POO

Modelare

D. Lucanu

Cuprins

- modelare
- UML
  - diagrame *use case*
  - diagrame de clase
    - cum modelam in UML
    - cum implementam in C++
- MVC
  - descriere
  - studiu de caz

MODELARE

Context

- modelarea apare in toate metodologiile
- principalele activitati in care intervine modelarea:
  - *business modeling*
  - specificarea cerintelor (requirements)
  - analiza
  - proiectare (design)

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

Ce este un model

- 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

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
  - proiectare
    - diagramele de clase (detaliat)
  - implementare
    - cum scriem cod C++ din specificatiile date de diagrame (detaliat)
- mai multe informatii la adresa https://www.uml-diagrams.org/
- mai mult la cursurile de IP

PROIECTARE

Analiza OO versus Proiectarea OO

- analiza este focalizata mai mult pe intelegerea domeniului problemei si mai putin pe gasirea de solutii
- este orientata mai mult spre
  - a intelege cine face, ce face si unde in cadrul domeniului afacerii
  - a formula ce trebuie furnizat pentru a ajuta actorii sa-si realizeze sarcinile
- presupune
  - formulare si specificare cerinte
  - investigarea domeniului
  - intelegerea domeniului problemei
- descrie obiectele (conceptele) din domeniul problemei

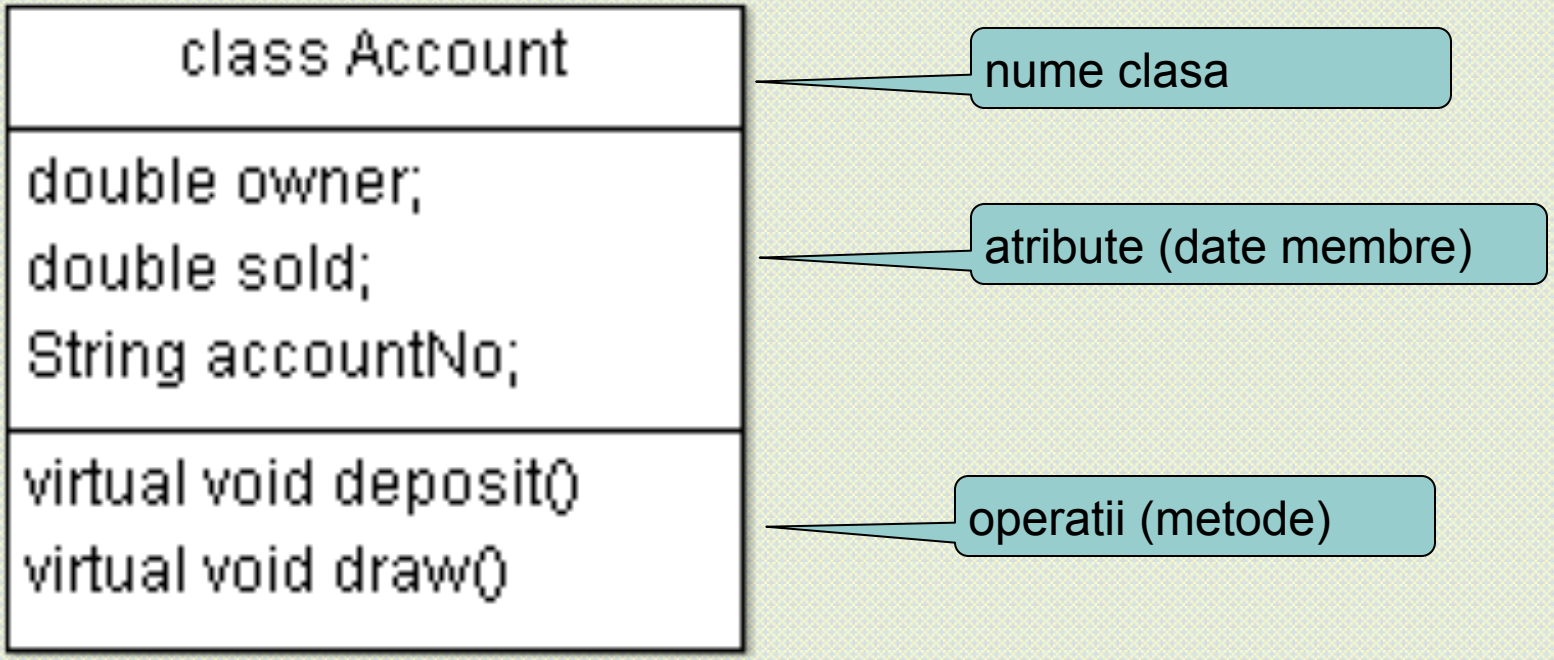
Analiza OO versus Proiectarea OO

- proiectarea (design) 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 solutia prin modul in care colaboreaza obiectele
- relatiile dintre concepte sunt descrise ca relatii intre clase

Proiectare: Diagrama de clase

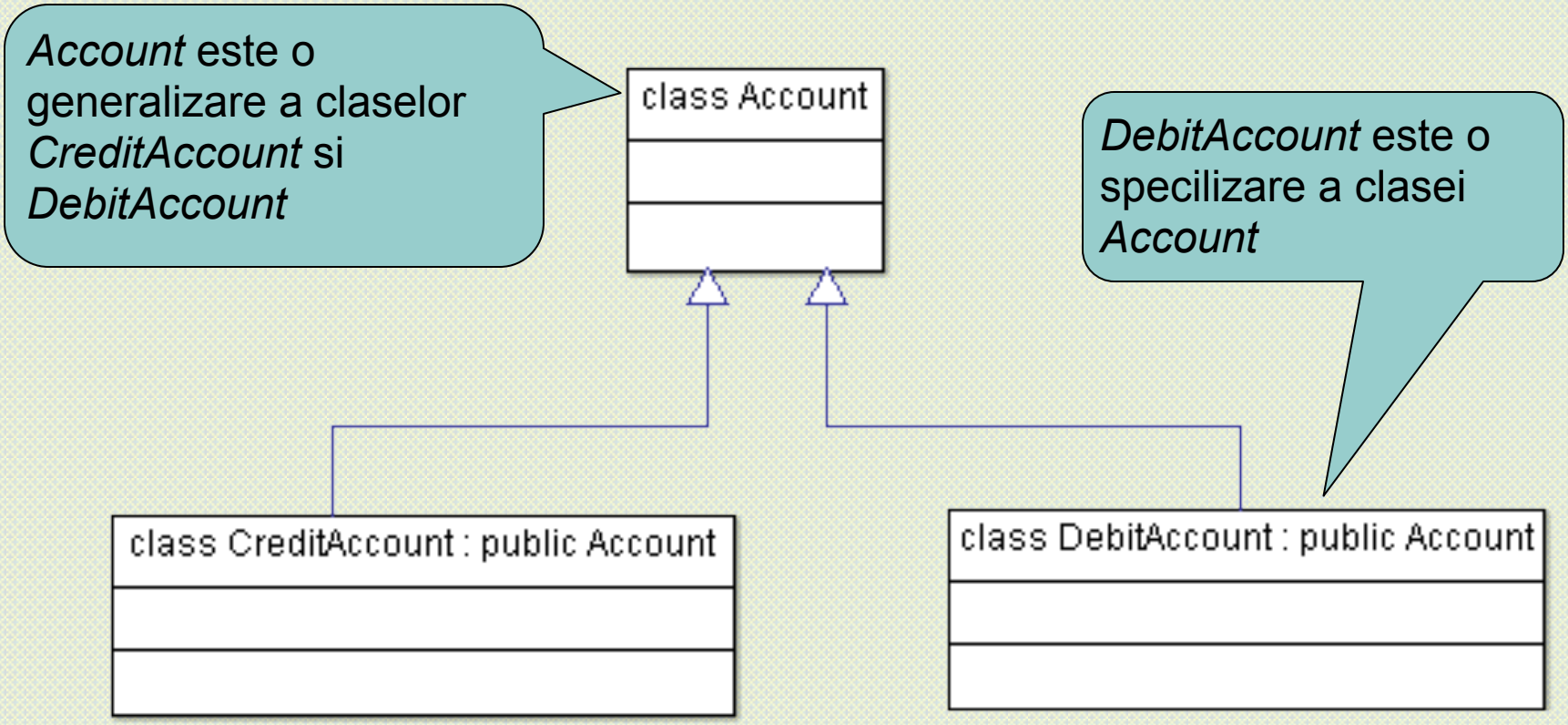
- include
  - clase
  - interfete
  - relatii intre clase
    - de generalizare/specializare
    - de asociere
    - de compozitie
    - de dependenta

