

# Limbajul C/C++

## Moștenire multiplă



Curs 13

# Relații între clase

- ❑ Conceptele nu există izolate. Ele **coexistă și interacționează**.
- ❑ La elaborarea modelului obiectual al unei aplicații se disting următoarele etape:
  - ❑ **Identificarea claselor** → corespund conceptelor aplicației - substantive
  - ❑ **Stabilirea relațiilor dintre clase** → corespunde specificațiilor aplicației – verbe
- ❑ Tipuri de relații
  - ❑ Asociere o relație în care obiectele unei clase știu de obiectele altei clase (has-a)
    - ❑ **Dependentă** o relație în care obiectele unei clase utilizează de obiectele altei clase (use-a)
    - ❑ **Agregare** o relație parte întreg (is-part-of)
    - ❑ **Compoziție** este similară cu asocierea dar mai strictă (contains)
  - ❑ **Mostenire** (specializare) este o relație de generalizare - specializare (is-a, kind-of)

# Moștenire (Inheritance)

## ❑ Definiție

- ❑ Moștenirea este un mecanism care permite unei clase A să **moștenească** **attribute** și **metode** ale unei clase B. În acest caz obiectele clasei A au acces la membrii clasei B fără a fi nevoie să le redefinim

## ❑ Terminologie

### ❑ Clasă de bază

- ❑ Clasa care este moștenită

### ❑ Clasă derivată

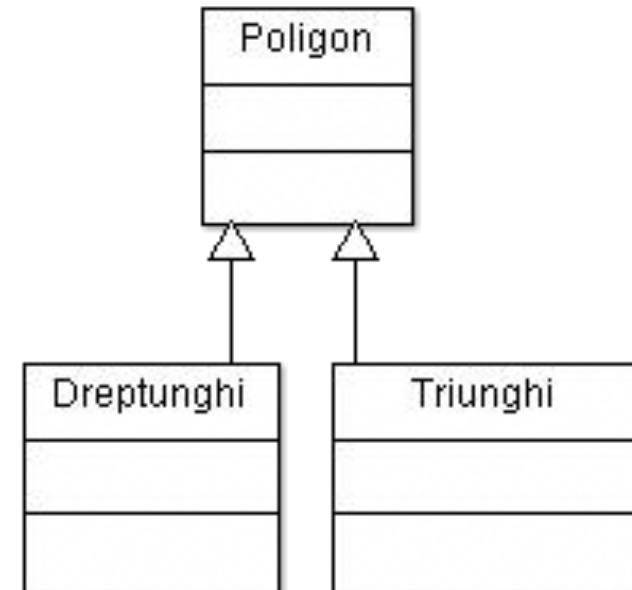
- ❑ O clasă specializată a clasei de bază

