POO

Modelare

D. Lucanu

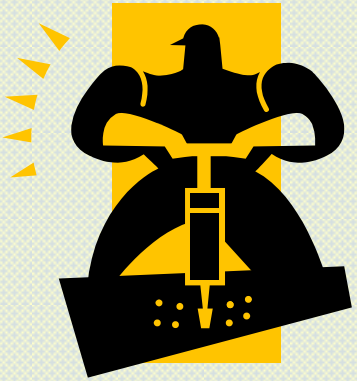
Cuprins

- fazele dezvoltarii unui produs soft
- modelare
- UML
  - diagrame *use case*
  - diagrame de clase
    - cum modelam in UML
    - cum implementam in C++
- MVC
  - descriere
  - studiu de caz

Cum dezvoltam un produs soft?

cautand solutii pentru probleme similare

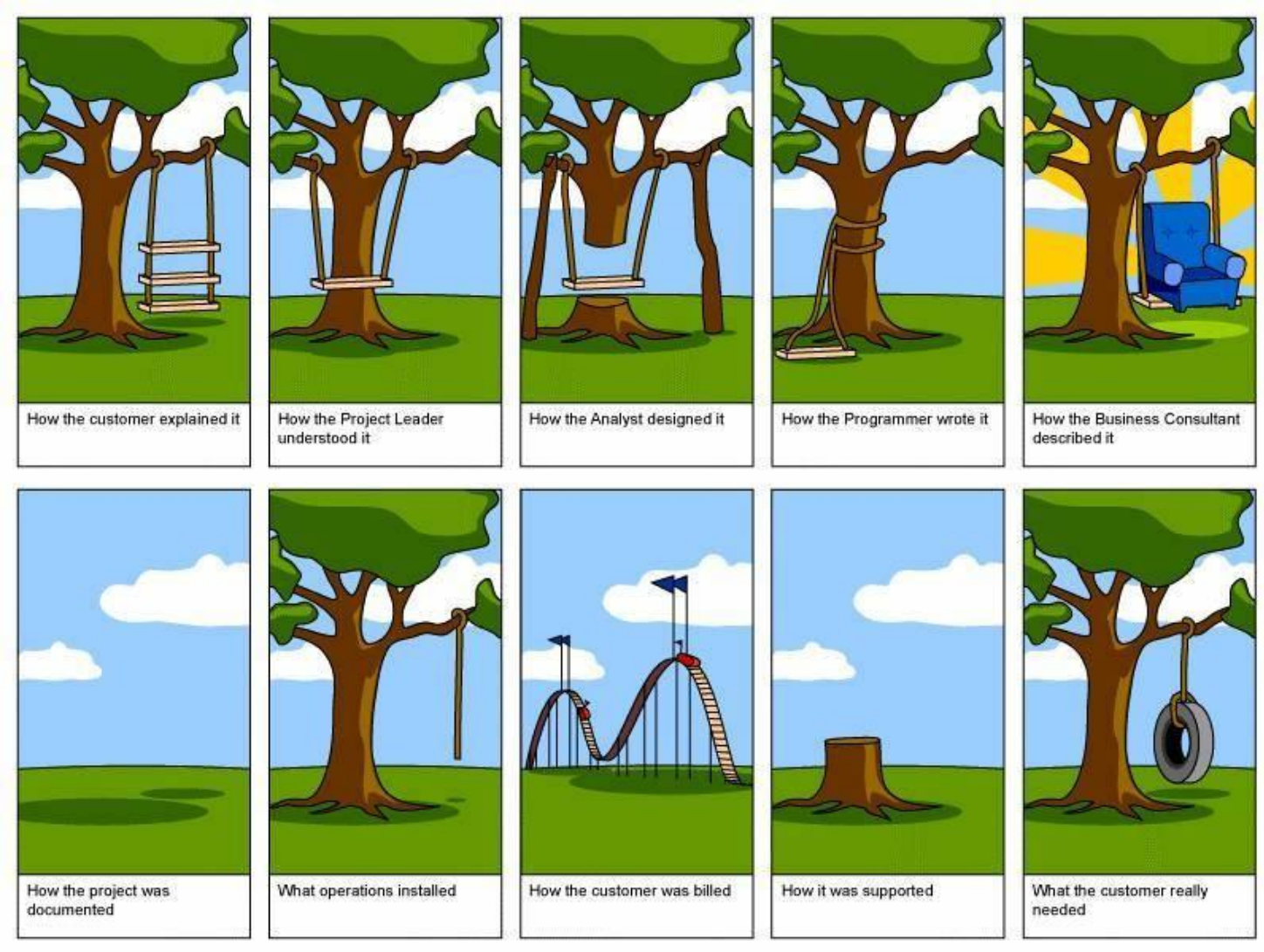
asambland componente existente



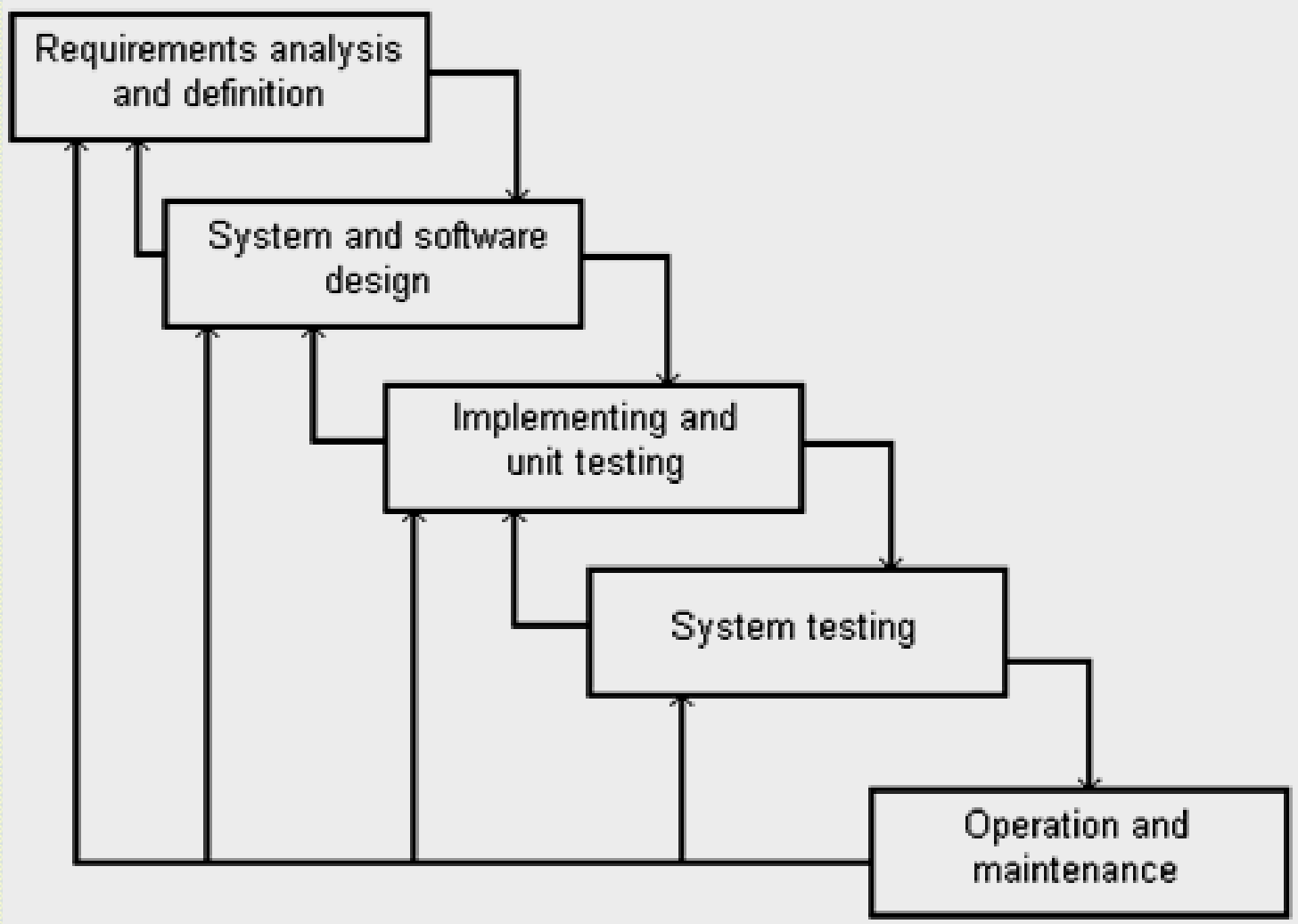
cautand noi solutii

... sau apeland la ajutor

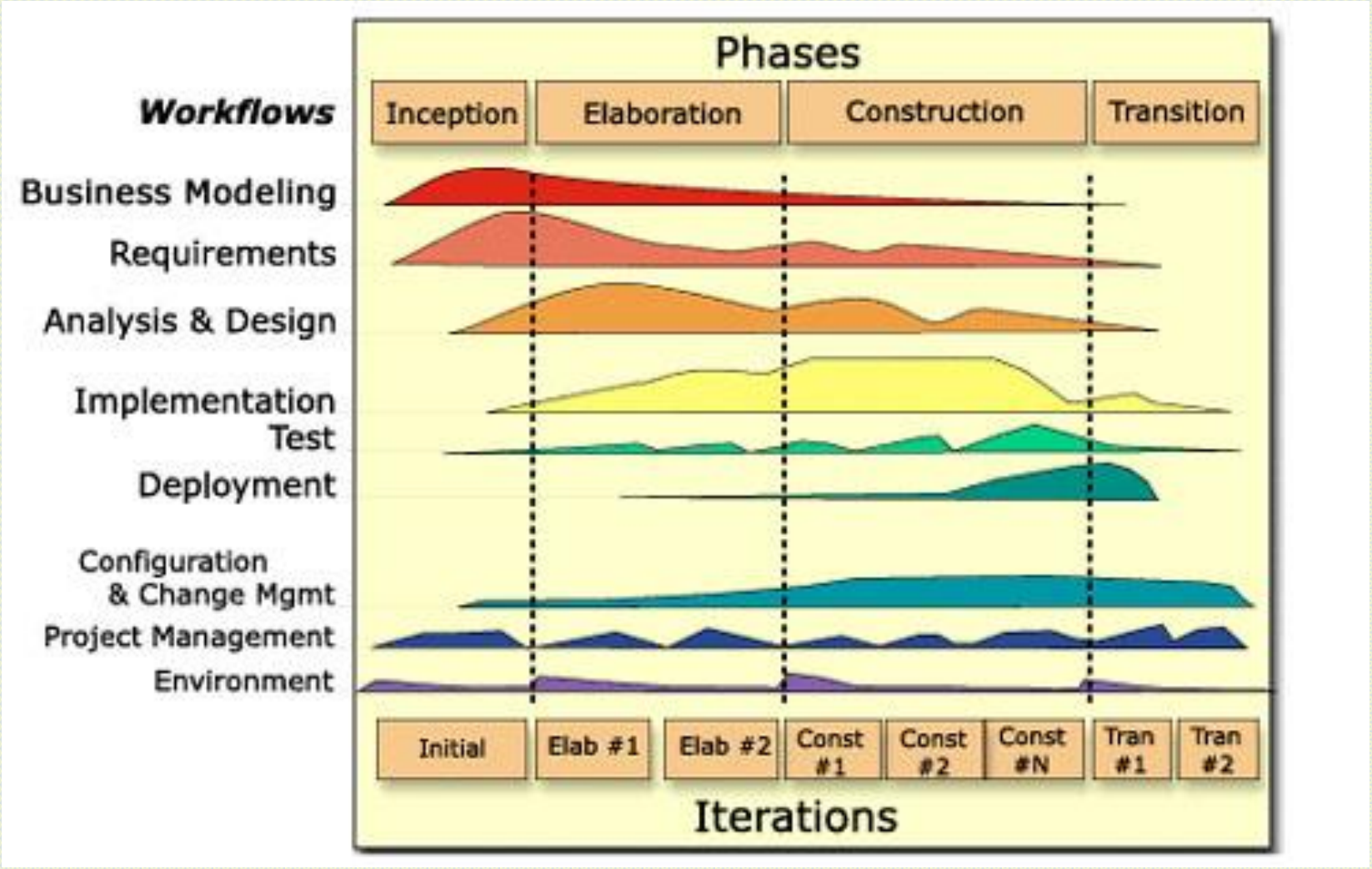
Relatia client – dezvoltator software



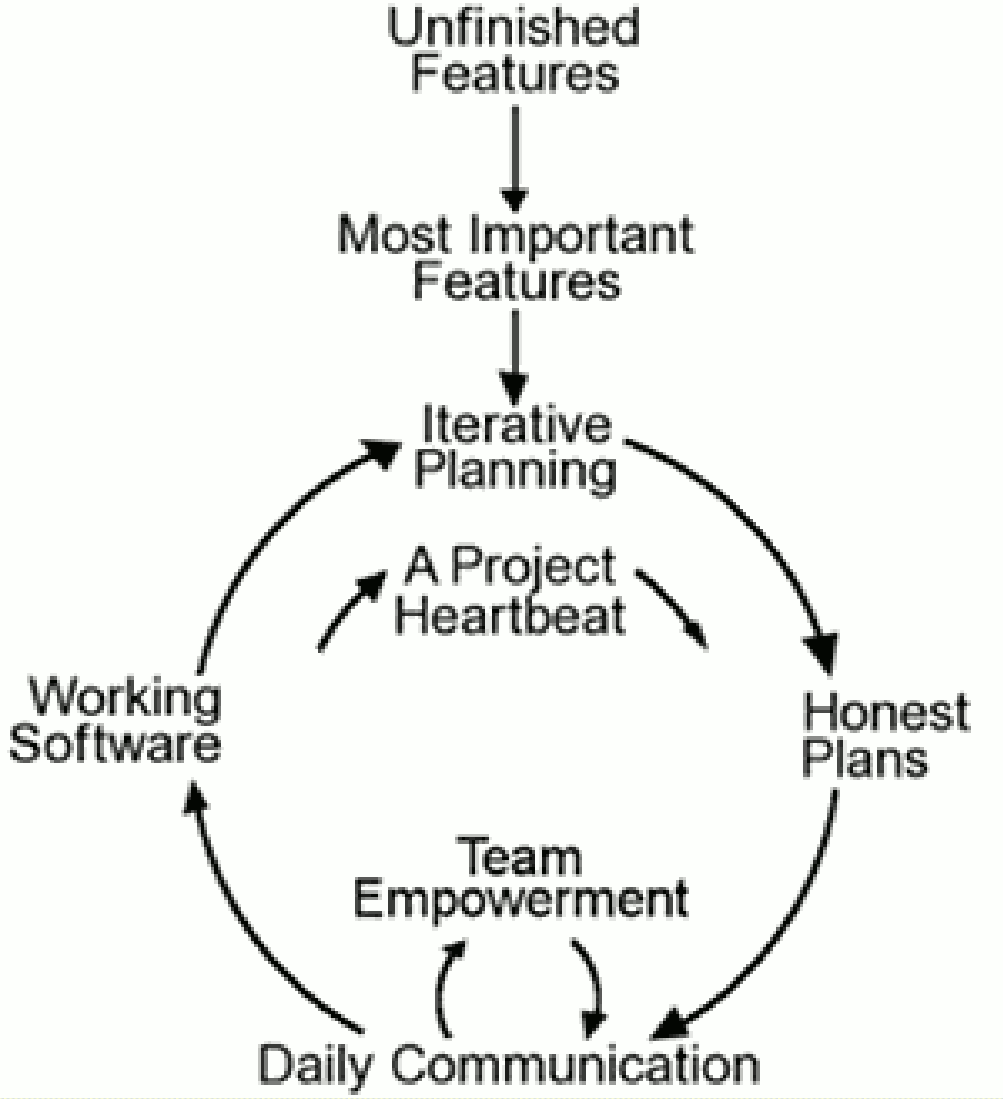
Dezvoltarea in cascada



Dezvoltarea in iteratii (RUP)



Dezvoltarea *agila* (extreme programming)



Important de tinut minte

- pentru proiectele realizate in timpul studiilor
  - clientul este profesorul
  - dezvoltatorul
    - studentul (proiecte individuale)
    - echipa de studenti (proiecte in echipa)

- mai multe despre metodologiile de dezvoltare a produselor soft la cursurile de IP

Ce este un model

- modelarea este esentiala in dezvoltarea eficienta de produse soft, indiferent de metodogia aleasa
- in principiu, rezultatele fazelor initiale si de elaborare sunt specificatii scrise ca modele
- un **model** este o simplificare a realitatii, fara insa a pierde legatura cu aceasta
- principalul motiv pentru care se construieste un model: necesitatea de a intelege sistemul ce urmeaza a fi dezvoltat
- cu cat sistemul este mai complex, cu atat importanta modelului creste
- alegerea modelului influienteaza atat modul in care problema este abordata cat si solutia proiectata
- in general, un singur model nu este suficient

UML – limbaj de modelare

- pentru a scrie un model, e nevoie de un limbaj de modelare
- UML (Unified Modeling Language) este un limbaj si o tehnica de modelare potrivite pentru programarea orientata-obiect