Clasa

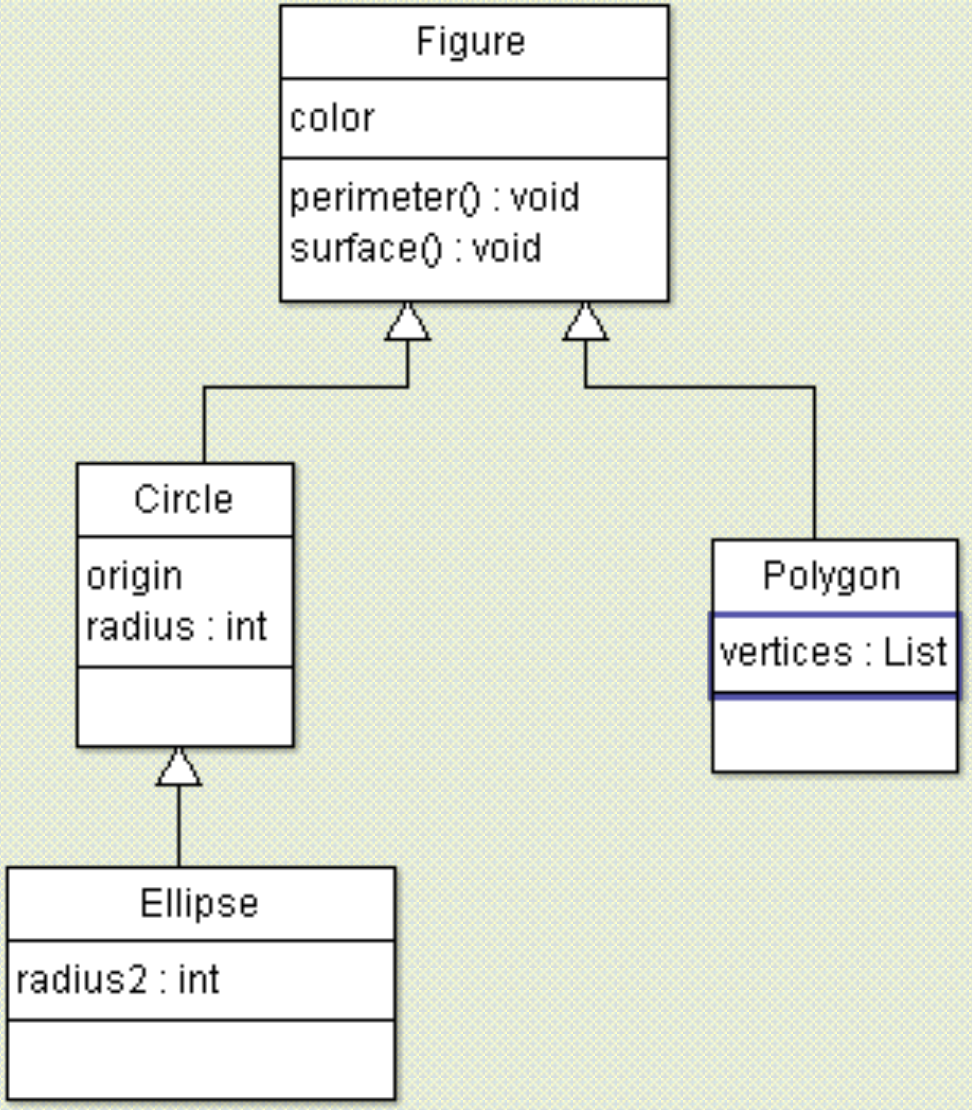

class Account
double owner;
double sold;
String accountNo;
virtual void deposit()
virtual void draw()
nume clasa
atribute (date membre)
operatii (metode)

Relatia de generalizare/specializare

- Relatia de mostenire este modelata in UML prin relatia de generalizare/specializer (un concept mai larg)



