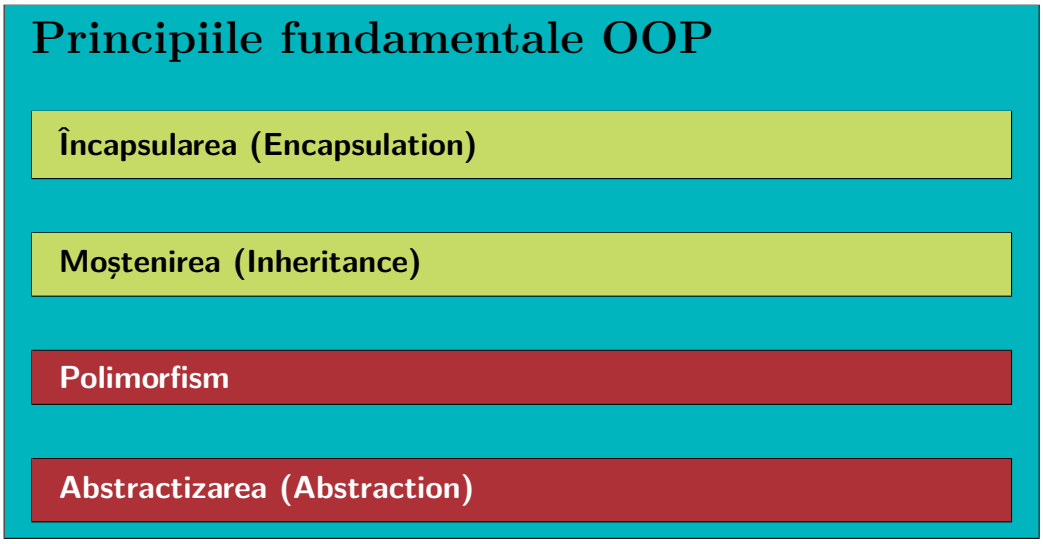
Polimorfism și Abstractizare

Vîlculescu Mihai-Bogdan

12 aprilie 2020

Contents

1 Introducere



Până acum am studiat 2 dintre cele 4 **principii fundamentale** ale programării orientate pe obiect:

- **Încapsularea (Encapsulation)**
- **Moştenirea (Inheritance)**

Astfel, a rămas să parcurgem ultimele 2 principii: **polimorfismul** şi **abstractizarea**.

2 Polimorfism

Pe parcurs am atins tangenţial acest subiect, mai exact am abordat câteva forme de polimorfism.

2.1 Definiţii

Polimorfismul se referă la abilitatea unui obiect de a lua mai multe forme.

Mai concret, *polimorfismul* se realizează prin următoarele metode:

```
supraîncărcare / suprascriere de metode
supraîncărcare de operatori
metode virtuale
```

2.2 Exemplu (din viața reală)

Exemplu

Un bărbat, de-a lungul vieții, poate îndeplini mai multe funcții: iubit, soț, tată, bunic...

2.3 Tipuri de polimorfism

Polimorfism la rulare (Run time polymorphism)

Polimorfism la compilare (Compile time polymorphism)

2.3.1 Compile time polymorphism

Mai este cunoscut şi sub denumirea de **early binding**. Se realizează prin 2 modalităţi:

- **Supraîncărcare de metode** (method overloading)
- **Supraîncărcare de operatori** (operator overloading)

Method overloading

2 metode în aceeași clasă cu același nume

Parametri diferă (nr sau tipul lor)

Operator overloading

C++ permite dezvoltatorului să particularizeze **acţiunile unor operatori** pentru animte **tipuri de date**.

Acesta e un alt exemplu clasic de **polimorfism**, fiindcă același operator execută instrucțiuni diferite în funcție de tipul obiectului.

Operatori care NU pot fi supraîncărcaţi

- **Operatorul * (pointer)**
- **Operatorul :: (rezoluţie)**
- **Operatorul .**
- **Operatorul ?:**
- **Operatorul sizeof**
- **Operatorul typeid**

Exemplu de supraîncărcare:

```
class B {
    public:
        int x;

        B(int i = 10) { x = i; }
        B operator+(B b) {
            return B(x + b.x);
        }
};

int main() {
    B b1, b2(20);
    B b3 = b1 + b2;
    cout << b3.x; // 30

    return 0;
}
```

Operator overload

Am supraîncărcat **operatorul** +, astfel încât de fiecare dată când se face o adunare între 2 **obiecte de tip B** au loc următoarele corespondenţe:

```
a + b <-- se apelează metoda corespunzătoare operatorului +
a <-- obiectul care apelează metoda (this)
b <-- parametrul metodei
```

IMPORTANT

Majoritatea operatorilor se pot supraîncărca şi în afara clasei ca **funcţii obişnuite**.

În acest caz trebuie acordată atenţie nr de parametri. **O funcţie obişnuită** NU are **obiectul this**.