- UML este utilizat pentru a vizualiza, specifica, construi si documenta sisteme orientate-obiect
- la acest curs vom utiliza elemente UML pentru a explica conceptele si legile POO
- instrumente soft free: Argouml (open source), Visual Paradigm UML (Community edition)

Ce include UML 2.0

- diagrame de modelare structurala

  definesc arhitectura statica a unui model

  - diagrame de clase
  - diagrame de obiecte
  - diagrame de pachete
  - diagrame de structuri compuse
  - diagrame de componente
  - diagrame de desfasurare (deployment)

Ce include UML

- diagrame de modelare comportamentala

  definesc interactiunile si starile care pot sa apara la executia unui model

  - diagrame de utilizare (use case)
  - diagrame de activitati
  - diagrame de stari (state Machine diagrams)
  - diagrame de comunicare
  - diagrame de secvente (sequence diagrams)
  - diagrame de timp (fuzioneaza diagrame de stari cu cele de secvente)
  - diagrame de interactiune globala (interaction overview diagrams) (fuzioneaza diagrame de activitati cu cele de secvente)

Cum sunt utilizate modelele UML

- pot fi utilizate in toate fazele de dezvoltare a produselor soft (a se vede ciclurile de dezvoltare)
  - analiza cerintelor, e.g.,
    - diagramele cazurilor de utilizare
  - proiectare, e.g.,
    - diagramele de clase
    - diagrame de comunicare/secvente
    - diagrame de activitate
  - implementare,
    - diagramele constituie specificatii pentru cod
  - exploatare, e.g.,
    - diagrame de desfasurare

La acest curs vom insista ...

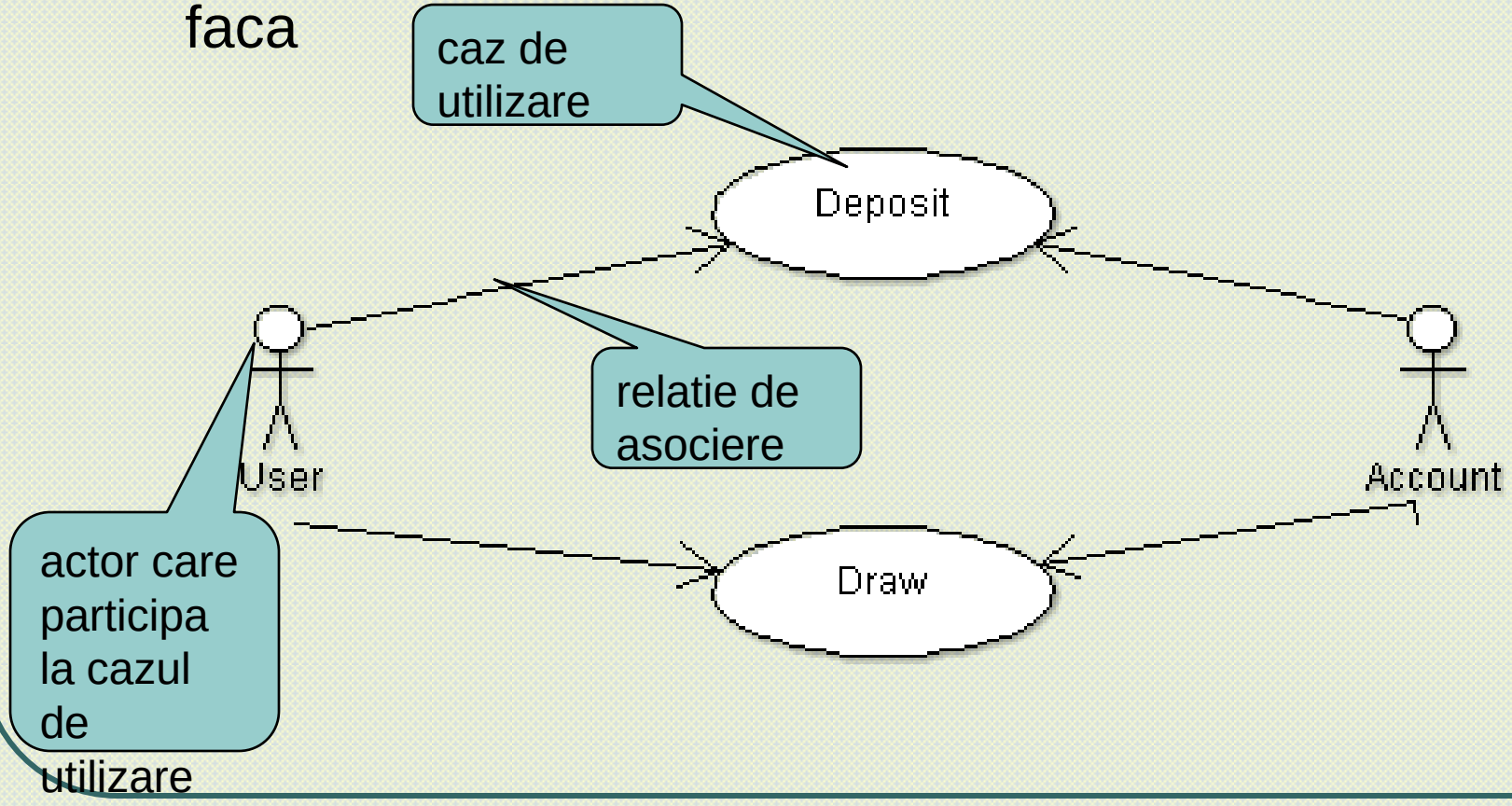
- ... doar pe
  - analiza cerintelor (la nivel introductiv)
    - diagramele cazurilor de utilizare
  - proiectare
    - diagramele de clase (detaliat)
  - implementare
    - cum scriem cod C++ din specificatiile date de diagrame (detaliat)
- mai mult la cursurile de IP