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; }
};
```

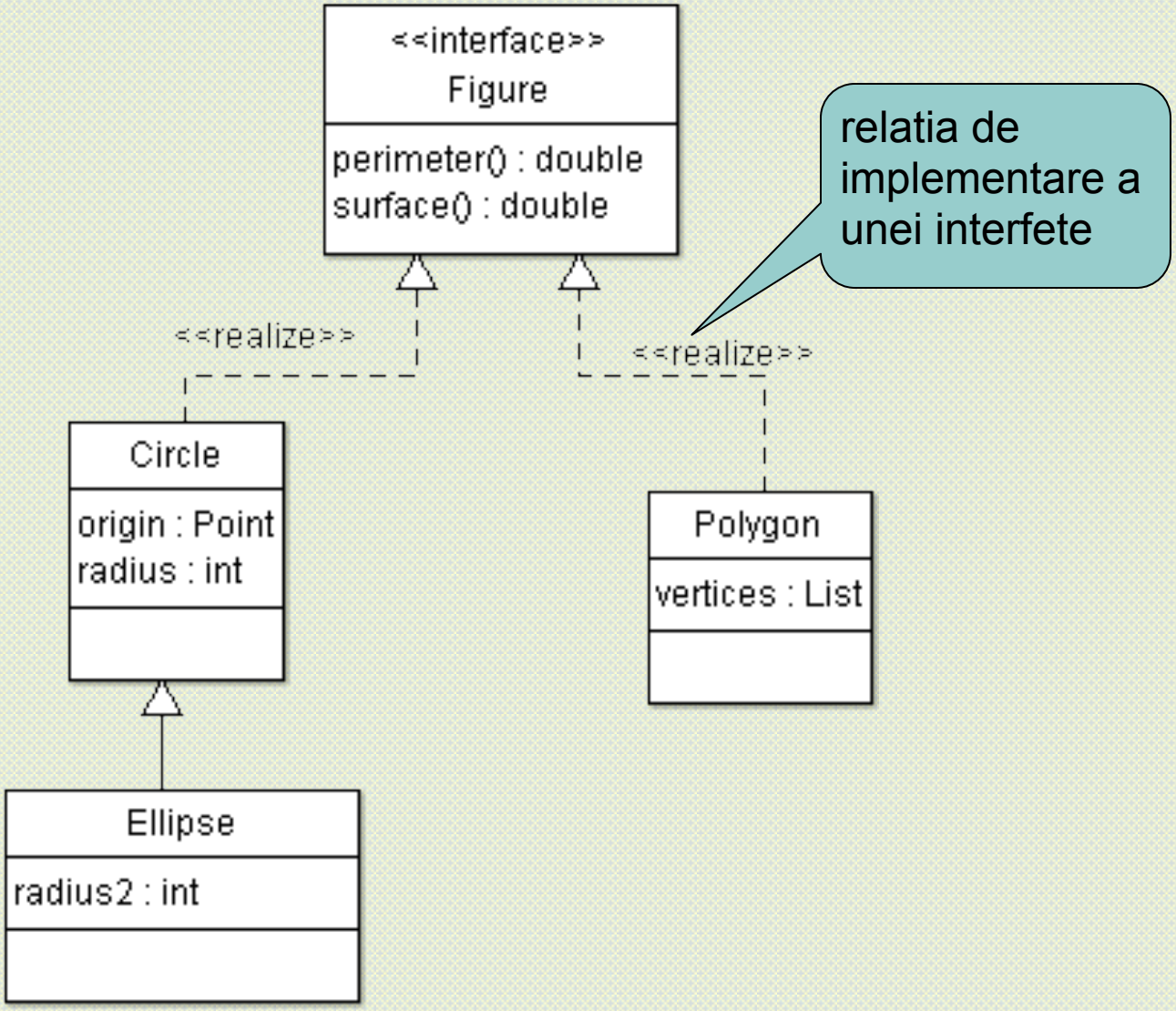
polimorfism prin suprascriere si legare dinamica

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

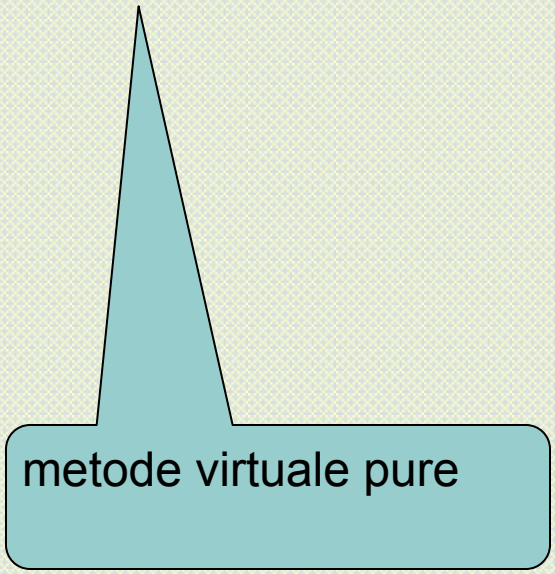


Interfata in C++

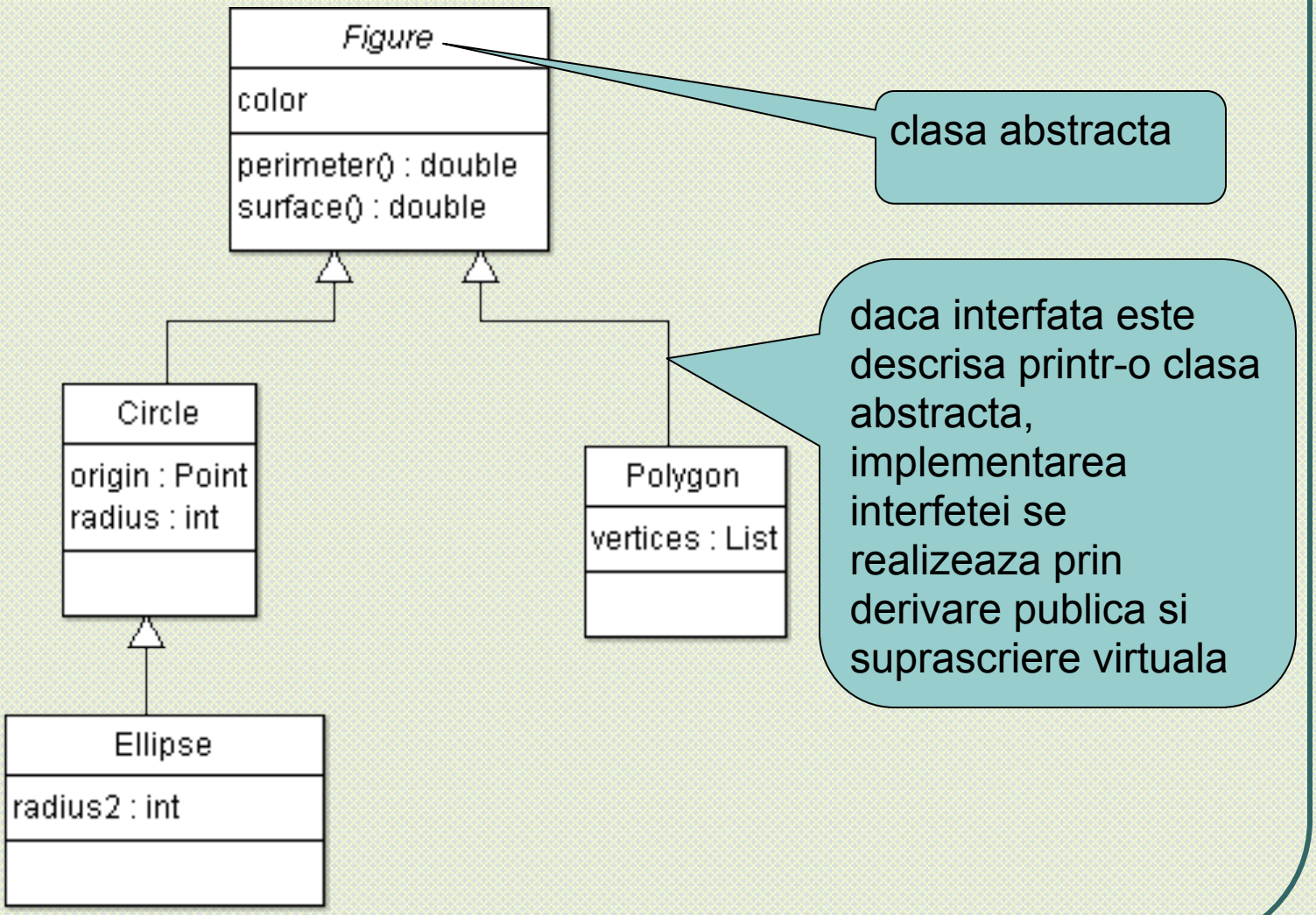
- interfetele in C++ sunt descrise cu ajutorul claselor abstracte
- o **clasa abstracta** nu poate fi instantiata, i.e., nu are obiecte
- de notat totusi ca **interfata si clasa abstracta sunt concepte diferite**
  - o clasa abstracta poate avea date membre si metode implementate