### ❑ Relația „**kind-of**” nivel de clasă

- ❑ Triunghiul este un tip (**kind-of**) Poligon

### ❑ Relația „**is-a**” nivel de obiect

- ❑ Obiectul triungiRosu este un (**is-a**) Poligon



# Moștenire

## ❑ Moștenire

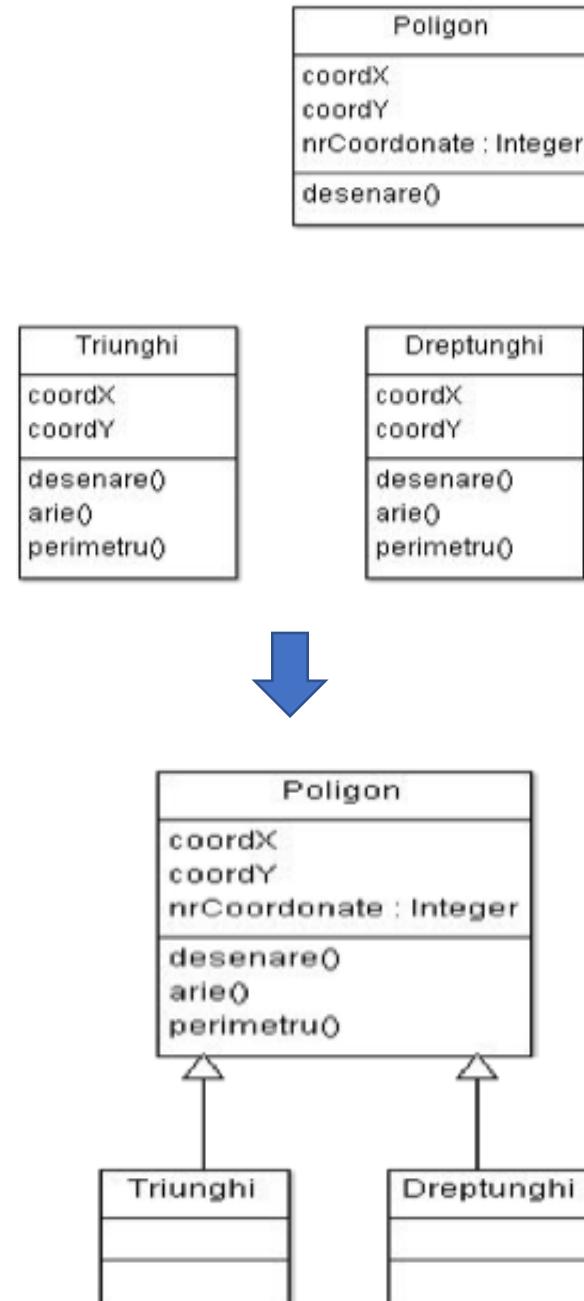
- ❑ O metodă care permite **refolosirea codului**
- ❑ Folosirea **moștenirii publice** pentru **polimorfism**
- ❑ Folosirea **moștenirii private** pentru **încapsulare**

## ❑ Polimorfism

- ❑ Funcții virtuale
- ❑ Folosirea de **pointeri la clasele de bază**
- ❑ Legătura statică/**dinamică**

## ❑ Run Type Information

- ❑ Regăsirea subtipului stocat în clasa de bază
- ❑ **dynamic\_cast<>**, **typeinfo**



# CUPRINS

❑ Runtime type information

❑ Moștenire multiplă

❑ Clase abstracte

# Runtime type information- RTTI

- Este o facilitate a limbajului care permite **identificarea tipului variabilelor la execuție**
- Pentru a functiona clasele trebuie să fie polimorfice, să conțină **cel puțin o funcție virtuală**
- Include
  - `dynamic_cast<>`
  - `typeid`
- Incluse in biblioteca `typeinfo`
  - `#include <typeinfo>`