Ce este analiza OO

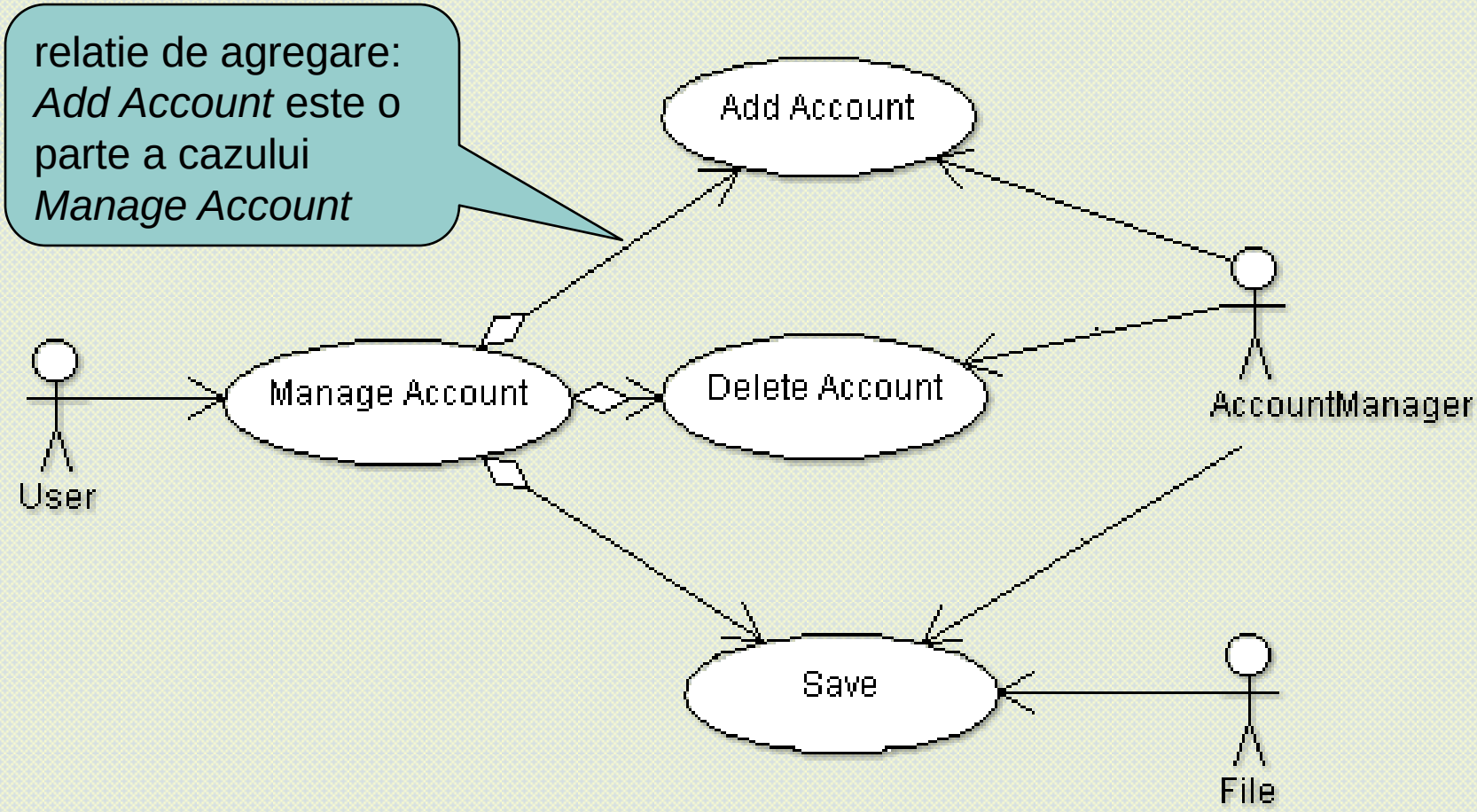
- analiza este focalizata mai mult pe intelegerea domeniului problemei si mai putin pe gasirea de solutii
- este orientata mai mult spre
  - ce (… trebuie, … inseamna, … relatii exista, …)
  - formulare si specificare cerinte
  - investigarea domeniului
  - intelegerea problemei
- descrie obiectele (conceptele) din domeniul problemei

Diagramele cazurilor de utilizare

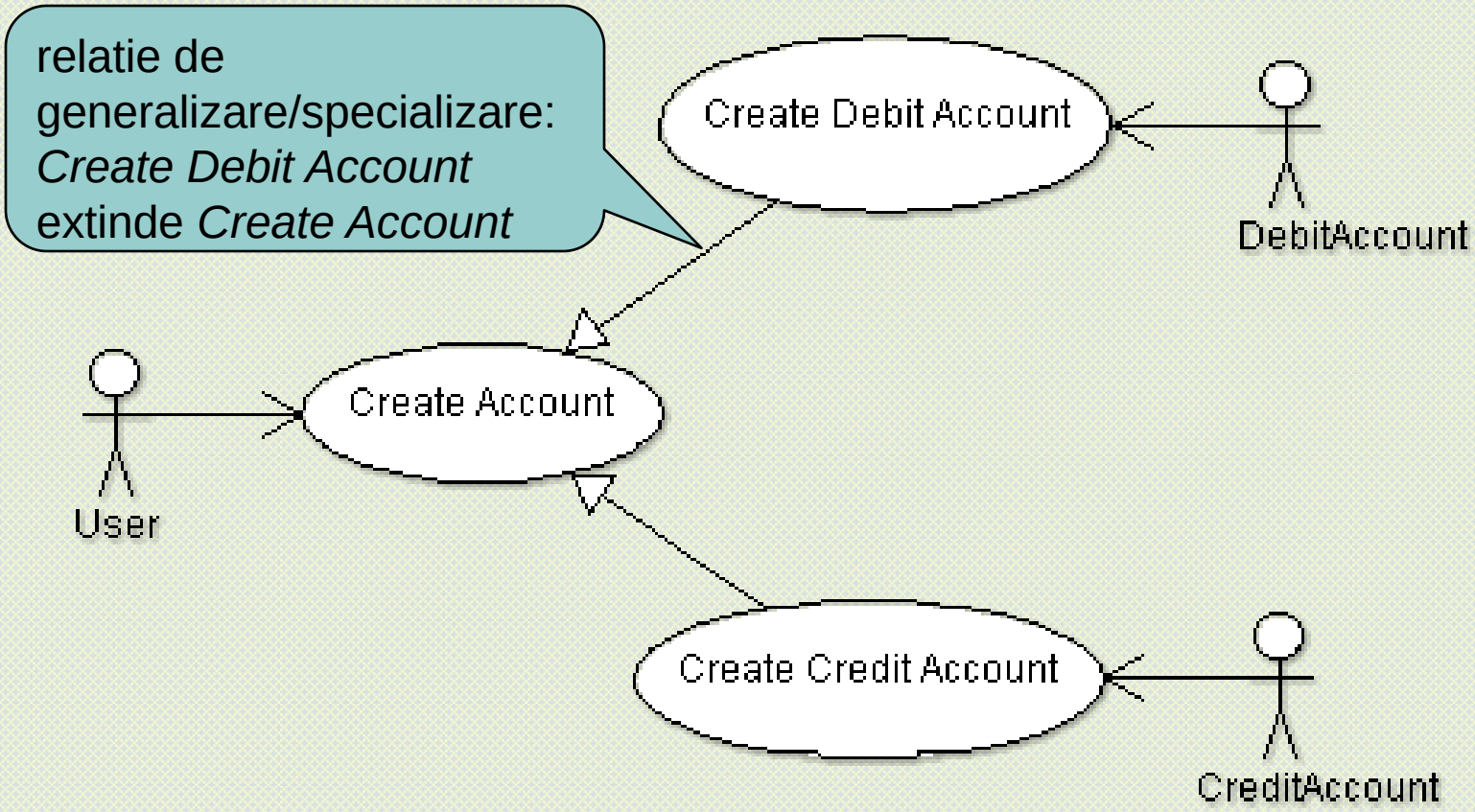
- modelul cazurilor de utilizare captureaza cerintele
- un **caz de utilizare (use case)** este un mijloc de a comunica utilizatorului ce intentioneaza sistemul sa faca



Relatia de includere intre cazuri de ut.



Relatia de extindere intre cazuri de ut.



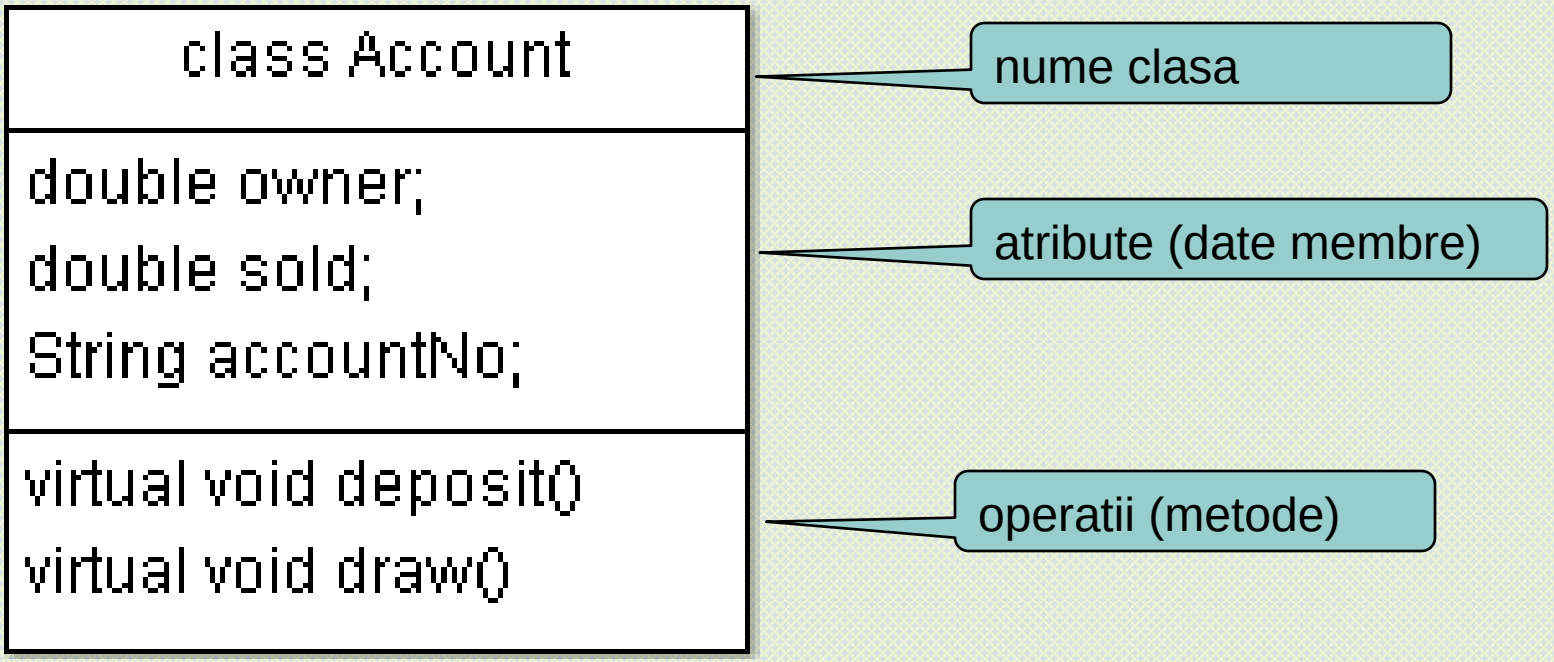
Proiectare OO (design)

- proiectarea se bazeaza pe solutia logica, cum sistemul realizeaza cerintele
- este orientata spre
  - cum
  - solutia logica
  - intelegerea si descrierea solutiei
- descrie obiectele (conceptele) ca avand atribute si metode
- descrie solutiam prin modul in care colaboreaza obiectele
- relatiile dintre concepte sunt descrise ca relatii intre clase