- in C++ o clasa este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

Clase abstracte in C++

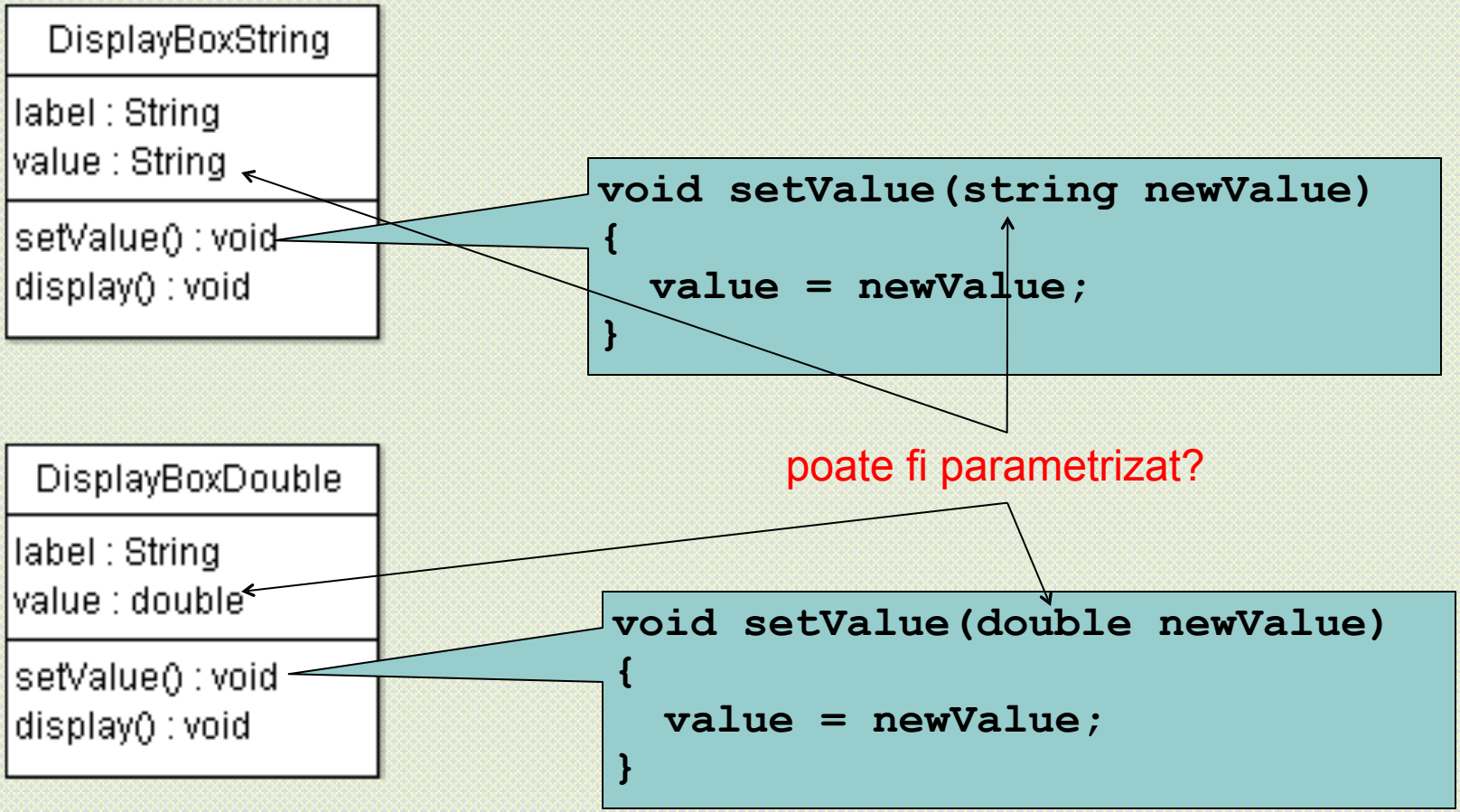
```
class Figure {
public:
   ...
   virtual void perimeter() = 0;
   virtual void surface() = 0;
   ...
};
```



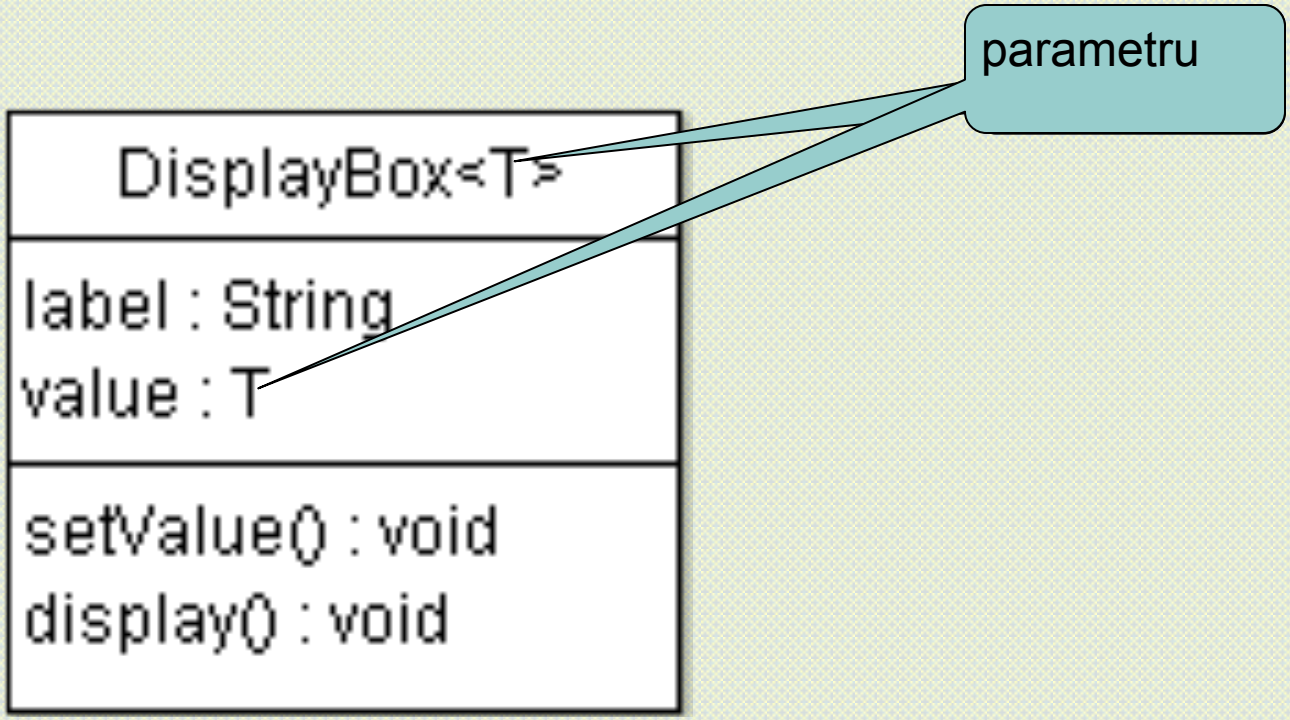
Diagrame cu clase abstracte



Abstractizare prin parametrizare

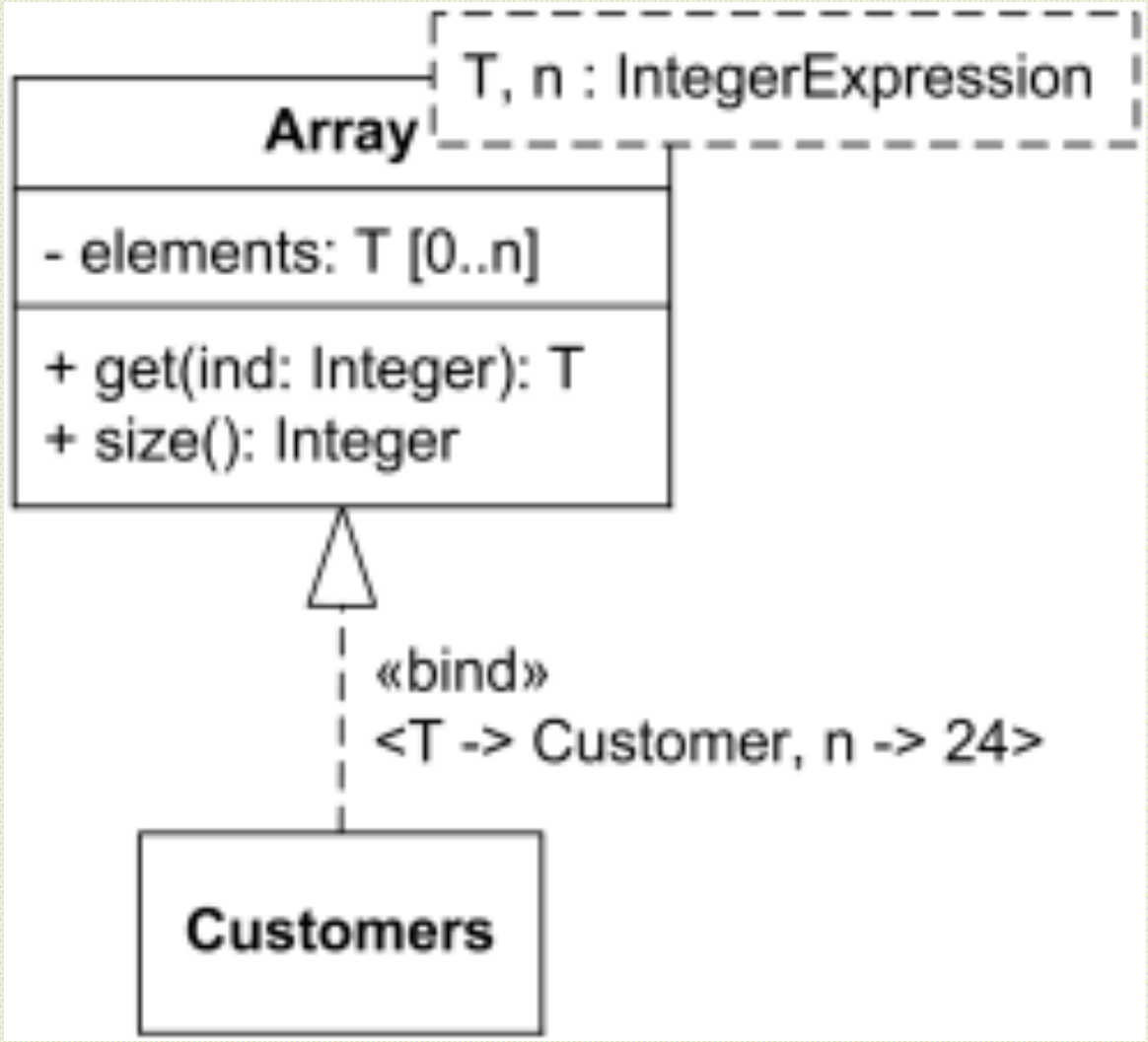


Clase parametrizate



Obs. Aceasta nu este chiar o notatie UML, Argouml nu are implementata notatia pentru clase parametrizate

Alt exemplu (https://www.uml-diagrams.org/template.html)



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;
}
```