# Runtime type information - RTTI

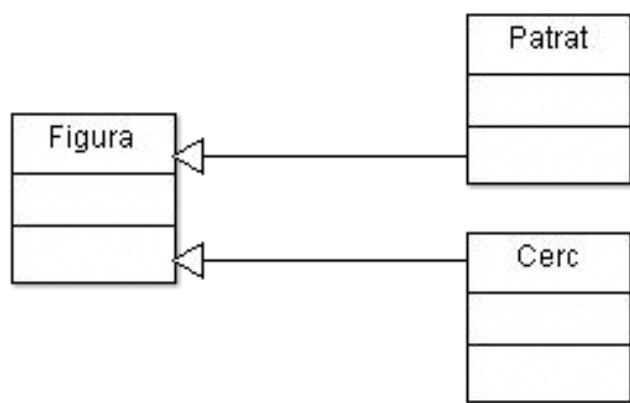
```
class Animal {
    virtual ~Animal();
};

class Mamifer : public Animal {
    virtual ~Mamifer();
};

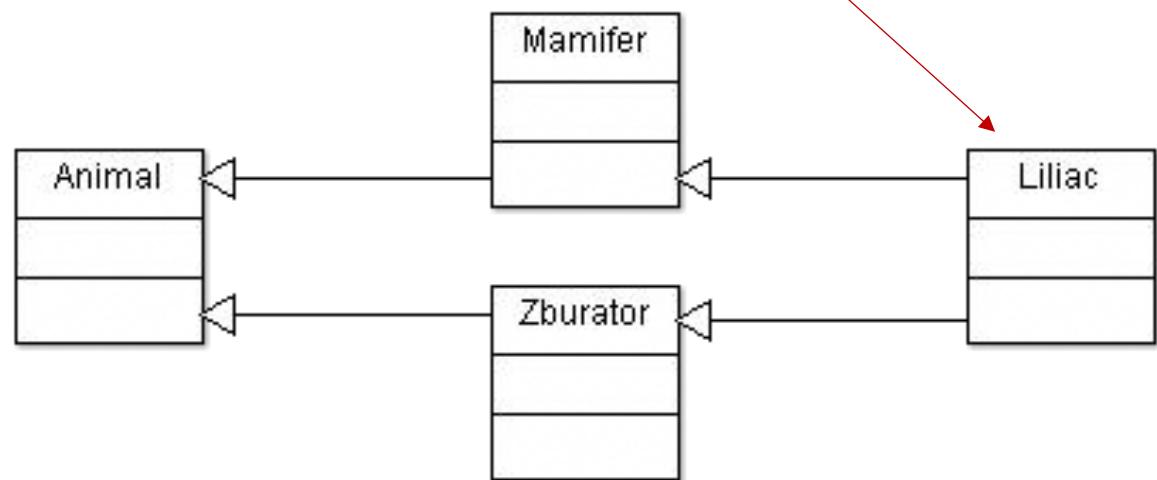
int main() {
    Animal a;
    Mamifer ;
    Animal *pa = &m;
    if (dynamic_cast<Mamifer*> (pa) != 0) {
        Mamifer *pm = (dynamic_cast<Mamifer*> (pa) ;
        Mamifer *p1=(Mamifer*) pa;
    }
}
```

# Tipuri de moștenire în C++

Moștenire Simplă  
O singură clasă de bază directă



Moștenire Multiplă  
Mai multe clase de bază directe



# Moștenire multiplă

## ❑ Definiție

❑ Moștenirea este multiplă dacă o clasă are două sau **mai multe clase de bază**

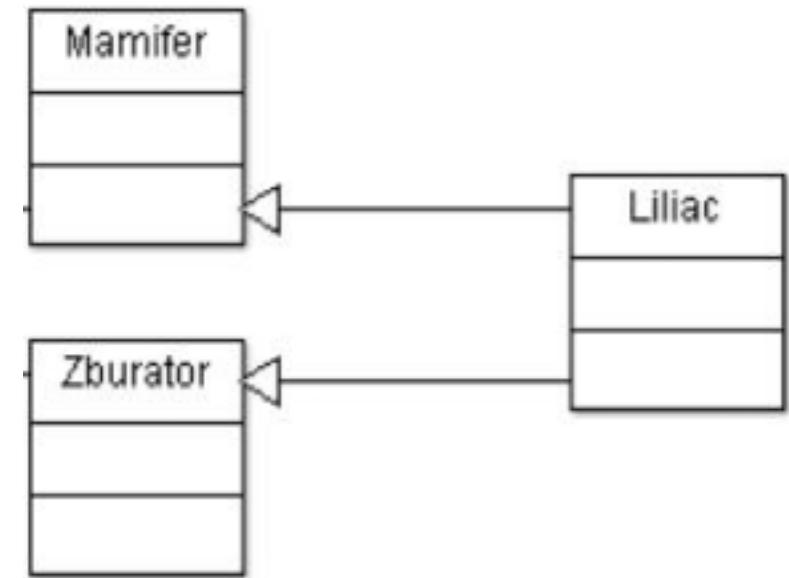
## ❑ Sintaxă

```
class ClasaDerivată : [modificatorDeAcces] ClasaDeBază1,  
                        [modificatorDeAcces] ClasaDeBază2,  
                        ...  
                        [modificatorDeAcces] ClasaDeBazăN {  
...  
};
```