Diagrama de clase

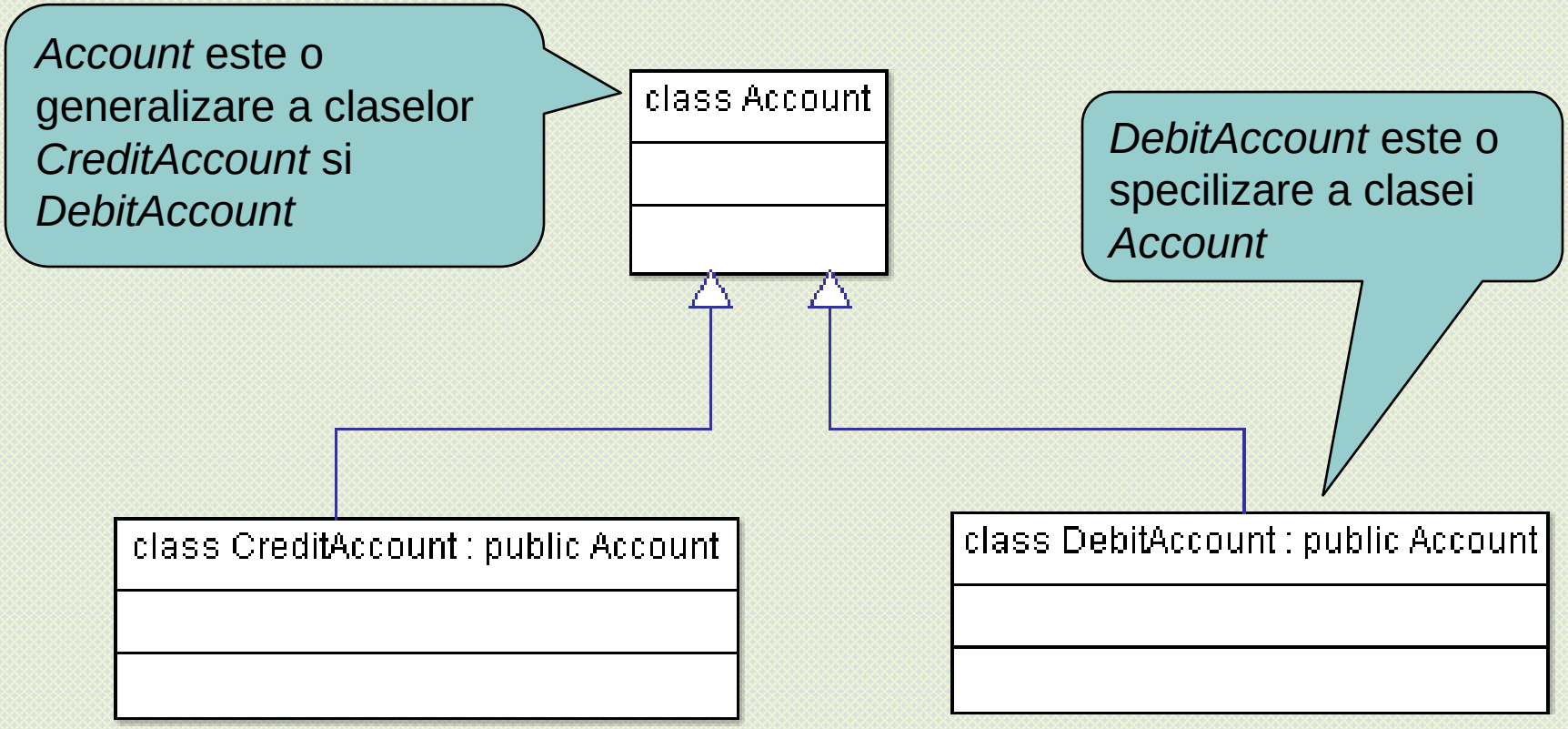
- include
  - clase
  - interfete
  - relatii intre clase
    - de generalizare/specializare
    - de asociere
    - de compozitie
    - de dependenta

Clasa

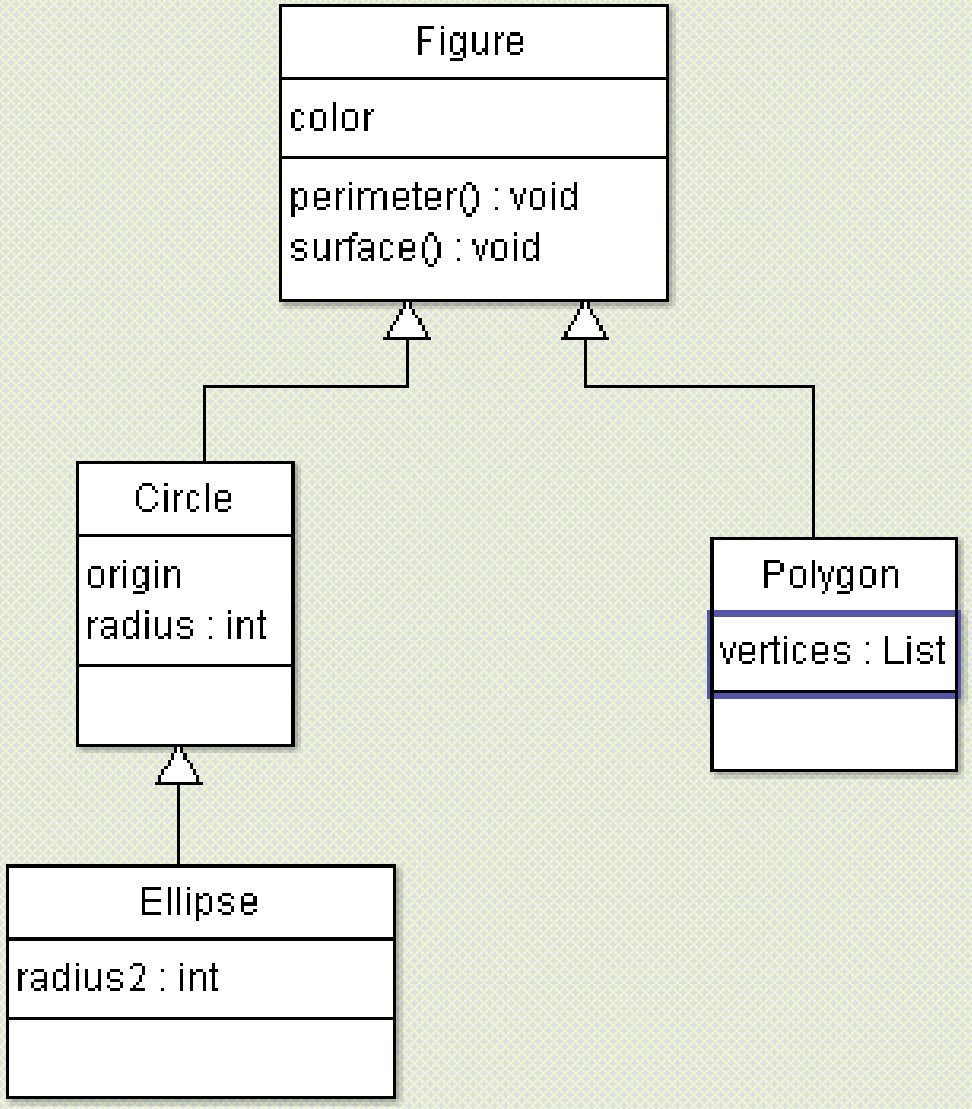

class Account
double owner;
double sold;
String accountNo;
virtual void deposit()
virtual void draw()
nume clasa
atribute (date membre)
operatii (metode)

Relatia de generalizare/specializare

- Relatia de mostenire este modelata in UML prin relatia de generalizare/specializare



Relatia de generalizare/specializare



Relatia de generalizare/specializare in C++

```
class Figure
{
...
};

class Circle
    : public Figure
{
...
};
```

```
class Ellipse
    : public Circle
{
...
};

class Polygon
    : public Figure
{
```

in C++ gen/spec se realizeaza prin relatia de derivare

Relatia de generalizare/specializare in C++

- operatiile ‘perimeter()’ si ‘surface()’ se calculeaza diferit de la figura la figura

```
class Figure {
public:
   virtual void perimeter() { return 0; }
};
class Circle : public Figure {
public:
   virtual void perimeter()
      { return 2 * 3.1415 * radius; }
};
```

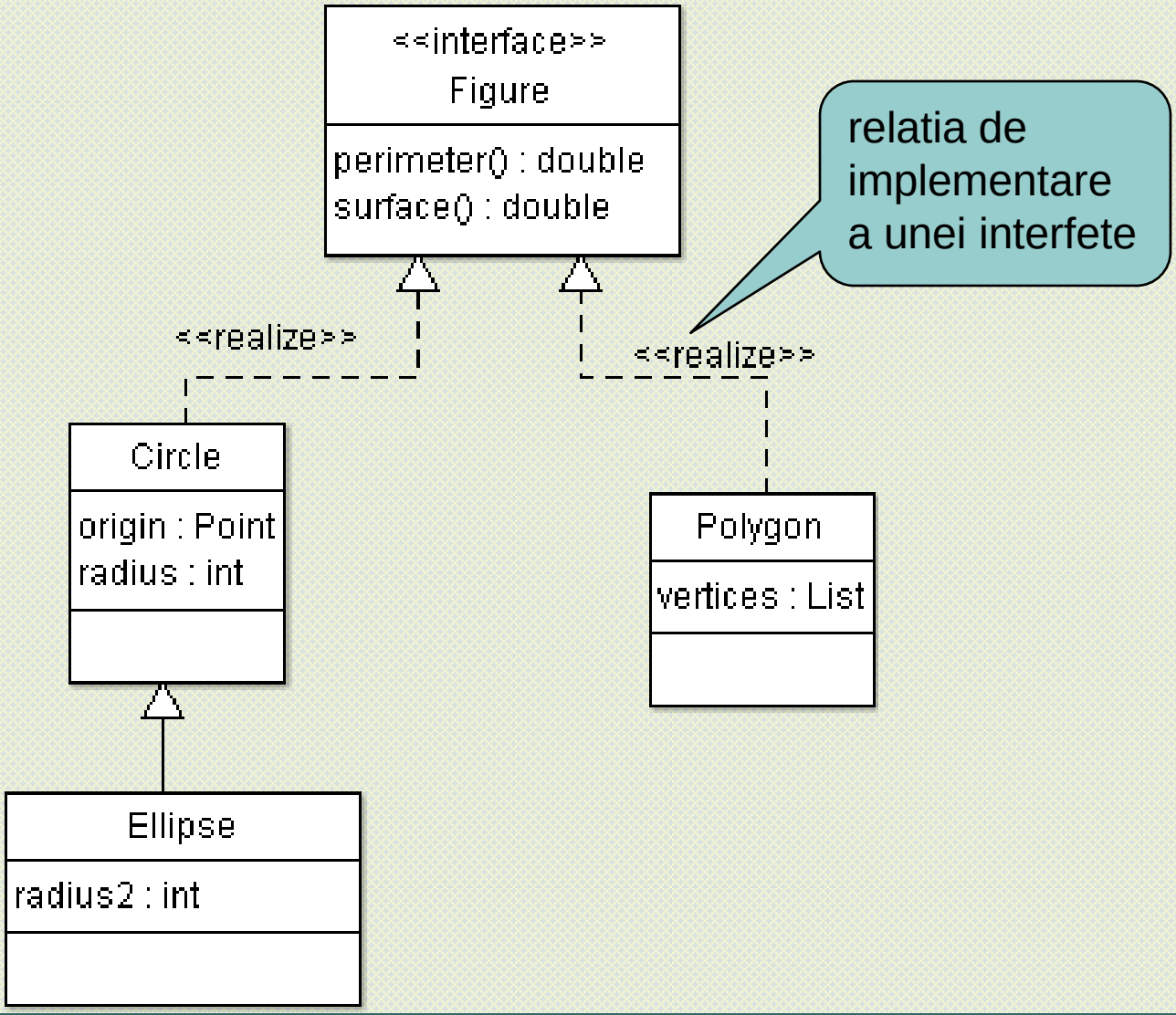
polimorfism prin suprascriere si legare dinamica