declaratie parametri

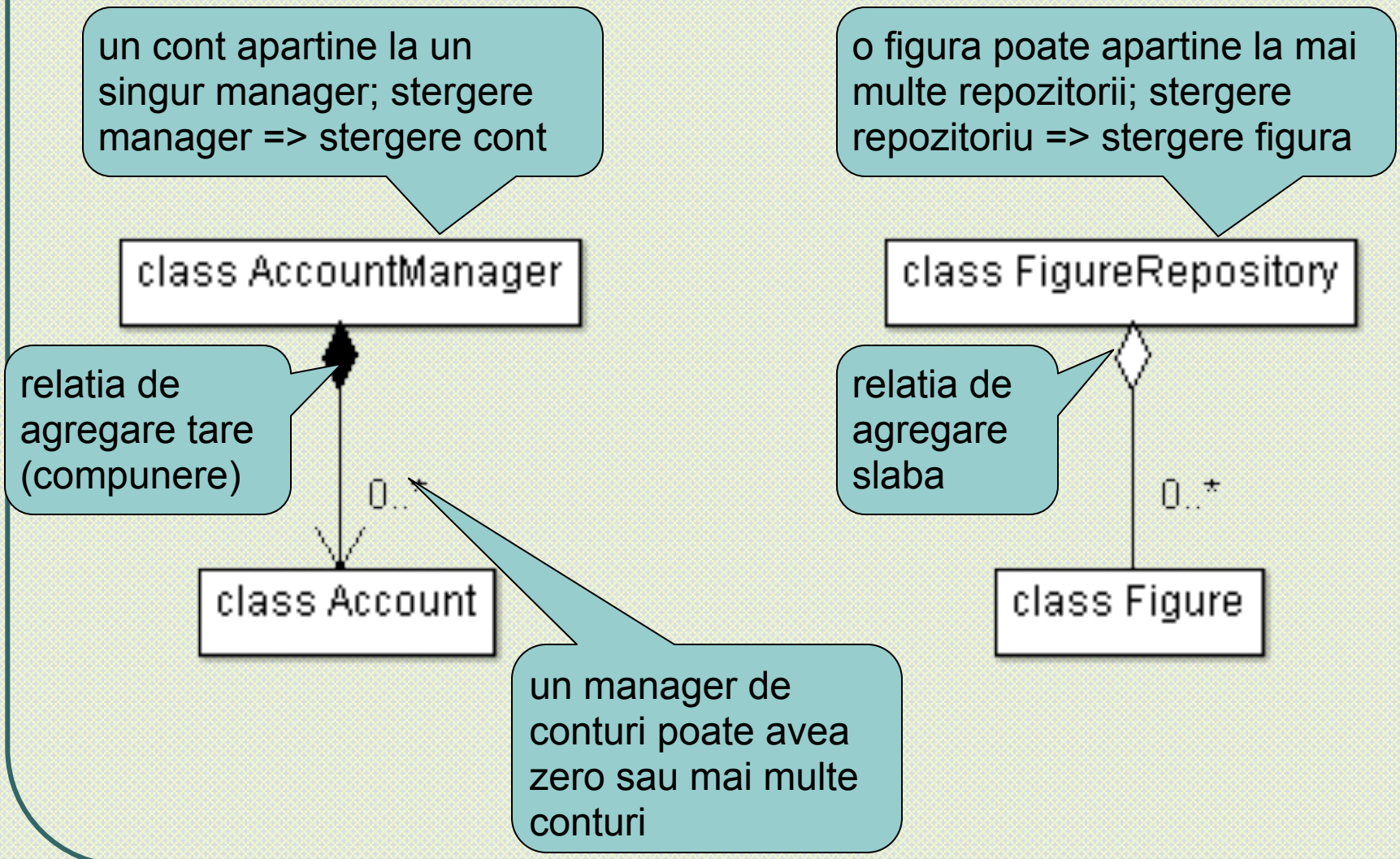
definitii parametrizate

utilizare parametri

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)

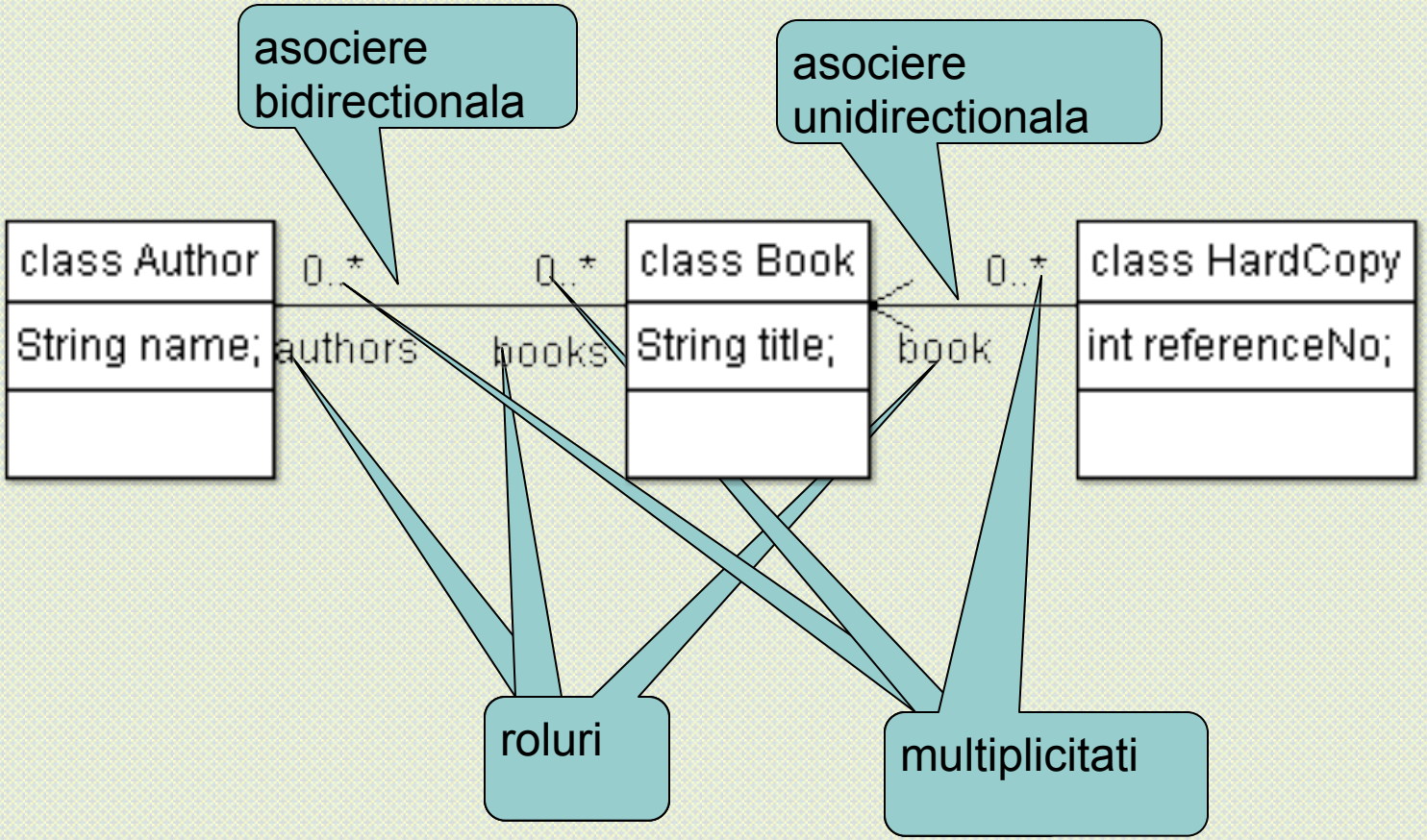
```
#include <list>
...
class FigureRepository {
private:
 list< Figure* > figures;
};
```