❑ Crește flexibilitatea ierarhilor de clase → **ierarhii în formă de graf**

# Moștenire multiplă. Exemplu

```
class Mamifer {  
    ...  
};  
  
class Zburator {  
    ...  
};  
  
class Liliac: public Mamifer, protected Zburator {  
    ...  
    Liliac(...): Zburator(...), Mamifer(...) //constructor  
    ...  
}  
...  
};
```

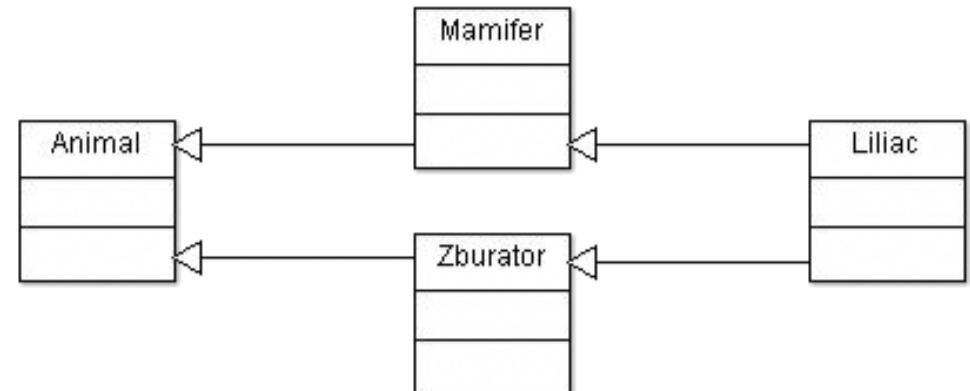


# Moștenire multiplă. Exemplu

```
class Animal {  
public:  
    int varsta;  
    ...  
};  
  
class Mamifer: public Animal {  
    ...  
};  
  
class Zburator: public Animal {  
    ...  
};  
  
class Liliac: public Mamifer,  
               protected Zburator {  
    ...  
};
```

```
int main(){  
    Liliac l;  
    l.varsta; //eroare accesare ambiguă  
    l.Mamifer::varsta=7;  
    l.Zburator::varsta=10;  
}
```

Ordine apelare constructori:  
-Animal, Mamifer, Animal, Zburator, Liliac



# Moștenire multiplă

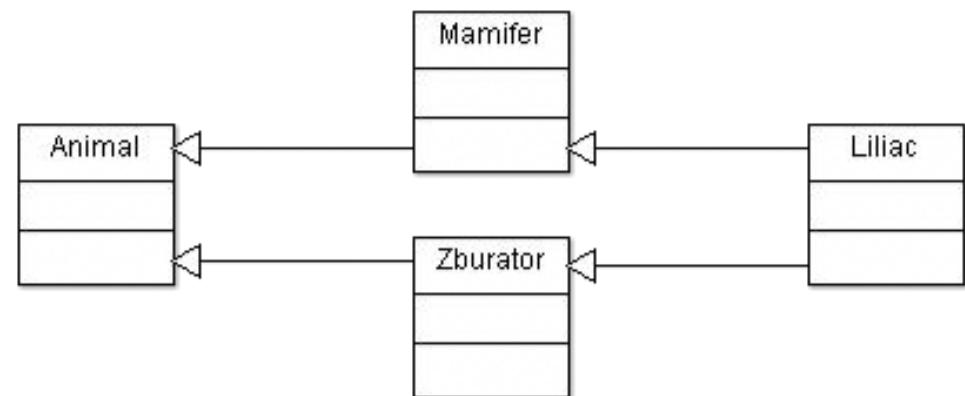
## ❑ Problema diamantului

❑ o instanță a clasei de bază Animal este **inclusă de două ori** clasa derivată Liliac (una de la clasa Mamifer și una de la clasa Zburator), ceea ce duce la:

- ❑ Pierderi la alocarea spațiului de memorie (toate atributele sunt duplicate)
- ❑ Ambiguități - probleme de accesare a membrilor clasei Animal

## ❑ Rezolvare

❑ Clase de bază virtuale



# Clase de bază virtuale

## ❑ Definiție

❑ Dacă o clasă de bază este declarată clasă de bază virtuală, atunci într-o ierarhie de tip diamant clasa de bază este instanțială **o singură dată**

## ❑ Sntaxă

```
class clasaDerivată :  
    [modificatorDeAcces] virtual clasaBaza {  
    . . .  
}
```

# Clase de bază virtuale

```
class Animal { public: int varsta; ... };

class Mamifer: public virtual Animal {
    Mamifer(...):Animal(...) {
        ...
    }
};

class Zburator: public virtual Animal { ... };

class Liliac: public Mamifer, protected Zburator {
    Liliac(...): Zburator(...), Mamifer(...), Animal(...) {
        ...
    }
};

int main() {
    Liliac l;
    l.varsta;
}
```

*Ordine apelare constructori:  
-Animal, Mamifer, Zburator, Liliac*

*Apel explicit al constructorului clasei de bază,  
NU se mai realizează implicit*

- ❑ Constructorul clasei de bază trebuie apelat explicit
- ❑ Pași pentru inițializarea unui obiect
  - ❑ Apelează constructorul clasei de bază **virtuale**
  - ❑ Apelează constructorii claselor de bază în **ordinea declarării lor**
  - ❑ Inițializarea membrilor clasei derive
  - ❑ Inițializarea obiectului **derivat**

# Moștenire multiplă

Nu toate limbajele de programare suportă moștenire multiplă

Cauza

problema diamantului

Soluție

Utilizare interfețe

Interfețe

Clase care conțin **doar declarații** de metode (metodele nu au implementare)

Are C++ definit conceptul de interfață?

NU

Putem defini metode în clase care nu sunt definite?