DEMO cu ArgoUML

Interfata

- obiecte de tip Figura nu exista la acest nivel de abstractizare
- clasa Figura este mai degraba o interfata pentru figurile concrete (cerc, poligon, elipsa …)
  - **interfata =** o colectie de operatii care caracterizeaza comportarea unui obiect

Interfata in UML

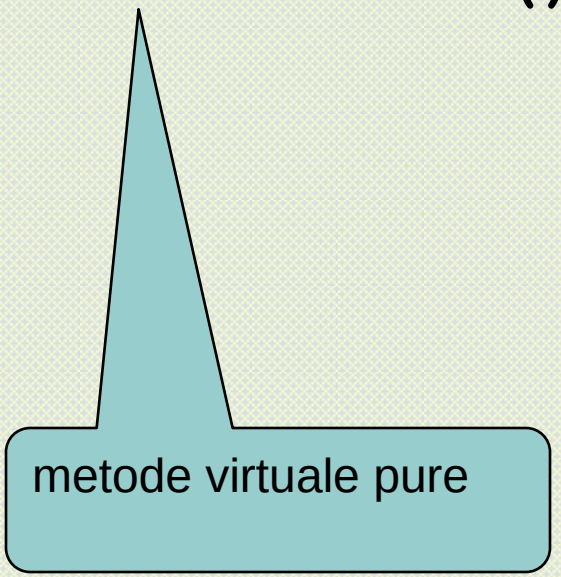


Interfata in C++

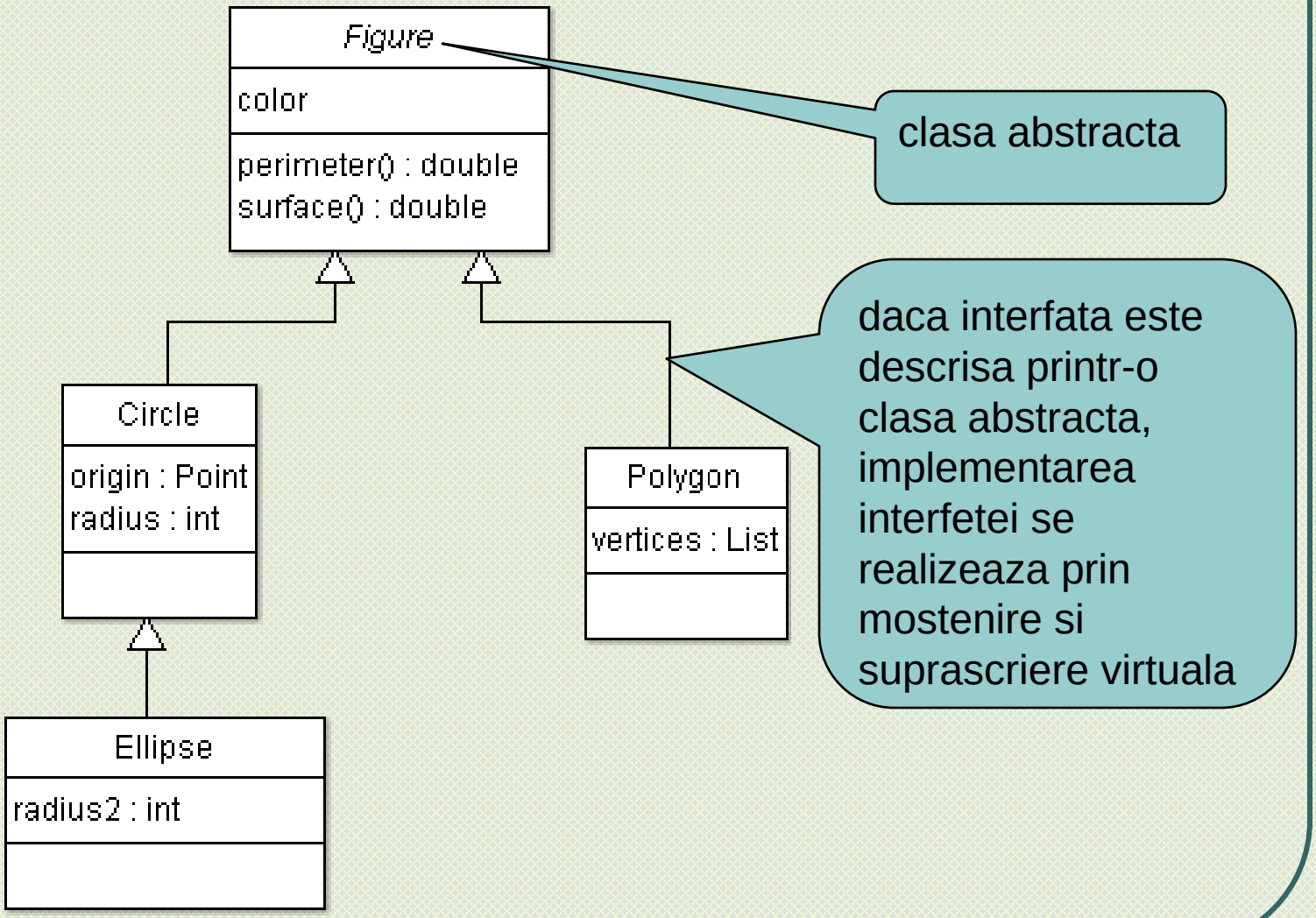
- interfetele in C++ sunt descrise cu ajutorul claselor abstracte
- o **clasa abstracta** nu poate fi instantiata, i.e., nu are obiecte
- de notat totusi ca **interfata si clasa abstracta sunt concepte diferite**
  - o clasa abstracta poate avea date membre si metode implementate
- in C++ o clasa este abstracta daca include **metode virtuale pure** (neimplementate)

Clase abstracte in C++

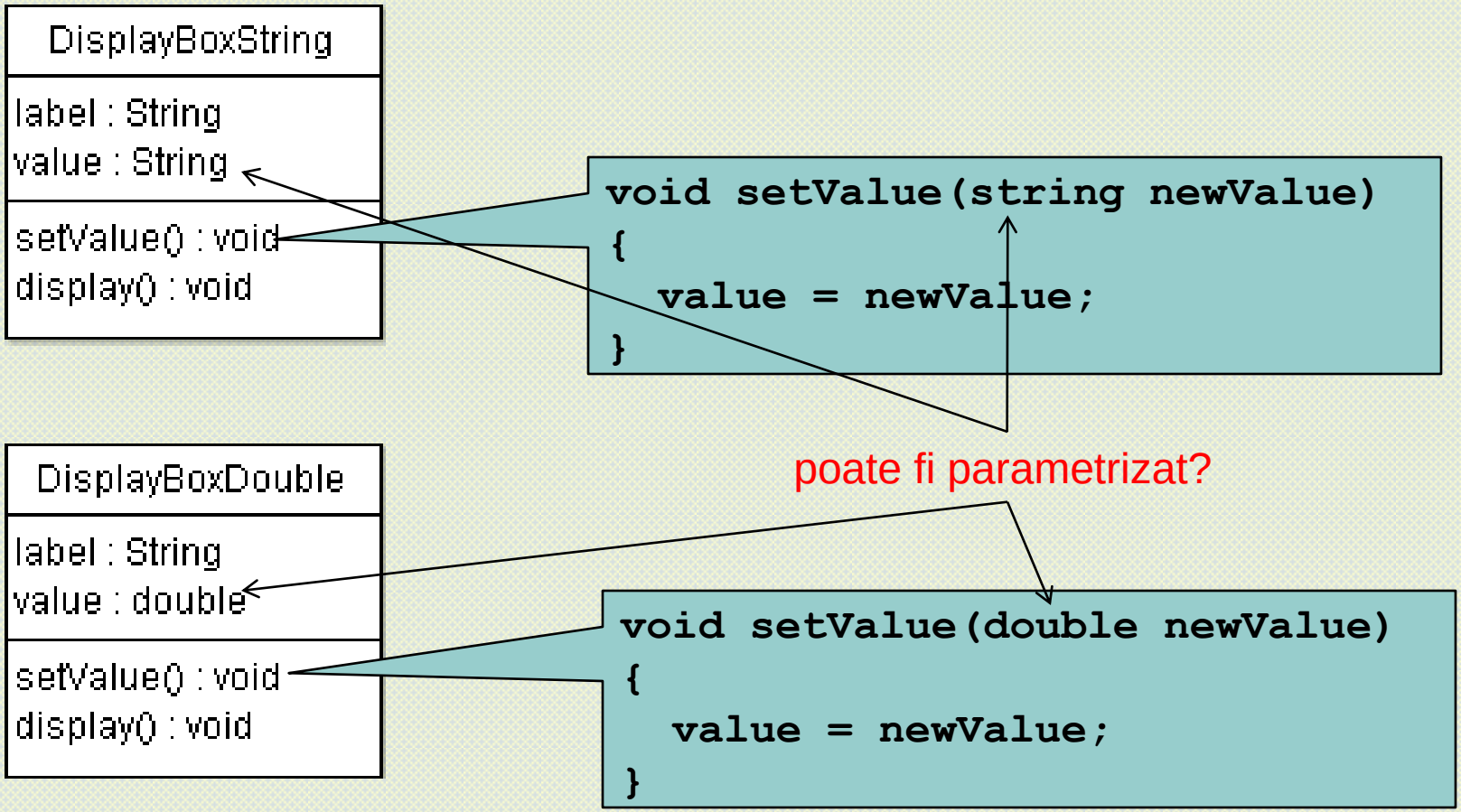
```
class Figure {
public:
   ...
   virtual void perimeter() = 0;
   virtual void surface() = 0;
   ...
};
```