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;
   ...
};
```

PRINCIPII

PRINCIPIUL SUBSTITUIRII AL LUI LISKOV

Barbara Liskow

- profesor la MIT, a primit premiul Turing pentru 2008 pentru contributii aduse la dezvolatrea limbajelor de programare, in special OO
-

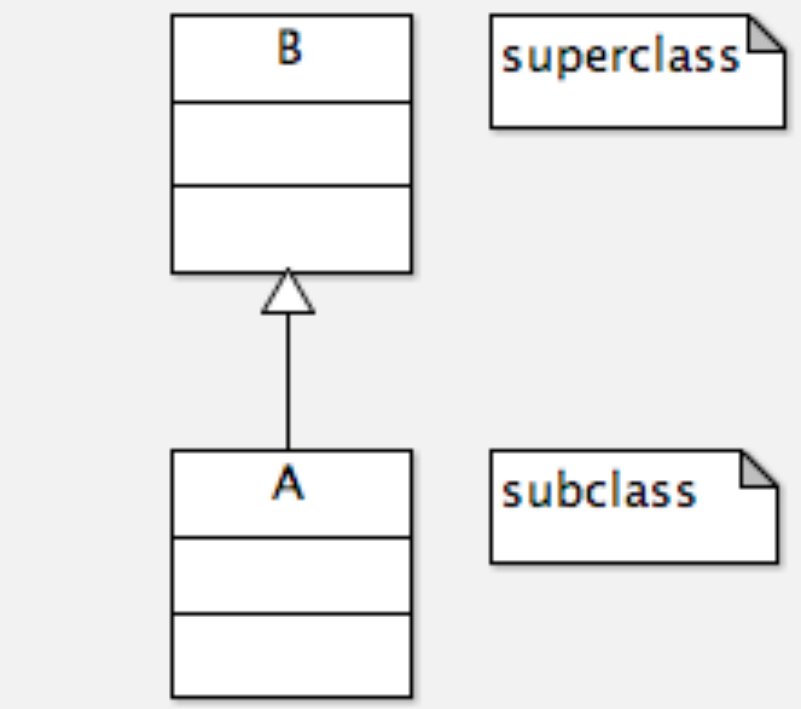
Clase versus Tipuri

- o clasa este o implementare a unui tip (de date)
- tipul definit de o clasa este dat de interfata (membrii publici)
- atat clasa cat si tipul de date au igrediente similare
  - atribute
  - operatii
- exista totusi o diferenta subtila: nu este posibil sa avem doua instante ale unui tip de data cu aceleasi valori

Relatia de subtip: Principiul substituirii al lui Liskov

- intutiv
  - S este un subtip al lui T (scriem S <: T) daca un obiect al lui S poate “juca rolul” unui obiect al lui T in orice context
- Formularea originala (impreuna cu J. Wing)
  - “Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.”

Realatia super-clasa - sub-clasa



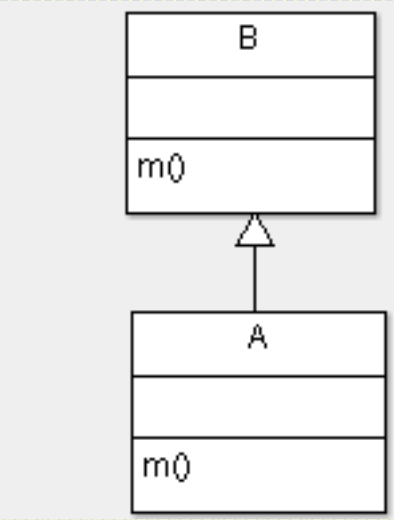
- Intrebare: tipul definit de A este subtip al tipului definit de B?
- Raspuns: nu intotdeauna relatia de subclasa implica relatia de subtip

Principiul substituirii al lui Liskov

- o formulare cu clase:
  - “If class A is a subtype of class B, then a reference of type A should be able to appear in any context where a reference of type B is expected without altering the correctness of the program.”

Suprascrierea de metode

- In cazul ierarhiilor de clase, suprascrierea metodelor poate duce la violarea principiului
-



Design by contract

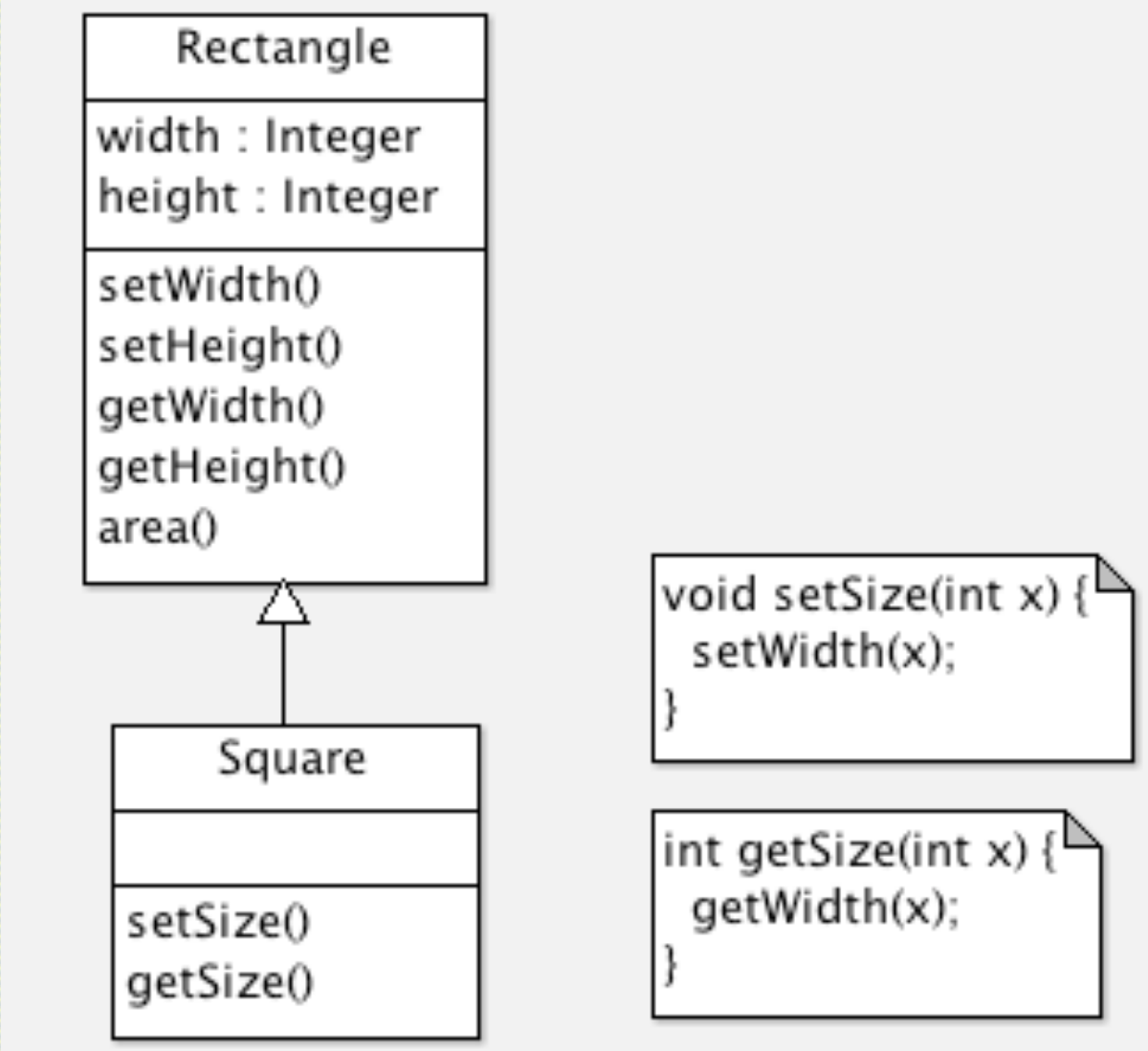
- Bertrand Meyer, 1986
  - o metaforă clară pentru ghidarea procesului de proiectare
  - aplicabila in contextul ierarhiilor de clasa, în special formalismului de redefinire și legare dinamică
- Reamintim conceptul design by contract – pentru metode:
  - o metoda: “if you give me a state satisfying the precondition, I give you a state satisfying the postcondition”
  - preconditia = proprietatile ce trebuie sa le satisfaca datele de intrare
  - postconditia = proprietatile ce trebuie sa le satisfaca datele de iesire

Principiul substituirii al lui Liskov

- In terminologia “design by contract”
  - preconditia unei metode suprascrise in clasa subtip (copil) nu trebuie sa fie mai tare decat cea din clasa supratip (parinte)
  - postconditia unei metode suprascrise in clasa subtip (copil) nu trebuie sa fie mai slaba decat cea din clasa supratip (parinte)
  - invariantii clasei supratip (parinte) trebuie pastrati de clasa subtip (copil)

- relatia de generalizare/specializare din UML si relatia de derivare din C++ nu definesc totdeauna o relatie de subtip (comportamental)

Exemplu: un patrat este un dreptunghi?



Exemplu: un patrat este un dreptunghi?

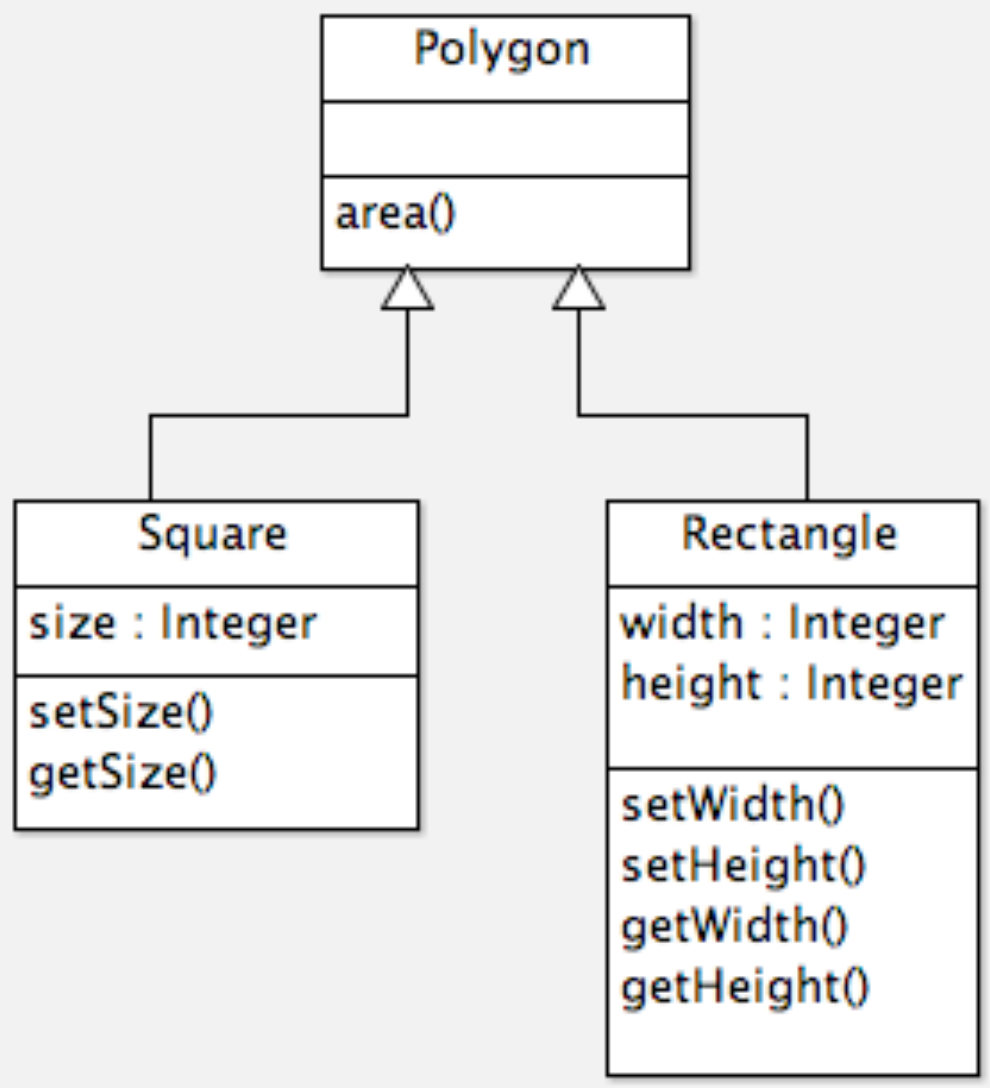
```
int Square::area() {
  return getSize() * getSize();
}