DA

Folosind funcții pur virtuale

# Functii pur virtuale

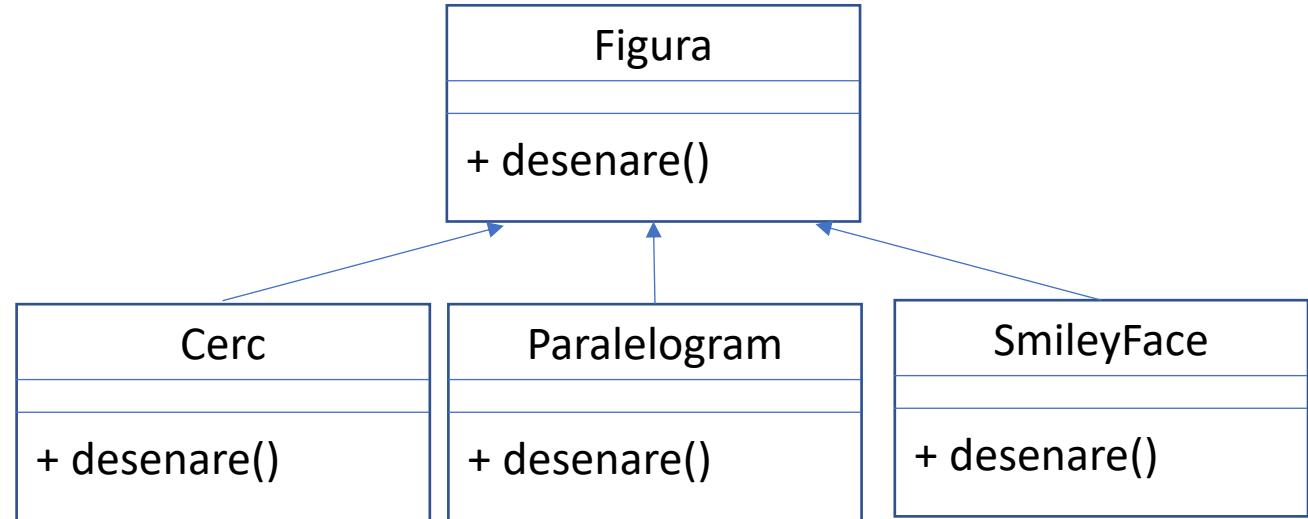
❑ Fie următoarea clase

- ❑ Figură
- ❑ Cerc
- ❑ Paralelogram
- ❑ Smiley Face

❑ Fiecare clasă conține o metodă de desenare

❑ Ce relații există între clase?

❑ Cum implementăm metoda de desenare?



Clasa Figură este clasă de bază pentru clasele Cerc, Palelogram și SmileyFace

Pentru clasa Figură știm cum să desenăm obiectul?

NU

# Functii pur virtuale

- ❑ Definiție
  - ❑ Sunt funcții care **sunt declarate virtuale**, dar **nu sunt implementate în clasa de bază**
  - ❑ Trebuie să fie suprascrise în toate clasele derivate, altfel rămân pur virtuale
- ❑ Sintaxă
  - ❑ **virtual tipDeReturn numeFunctie (listaDeParametri) = 0;**

# Clase abstracte

- ❑ Definiție
  - ❑ Dacă o clasă conține o funcție **pur virtuală** ea se numește **abstractă**
- ❑ Clasele abstracte **nu** pot fi **instantiate**
- ❑ Se pot utiliza pointeri la clasele virtuale
- ❑ Utile în cazul polimorfismului

# Clase abstracte. Exemplu

```
class A {  
public:  
    virtual void x() = 0;  
    virtual void y() = 0;  
};  
class B : public A {  
public:  
    void x() { ... }  
};  
class C : public B {  
public:  
    void y() { ... }  
};
```

Funcțiile x() și y() sunt pur virtuale =>  
Clasă abstractă

Funcția y() nu implementată => y()  
funcție pur virtuală => Clasă abstractă

Toate funcțiile virtuale sunt implementate  
=> clasa poate fi instantiată

```
int main () {  
    A * ap = new C;  
  
    ap->x ();  
    ap->y ();  
  
    delete ap;  
    return 0;
```