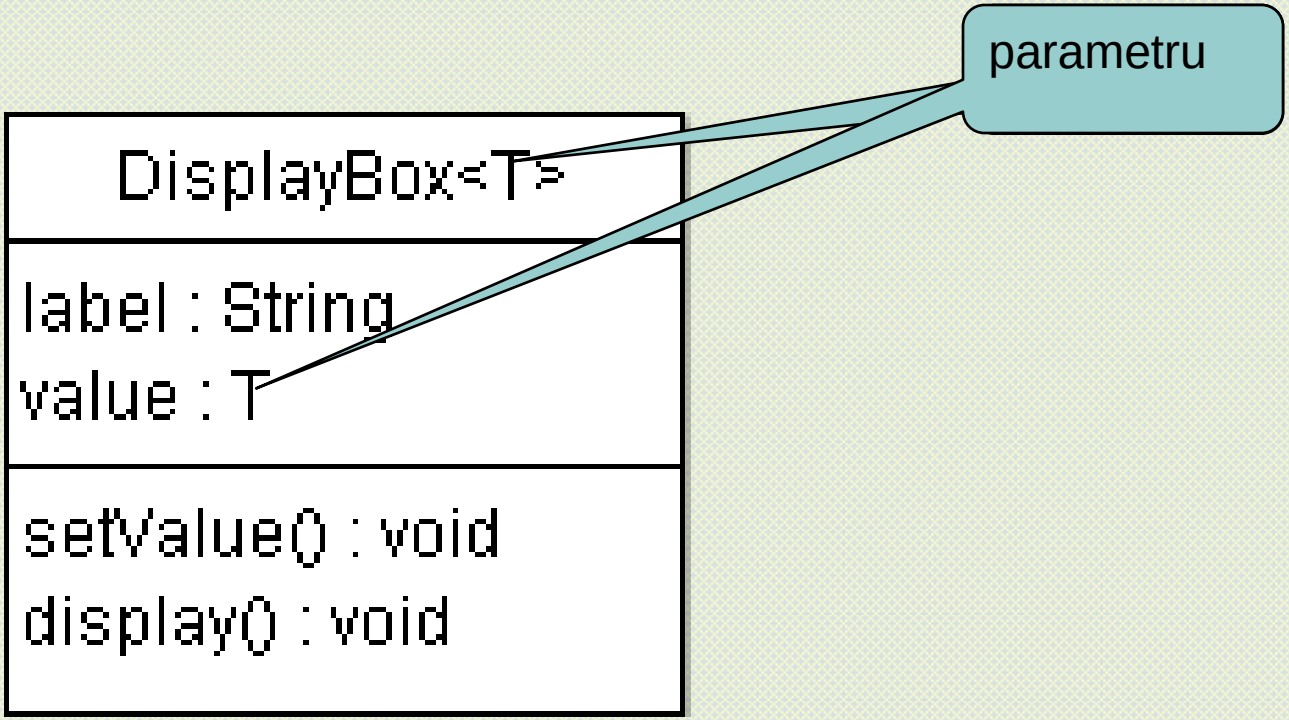
Diagrame cu clase abstracte



Abstractizare prin parametrizare

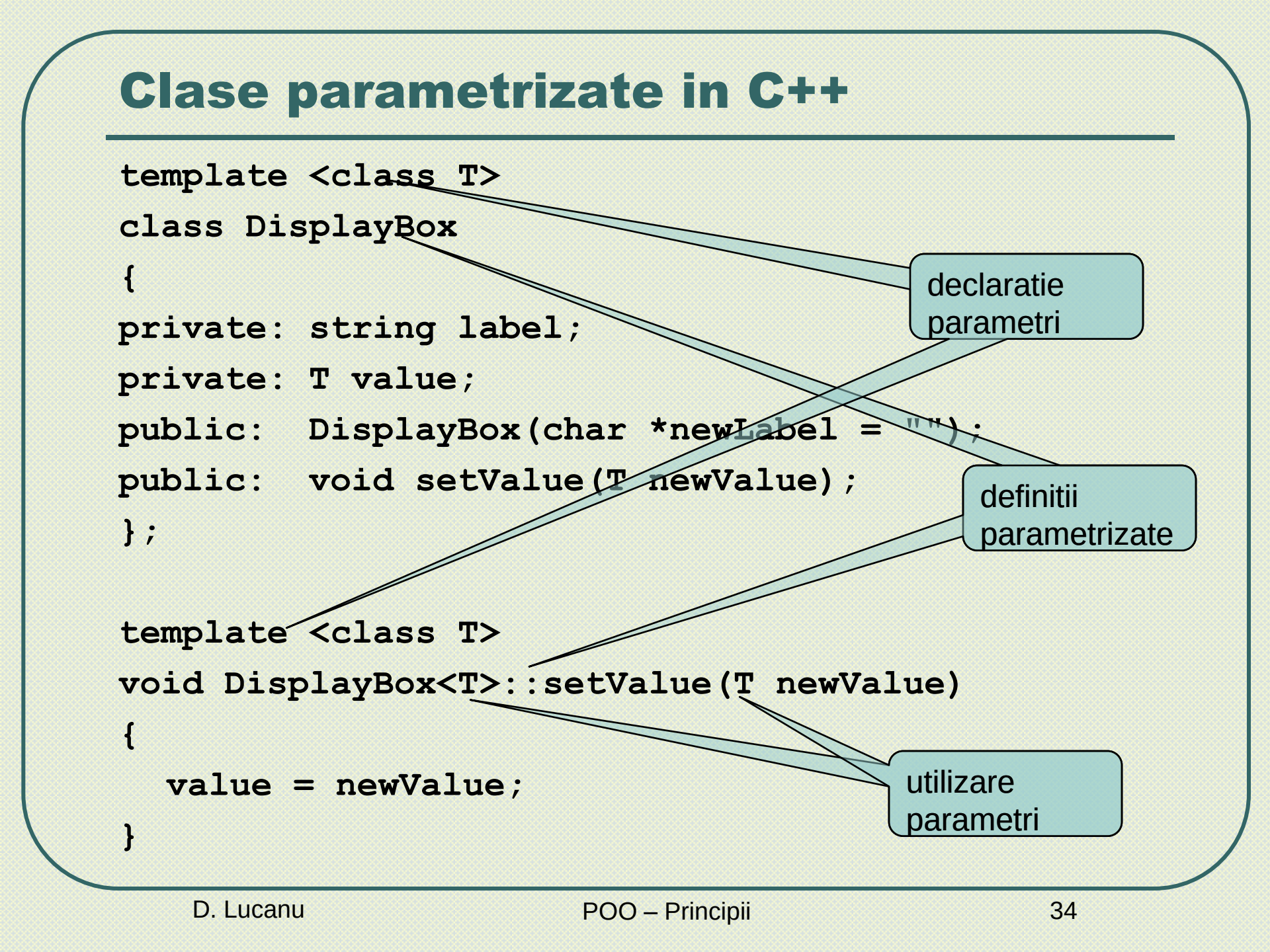


Clase parametrizate



Clase parametrizate in C++

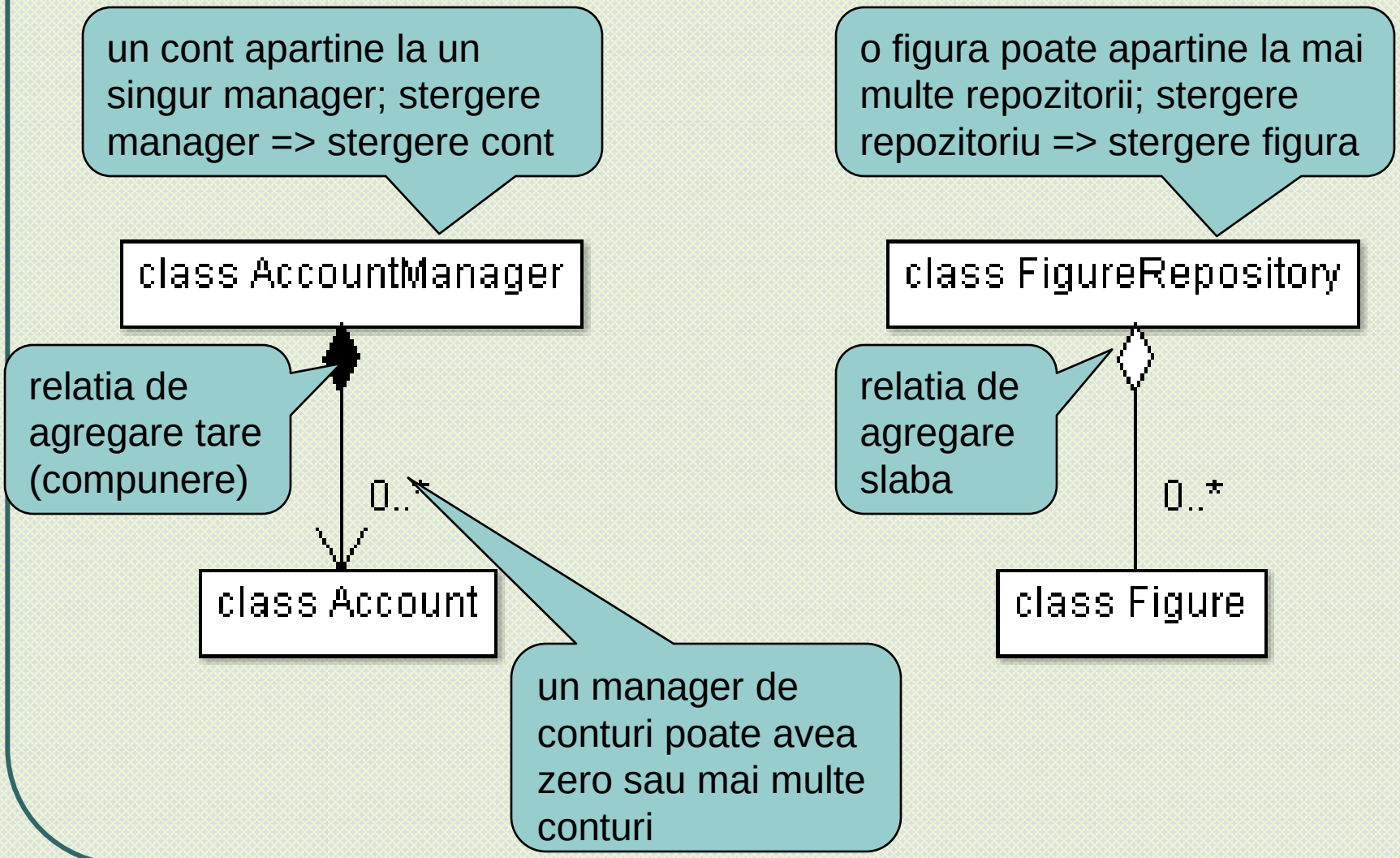
```
template <class T>
class DisplayBox
{
private: string label;
private: T value;
public:  DisplayBox(char *newLabel = "");
public:  void setValue(T newValue);
};

template <class T>
void DisplayBox<T>::setValue(T newValue)
{
   value = newValue;
}
```



Relatia de agregare (compunere)

- arata cum obiectele mai mari sunt compuse din obiecte mai mici
- poate fi privita si ca o relatie de asociere speciala
- exista doua tipuri de agregare
  - agregare slaba (romb neumplut), cand o componenta poate apartine la mai multe agregate (obiecte compuse)
  - agregare tare (romb umplut cu negru), cand o componenta poate apartine la cel mult un agregat (obiect compus)

Relatia de agregare



Agregare in C++

- agregare tare (compunere)

```
#include <list>
...
class AccountManager {
private:
   list< Account > accounts;
};
```

header pt. listele STL

liste in care componentele sunt obiecte Account

- agregare (slaba)