Square s;
s.setWidth(5);        // metoda mostenita
s.setHeight(10);    // metoda mostenita
```

Mai este pastrat invariantul de la dretptunghi

s.area() = s.getWidth() * s.getHeight() ?

Exemplu: patrat sau dreptunghi?



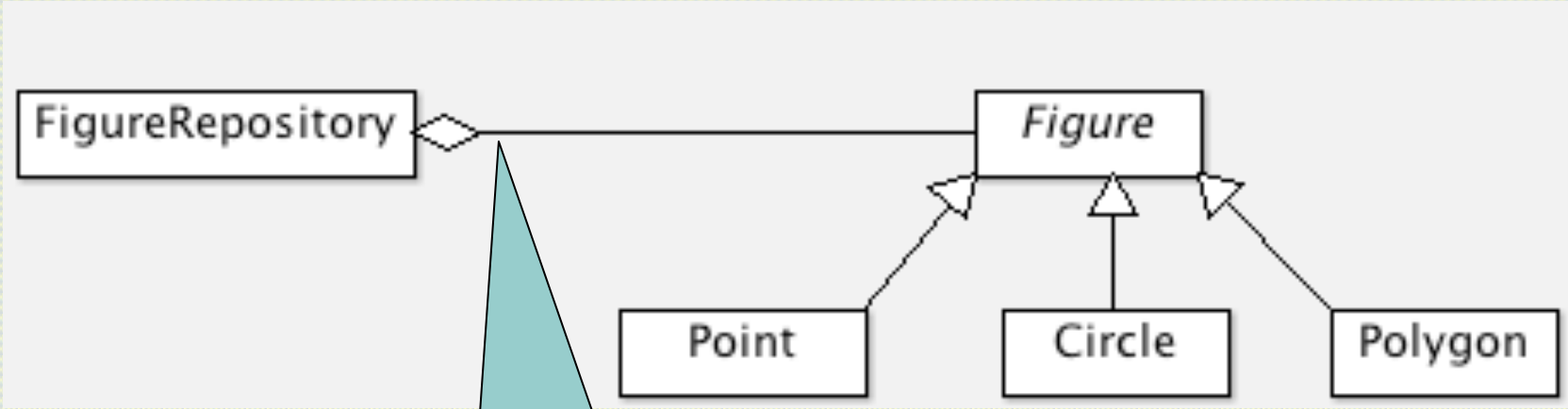
PRINCIPII

PRINCIPIUL DE INVERSARE A DEPENDENTELOR

Principiul de inversare a dependentelor

- A. “Modulele de nivel inalt nu trebuie sa depinda de modulele de nivel jos. Amandoua trebuie sa depinda de abstractii.”
- B. “Abstractiile nu trebuie sa depinda de detalii. Detaliile trebuie sa depinda de abstractii.”
- programele OO bine proiectate inverseaza dependenta structurala de la metoda procedurala traditionala
  - metoda procedurala: o procedura de nivel inalt apeleaza o procedura de nivel jos, deci depinde de ea

Principiul de inversare a dependentelor



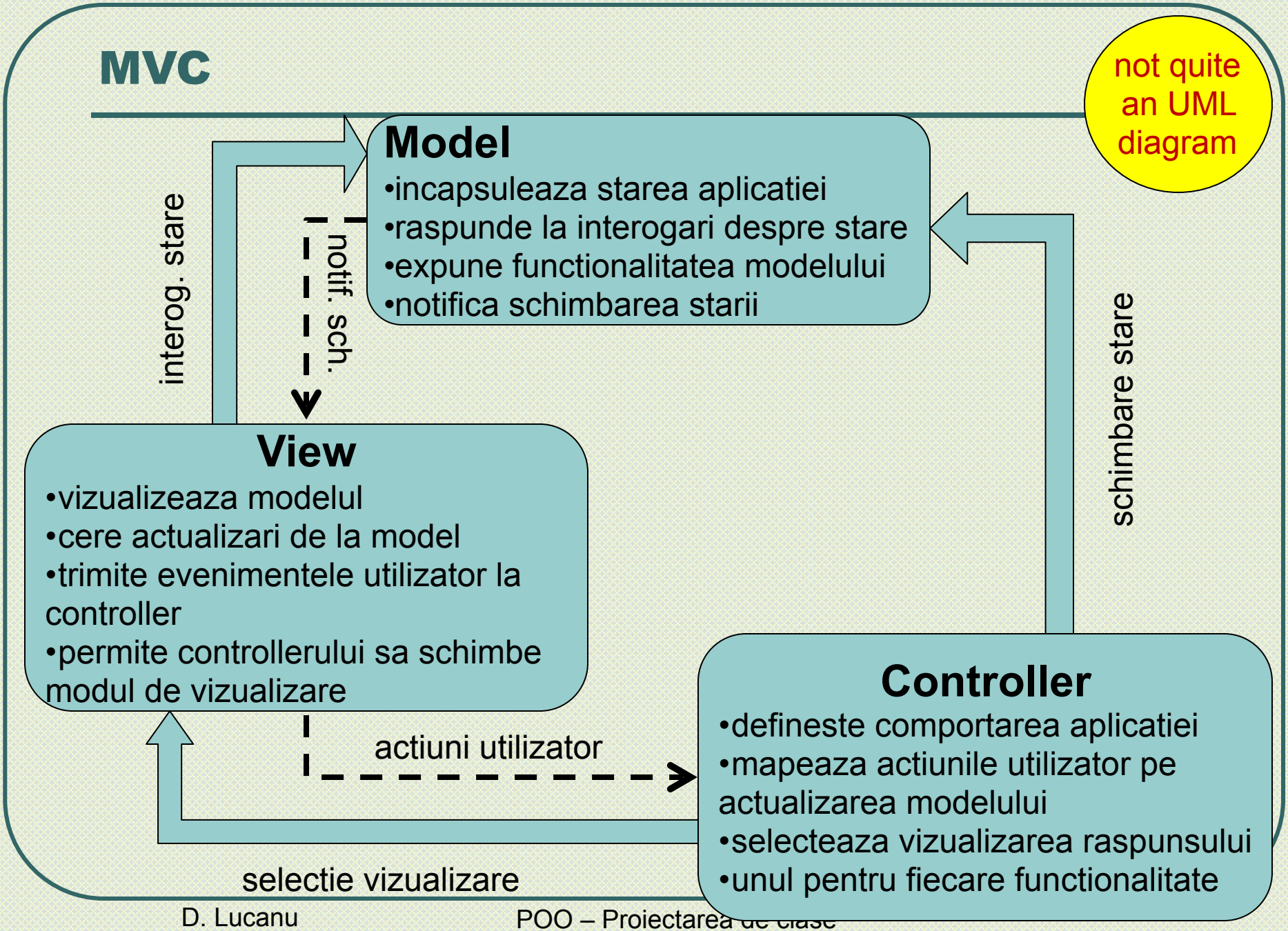
FigureRepository nu trebuie sa depinda de modul particular de definire a figurilor (puncte, cercuri, poligoane...); toate depind de abstractia *Figure*

MVC

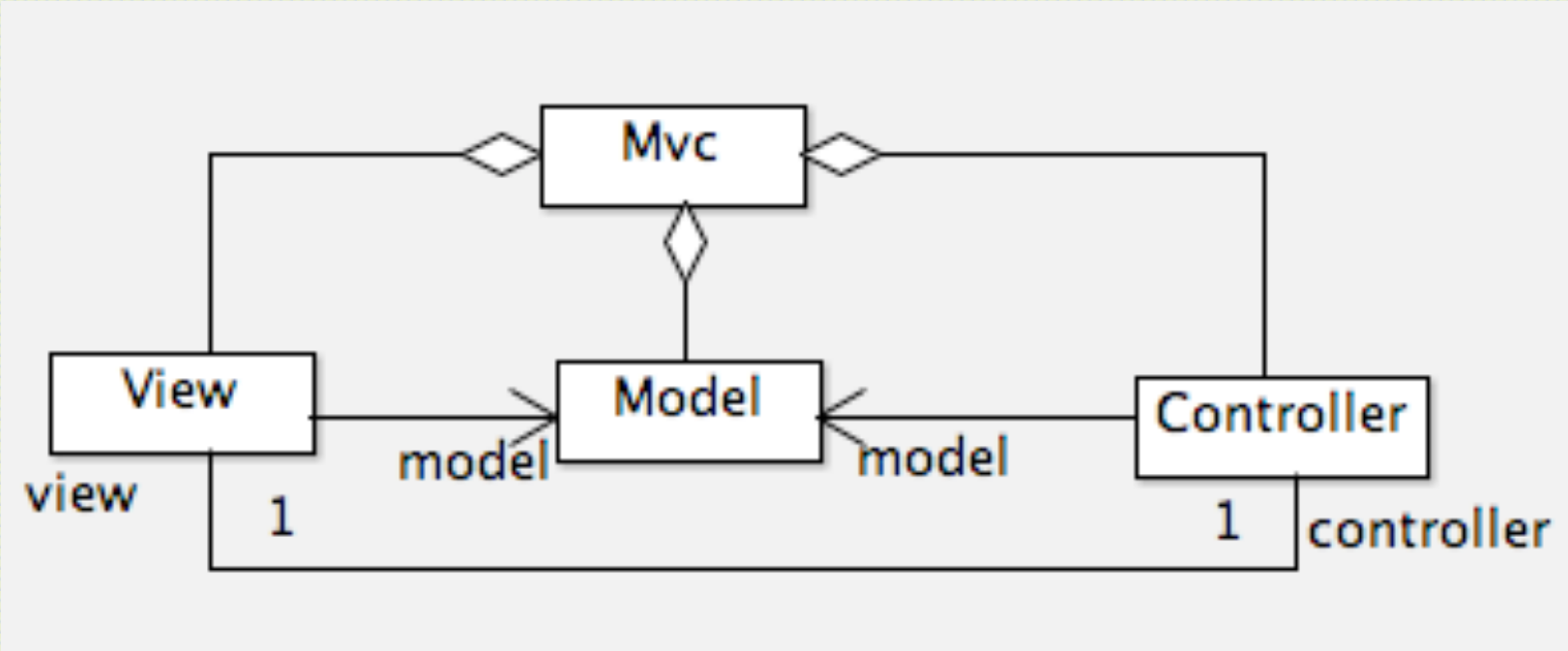
Cum construim o aplicatie OO?