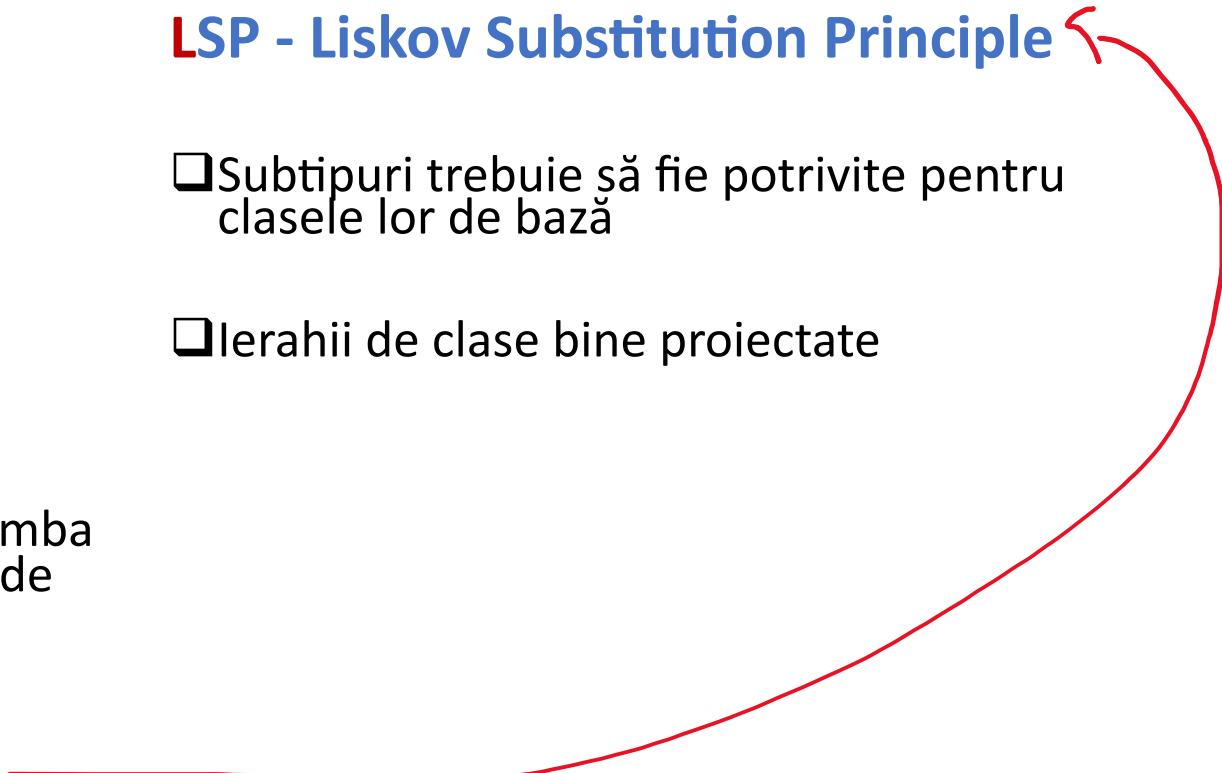
# SOLID – Moștenire

## OCP - Open-close Principle

- Clasele ar trebui să fie deschise pentru extensii dar închise pentru modificare
- Permite schimbări fără a modifica codul existent
- Folosirea moștenirii pentru a extinde/schimba codul funcțional existent și a nu se atinge de codul care funcționează
- Poate fi implementat și prin intermediul compozиiei

## LSP - Liskov Substitution Principle

- Subtipuri trebuie să fie potrivite pentru clasele lor de bază
- Ierarhii de clase bine proiectate



# OCP- Open-close Principle



```
class Figura {  
    int tip;  
    void desenarePoligon () { /*... */ }  
    void desenarePunct () { /* ... */ }  
public:  
    void desenare();  
};  
void Figura:: desenare() {  
    switch(tip) {  
        case POLIGON: desenarePoligon ();  
                        break;  
        case PUNCT: desenarePunct ();  
                        break;  
    } }
```



```
class Figura {  
public:  
    virtual void desenare() = 0;  
};  
class Poligon : public Figura {  
public:  
    void draw();  
};  
class Punct : public Figura {  
public:  
    void draw();  
};  
void Poligon::draw() { /* ... */ }  
void Punct::draw() { /* ... */ }
```

Ce se întâmplă dacă adaugăm un nou tip de figură?