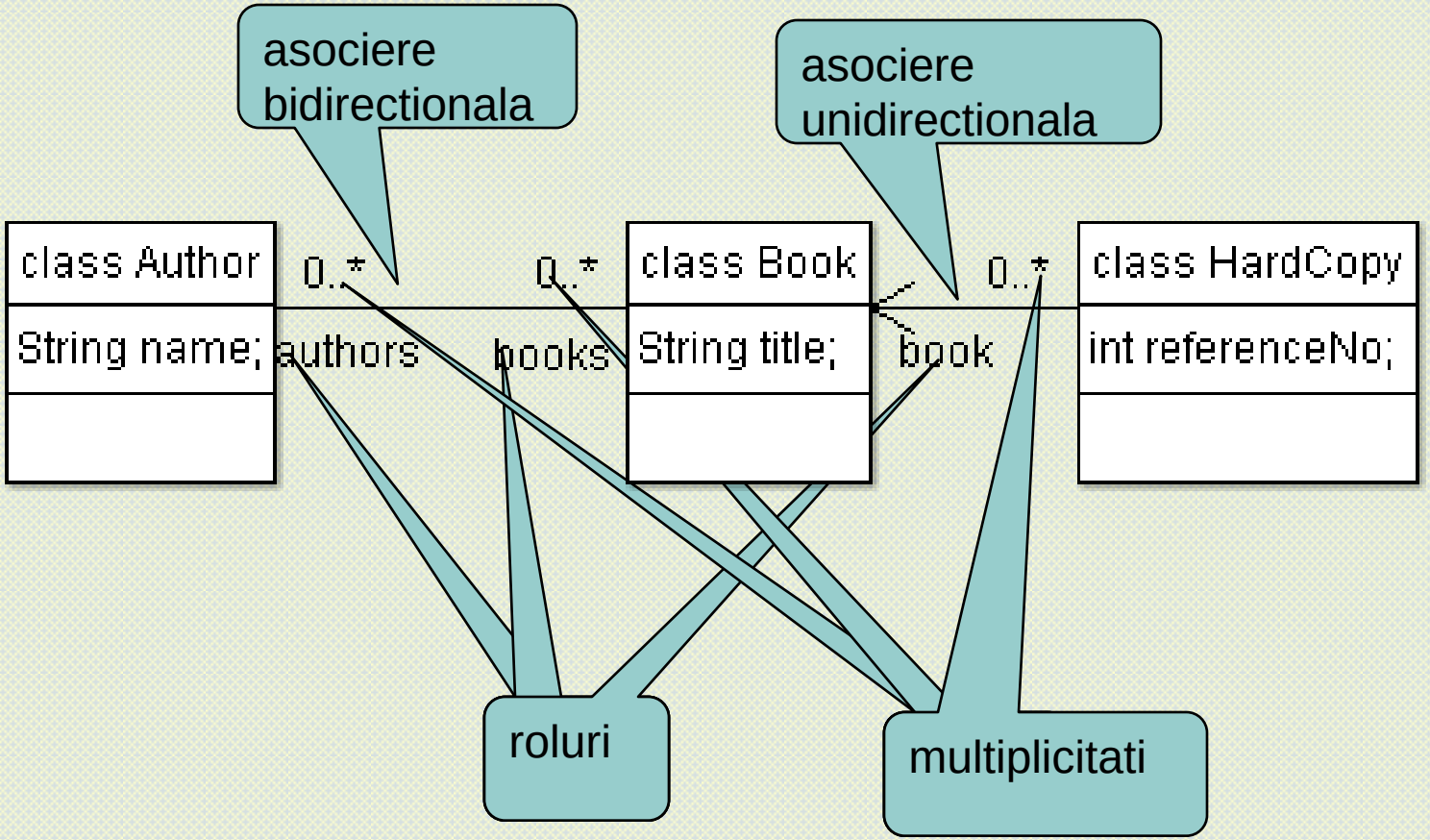
```
#include <list>
...
class FigureRepository {
private:
 list< Figure* > figures;
};
```

liste in care componentele sunt pointeri la obiecte Figure

DEMO

Relatia de asociere

- modeleaza relatii dintre obiecte



Relatia de asociere in C++

```
class Author {
private:
   list< Book* > books;
   ...
};

class Book {
private:
   list< Author* > authors;
   ...
};

class HardCopy {
private:
   Book book;
   ...
};
```

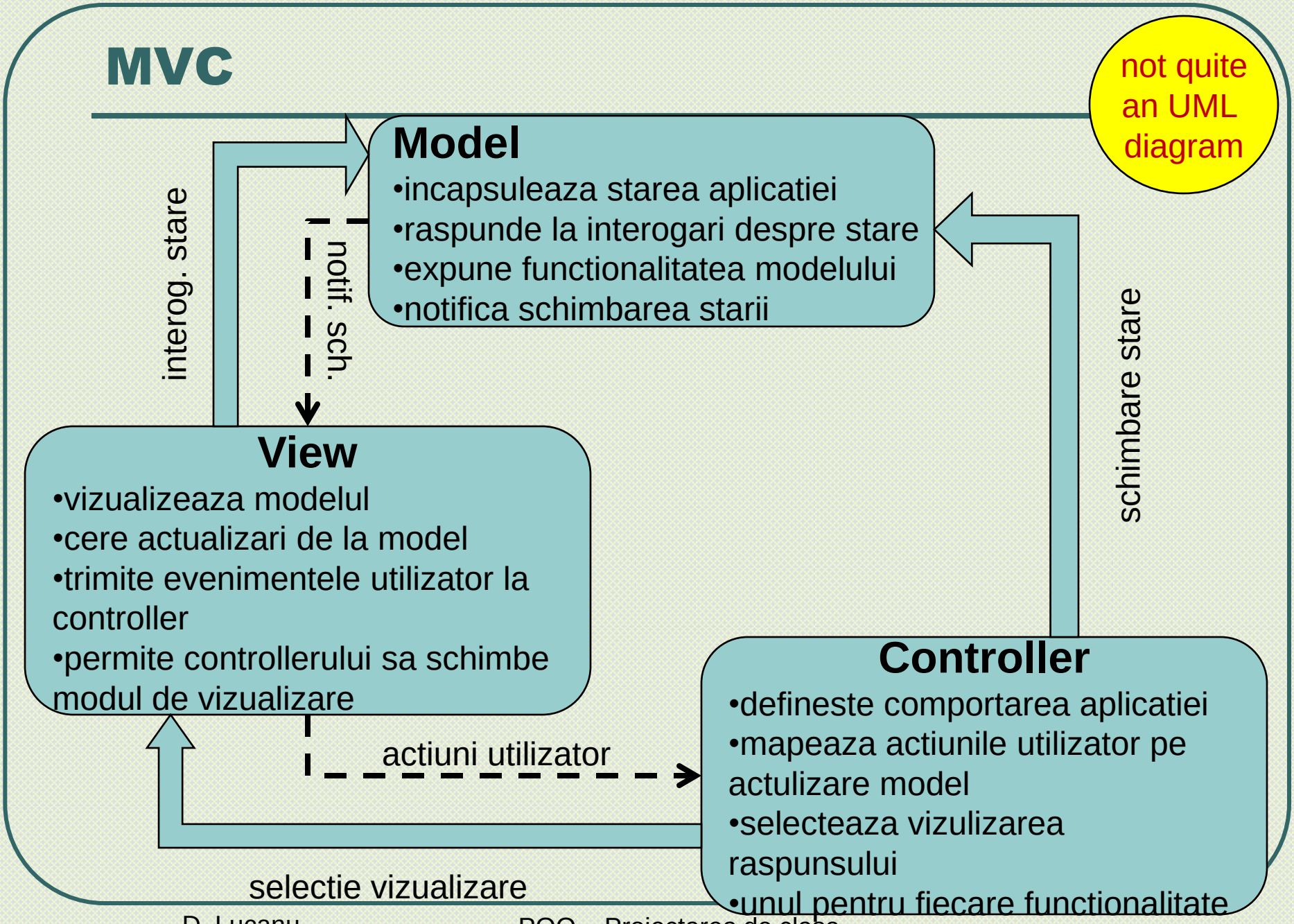
Cum construim o aplicatie OO?

- 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* (sablaonele) sunt aplicate cu succes
- patternurile pentru POO sunt similare structurilor de control (programarea structurata) pentru pogramarea 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 sistemului din lumea reala
- *view* – “interpreteaza” continutul modelului
  - are responsabilitateaa de a mentine consistenta dintre schimbarile in model si reprezentare
- *controller* – translateaza interactiunili 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

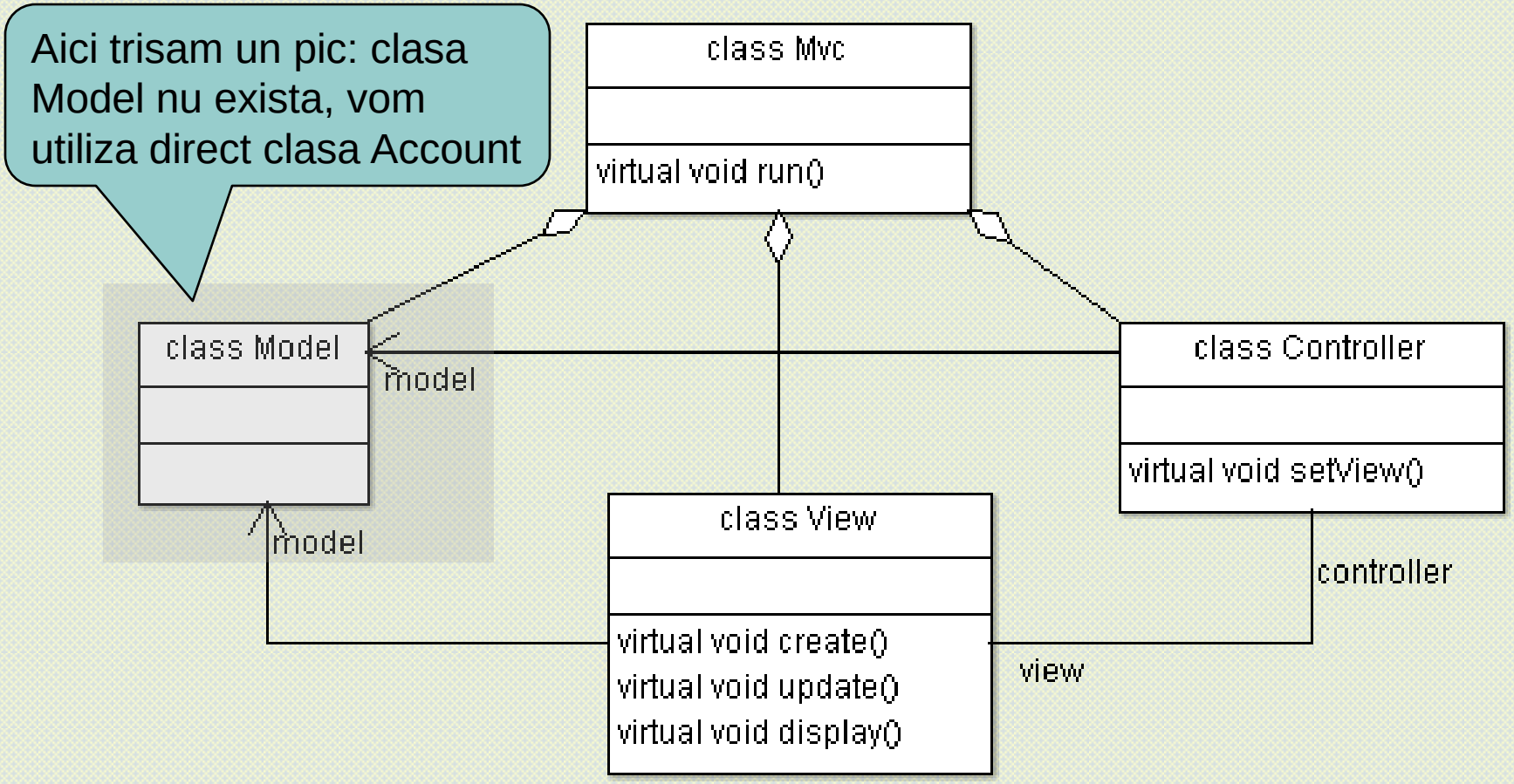
MVC


not quite an UML diagram
Model
•incapsuleaza starea aplicatiei
•raspunde la interogari despre stare
•expune functionalitatea modelului
•notifica schimbarea starii
interog. stare
notif. sch.
schimbare stare
View
•vizualizeaza modelul
•cere actualizari de la model
•trimite evenimentele utilizator la controller
•permite controllerului sa schimbe modul de vizualizare
actiuni utilizator
Controller
•defineste comportarea aplicatiei
•mapeaza actiunile utilizator pe actulizare model
•selecteaza vizulizarea raspunsului
•unul pentru fiecare functionalitate
selectie vizualizare

MVC – studiu de caz 1/5

- o mica aplicatie care simuleaza operatiile cun cont bancar
- model = clasa Account
- view
  - vizualizare date despre cont si un meniu cu operatiile principale asupra contului
- controller
  - preia optiunile din meniu ale utilizatorului si le transpune asupra modelului