- 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* (sabloanele) 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

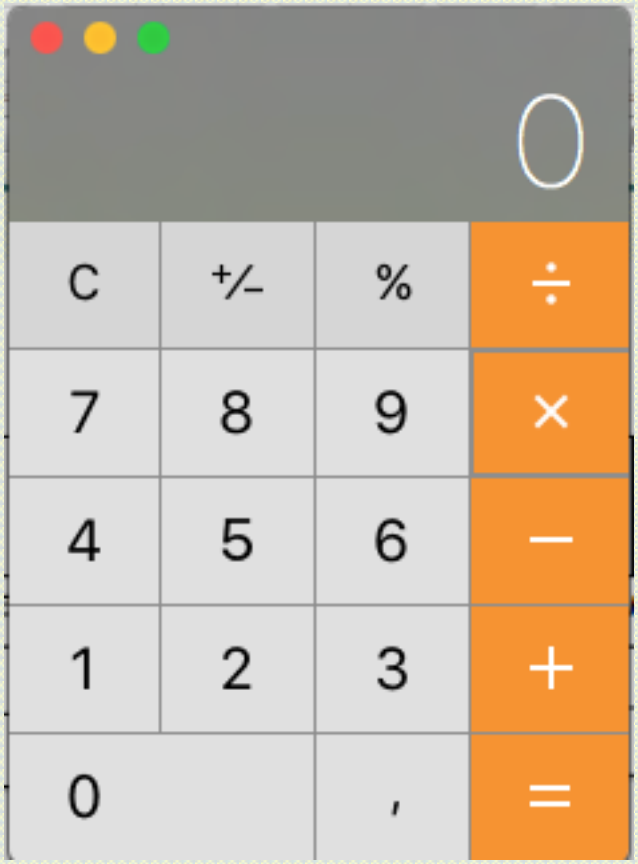
MVC


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 actualizarea modelului
•selecteaza vizualizarea raspunsului
•unul pentru fiecare functionalitate
selectie vizualizare

MVC – modelarea cu clase



MVC – studiu de caz



MVC – studiu de caz

- o mica aplicatie care simuleaza operatiile unui minicalculator
- model = calculatorul
- view
  - vizualizare (text) meniu Calculator
- controller
  - preia optiunile din meniu ale utilizatorului si le transpune asupra modelului

MVC – studiu de caz

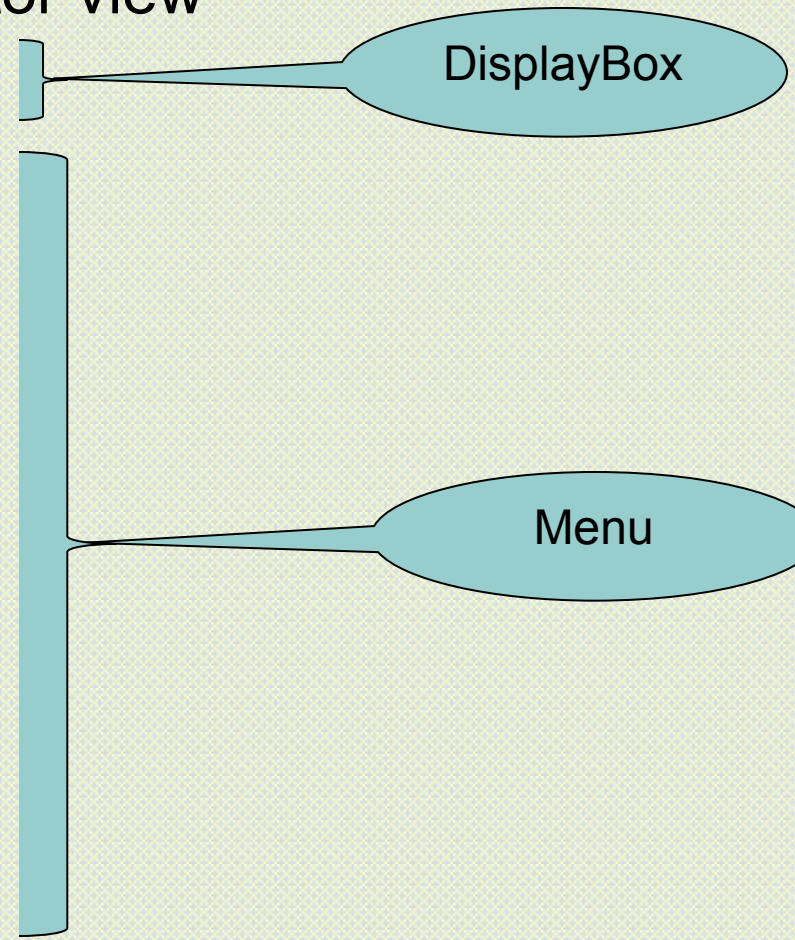
*** Calculator view ***

Result: 0

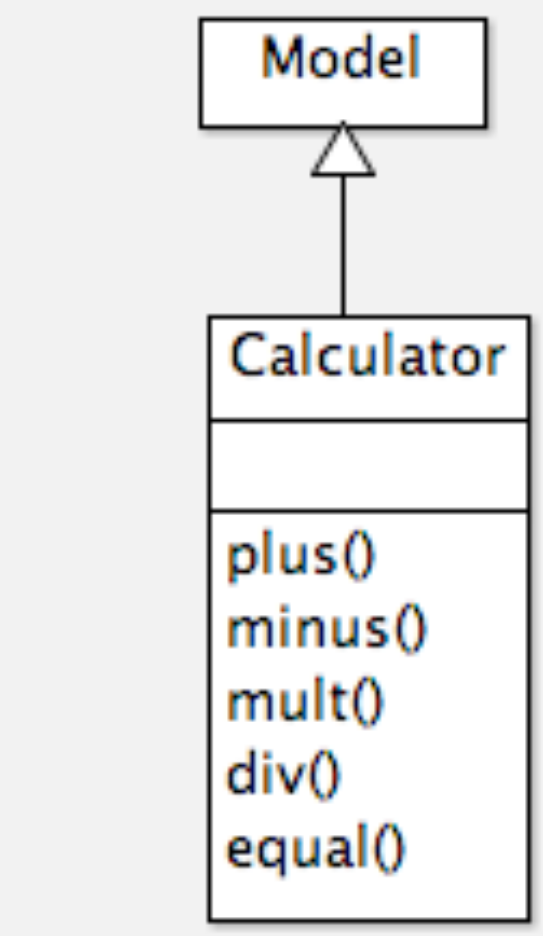
Menu:

1. Plus
2. Minus
3. Mult
4. Div
5. Equal
6. Value
0. Exit

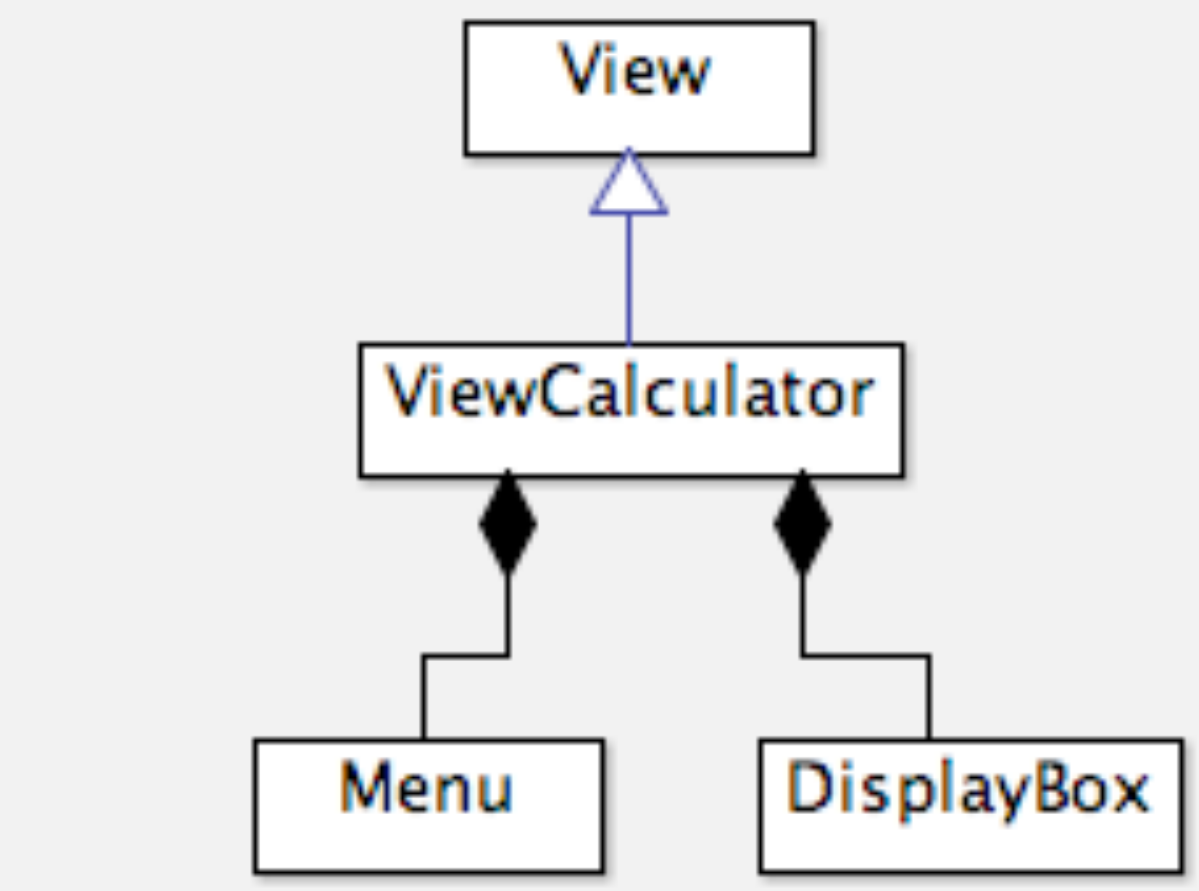
Option:



Modelul



clasa ViewCalculator



clasa ControllerCalculator

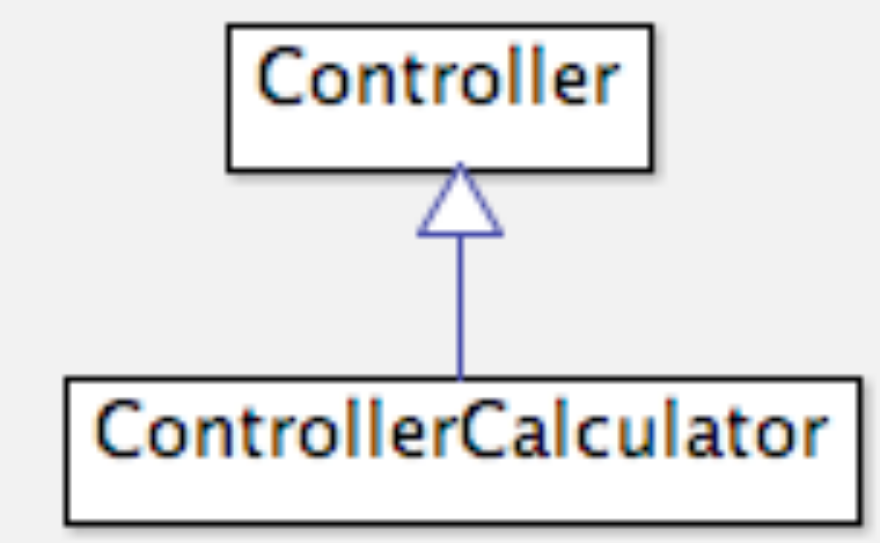
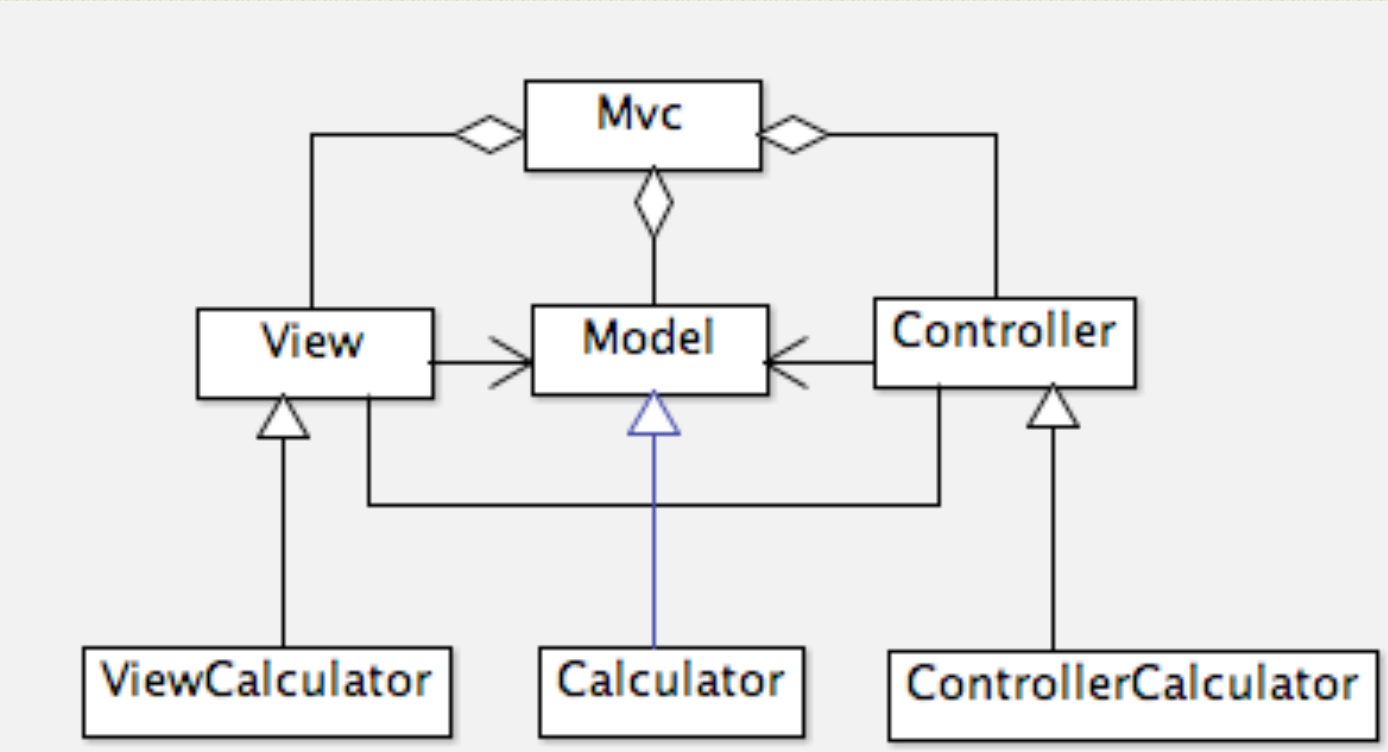


Diagrama MVC revizuita



S-a respectat principiul inversarii dependentelor!