# LSP- Liskov Substitution Principle

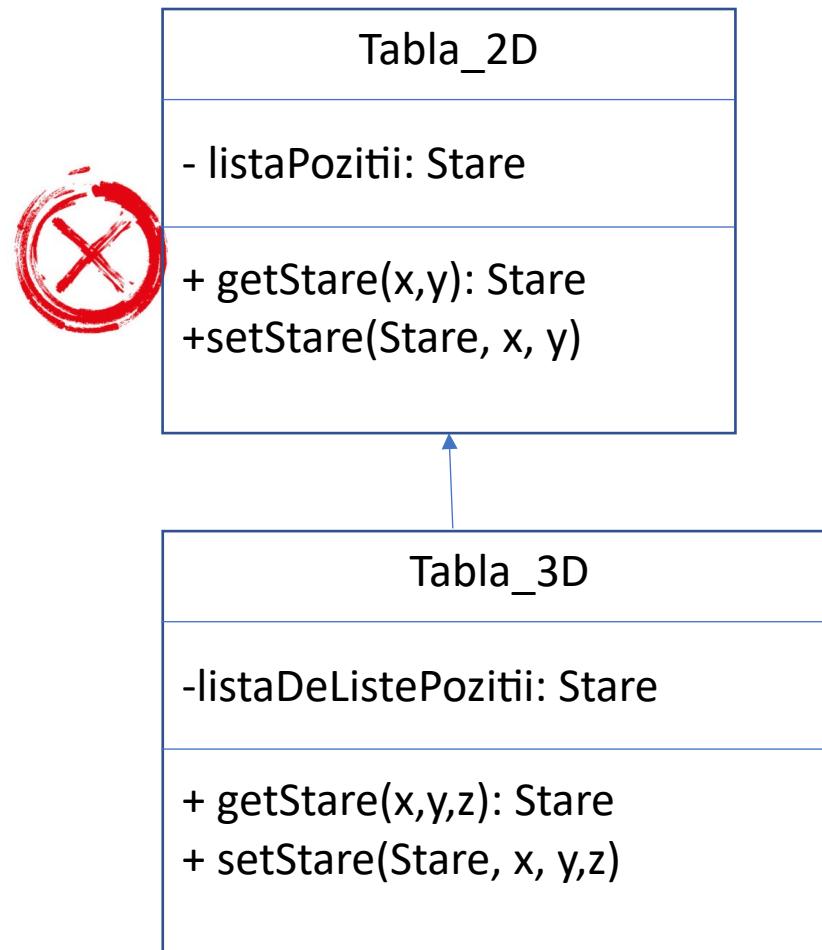


tabla = Tabla\_3D()  
tabla.getStare(4, 5) // nu are sens pentru table 3D

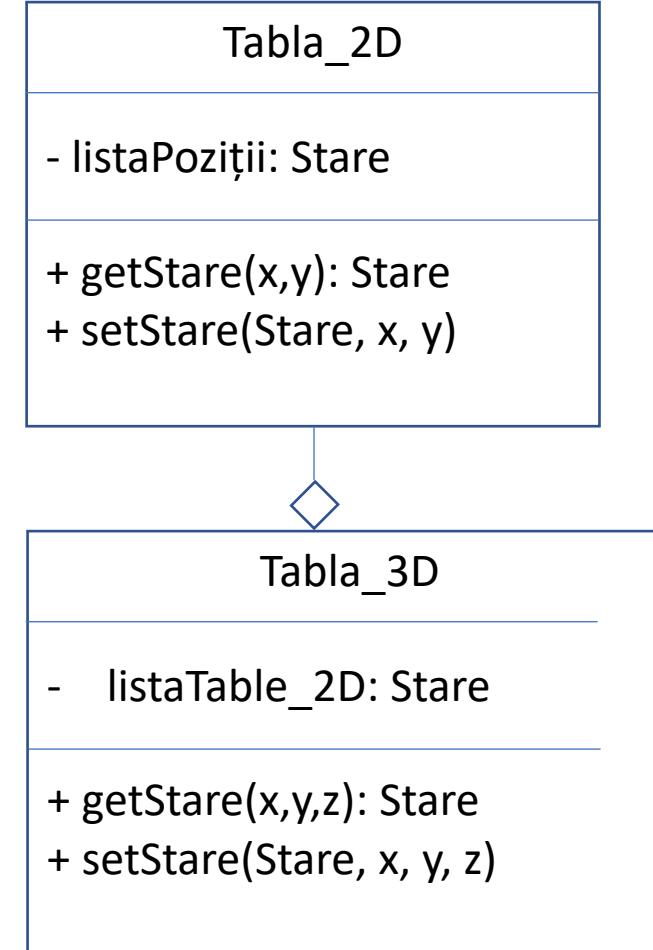


tabla = Tabla\_3D()  
tabla.getStare(1, 4, 5)



ÎNTREBĂRI