MVC – studiu de caz 2/5



MVC – studiu de caz 3/5

```
class View {
public:
   virtual void create() = 0;  // pure virtual
   virtual void update() {};   // empty method
   virtual bool display() = 0; // pure virtual
};

class Controller {
protected:
   View *view;
public:
   void setView(View *newView)
   {
       view = newView;
   }
};
```

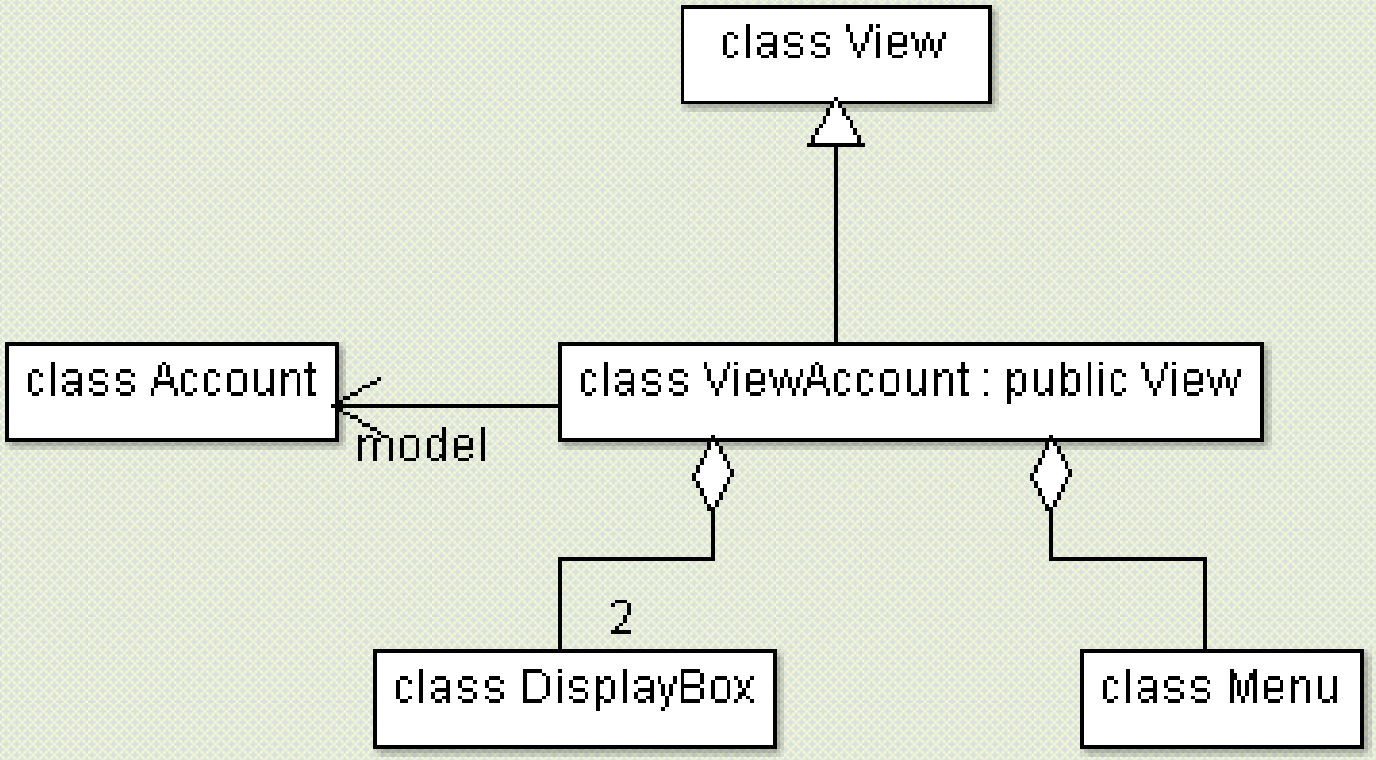
MVC – studiu de caz 4/5

```
template <class M, class V, class C>
class Mvc
{
private:
   C *c;
   V *v;
   M *m;
public:
   Mvc(M *newM = 0)
   {
       m = newM;
       c = new C(m);
       v = new V(c, m);
       c->setView(v);
   }
```

MVC – studiu de caz 5/5

```
public:
   ~Mvc()
   {
       delete c;
       delete v;
   }
public:
   void run()
   {
       while (v->display())
       {
             //nothing
       }
   }
};
```

clasa ViewAccount 1/4



clasa ViewAccount 2/4

```
class ViewAccount : public View
{
private:
   Account *model;
   ControllerAccount *controller;
   DisplayBox<string> dbOwner;
   DisplayBox<double> dbBalance;
   Menu menu;
public:
   ViewAccount(ControllerAccount *newController,
               Account *newModel)
   {
       controller = newController;
       model = newModel;
       create();
   };
```

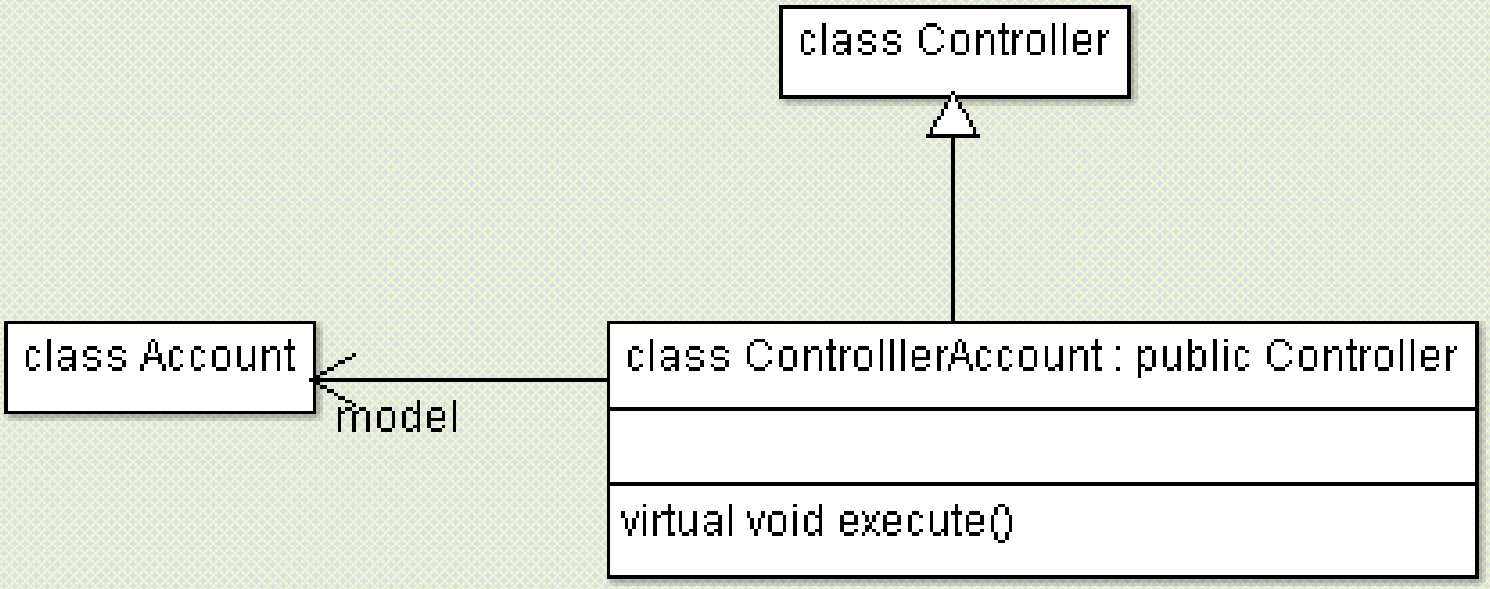
clasa ViewAccount 3/4

```
public:
   void create()
   {
       menu.addOption("1. Deposit 50");
       menu.addOption("2. Deposit 100");
       menu.addOption("3. Draw 50");
       menu.addOption("4. Draw 100");
       menu.addOption("0. Exit");
       dbOwner.setLabel("Owner");
       dbBalance.setLabel("Balance");
   }
public:
   void update()
   {
       dbOwner.setValue(model->getOwner());
       dbBalance.setValue(model->balance());
   }
```

clasa ViewAccount 4/4

```
public:
   bool display() {
        int option;
        cout << "*** Account view ***" << endl;
        update();
        dbOwner.display();
        dbBalance.display();
        menu.display();
        option = menu.getOption();
        if (option != 0)  {
              controller->execute(option);
              return true;
        }
        else
              return false;
   }
};
```

clasa ControllerAccount 1/3



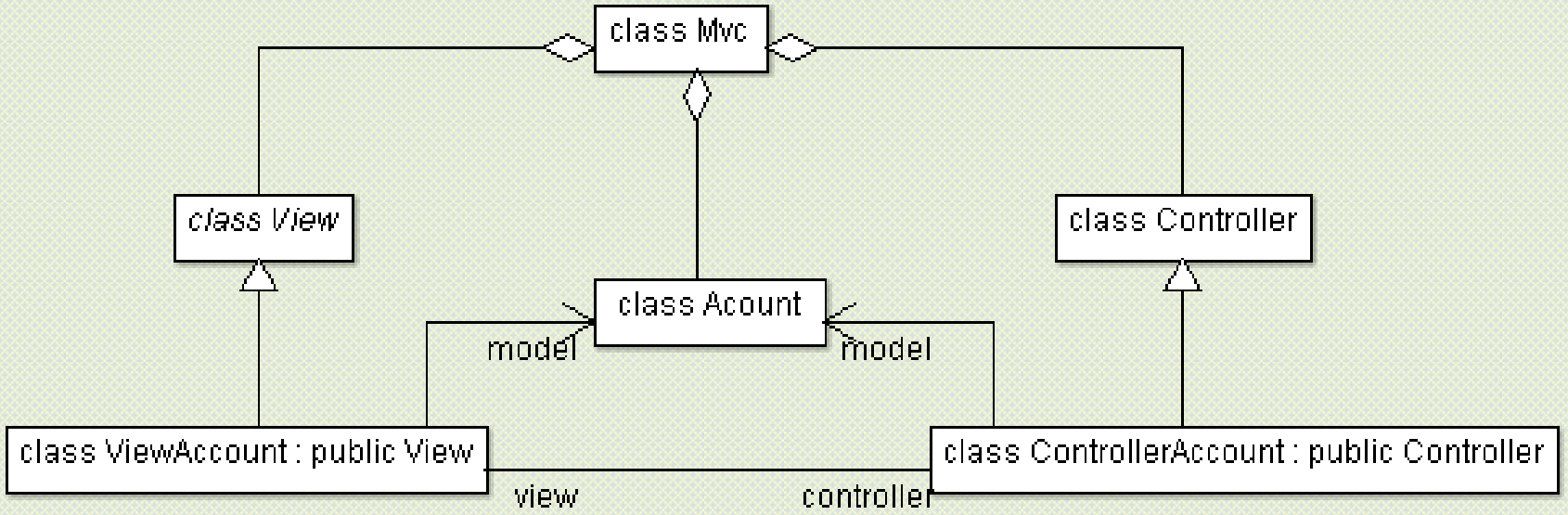
clasa ControllerAccount 3/3

```
class ControllerAccount : public Controller
{
private:
   Account *model;
public:
   ControllerAccount(Account *newModel)
   {
       model = newModel;

   }
```

clasa ControllerAccount 2/3

```
public: void execute(int option) {
        switch (option) {
                case 0:     break;
                case 1:
                        model->deposit(50);  break;
                case 2:
                        model->deposit(100); break;
                case 3:
                        model->draw(50);     break;
                case 4:
                        model->draw(100);    break;
                default: exit(1);
        }
   }
};
```

Diagrama MVC revizuita



DEMO