternurile* (sablaoanele) sunt aplicate cu succes
- patternurile pentru POO sunt similare structurilor de control (programarea structurata) pentru programarea imperativa
- noi vom studia
 - un pattern arhitectural (MVC)
 - cateva patternuri de proiectare
- mai mult la cursul de IP din anul II

Patternul model-view-controller (MVC)

- isi are radacinile in Smalltalk
 - maparea intrarilor, iesirilor si procesarii intr-un GUI model
- model – reprezinta datele si regulile care guverneaza actualizarile
 - este o aproximare software a sist. din lumea reala
- view – “interpreteaza” continutul modelului
 - are responsabilitatea de a mentine consistenta dintre schimbarile in model si reprezentare
- controller – translateaza interactiunile cu “view”-ul in actiuni asupra modelului
 - actiunile utilizatorului pot fi selectii de meniu, clickuri de butoane sau mouse
 - in functie de interactiunea cu utliz. si informatiile de la model, poate alege “view”-ul potrivit

MVC



MVC – studiu de caz

- MVC poate fi aplicat si pentru interactiune utilizator in mod text
- un studiu de caz simplu:
 - model = clasa Cont
 - view = afisare sold (interogare stare) si un meniu de actiuni (similar bancomat)
 - controller = transpune optiunile meniu in actualizare cont
- (deocamdata) fara notificari ale modelului

View

```
class View
{
private:
    Controller *controller;
    Count *model;
public:
    View(Controller *newController,
          Count *newCount);
    void display();
    int getUserAction();
};
```


View::display

```
void View::display()
{
    cout << endl << "New window" << endl;
    cout << "Balanta: "
         << model->balance() << endl;
    cout << "Commands:" << endl;
    cout << "1. Depune 50" << endl;
    cout << "2. Depune 100" << endl;
    cout << "3. Extrage 50" << endl;
    cout << "4. Extrage 100" << endl;
    cout << "0. Exit" << endl;
}
```

View::getUserAction

```
int View::getUserAction()  
{  
    int option;  
    cout << "Option: ";  
    cin >> option;  
    return option;  
}
```

Controller

```
class Controller
{
private:
    Count *model;
    View *view;
public:
    Controller(Count *newModel); public:
    void setView(View *newView);
    void execute();
    void finish();
};
```

Controller::execute

```
void Controller::execute()
{
    int option = view->getUserAction();
    switch (option)
    {
        case 0:
            finish();
            break;
        case 1:
            model->deposit(50);
            break;
        ...
        default: exit(1);
    }
}
```

Demo

```
int main() {
    Count *model = new Count("Ionescu");
    Controller *controller = new Controller(model);
    View *view = new View(controller, model);
    controller->setView(view);
    try {
        while (true) {
            view->display();
            controller->listen();
        }
    }
    catch (char *mes_err) {
        cout << mes_err << endl;
    }
    return 0;
}
```