Astfel, operatorii supraîncărcaţi în afara clasei au **cu un parametru mai mult**.

```
class B {
    public:
        int x;
        B(int i = 10) { x = i; }
};

B operator+(B b1, B b2) {
    return B(b1.x + b2.x);
}

int main() {
    B b1, b2(20);
    B b3 = b1 + b2;
    cout << b3.x; // 30

    return 0;
}
```

Supraîncărcare operator (funcție)

2.3.2 Run time polymorphism

Mai este cunoscut şi sub denumirea de **late binding**. Se realizează prin 2 modalităţi:

- **Suprascriere de metode** (method overriding)
- **Metode virtuale**

Method overriding

2 metode cu același nume și aceiași parametri

O metodă se află în clasa de bază, alta în cea derivată

Metoda din clasa de bază poate fi accesată de către un obiect al clasei derivate prin operatorul de rezoluție

Virtual

Se folosește la suprascrierea unei metode din clasa de bază

Asigură apelarea metodei corecte dacă se realizează upcasting

Exemplu:

```
class Shape {
    public:
        void draw() {
            cout << "Drawing a shape";
        }

        virtual void erase() {
            cout << "Erasing a shape";
        }
};

class Square : public Shape {
    public:
        void draw() {
```

```
            cout << "Drawing a square";
        }

        void erase() {
            cout << "Erasing a square";
        }
};

int main() {
    Shape* s = new Square; // upcasting

    s->draw(); // Drawing a shape
    s->erase(); // Erasing a square

    return 0;
}
```

Exemplu metode virtuale

RTTI (Run Time Type Information)

- Un mecanism care expune informații despre tipul unui obiect la **execuție**.
- Este disponibil doar pentru clasele care conțin **cel puțin o metodă virtuală**.
- ***Exemplu:*** am vorbit data trecută despre operatorul

```
dynamic_cast => Deduce tipul real de la
             => upcasting(dreapta egalului)
             => al obiectului la execuție.
             => Are nevoie de cel puțin o
             => metodă virtuală.
```

3 Abstractizare

3.1 Introducere

3.1.1 Definiții

Abstractizarea (abstraction) se referă la ascunderea detaliilor implementării faţă de utilizatorul unei clase.

Adică, în momentul în care un developer foloseşte o clasă *implementată de altcineva*, acesta nu are acces decât la **metodele şi datele publice** ale acesteia, fără să aibă nevoie să vadă şi **implementările**.

3.1.2 Exemple (din viața reală)

Când folosim un **aparat de cafea**, nu avem nevoie să știm decât **pașii** pentru a-l folosi. Nu este nevoie să cunoaștem **structura sa internă**.

Când conducem o **mașină**, trebuie să cunoaștem doar **modalitatea de utilizare** a mașinii, nu este necesar să avem cunoștințe despre **structura sa internă**.

3.1.3 Exemple (development)

Când includem o **bibliotecă**, folosim **funcțiile / metodele** din acea bibliotecă, fără a fi nevoie să cunoaștem **implementarea** acestora.

Când scriem noi o clasă într-un **fișier de tip header**, cu extensia **.h**, în programul principal doar apelăm **metodele** publice, fără a fi nevoie să cunoaștem **implementarea** acestora.

Acum că am stabilit **CE** este *abstractizarea* şi **CÂND** o folosim în mod obişnuit, trebuie să urmărim **CUM** realizăm în practică acest lucru.

3.2 Clase abstracte

3.2.1 Introducere

METODE PUR VIRTUALE

O metodă pur virtuală este o metodă virtuală fără implementare.

CLASE ABSTRACTE

- **Clasele abstracte** sunt clase care conțin cel puțin o **metodă pur virtuală**.
- **NU** pot fi instanțiate.
- **Clasele lor derivate** pot fi instanțiate doar dacă au fost implementate toate **metodele pur virtuale**.

3.2.2 Exemplu

Să ne gândim la următoarea situaţie: vrem să modelăm mai multe clase de animale: **Câine, Pisică, ...**

Să studiem *ce au în comun pattern-urile:*

- **Orice animal** *mănâncă* şi *doarme*.
- **Câinele** *latră*.
- **Pisica** *miaună*.

Astfel, fiecare clasă derivată are **metodele sale specifice**. Clasa de bază va avea 2 **metode comune** tuturor derivatelor. **DAR**, implementarea diferă în funcţie de clasă.

Din această cauză, clasa de bază va fi o **clasă abstractă**.

```
class Animal {
    public:
        virtual void eat() = 0;
        virtual void sleep() = 0;
};

class Dog : public Animal {
    public:
        void eat() {
            cout << "Dog::eat()";
        }
        void sleep() {
            cout << "Dog::sleep()";
        }
        void bark() {
            cout << "Dog::bark()";
        }
};

class Cat : public Animal {
    public:
        void eat() {
            cout << "Cat::eat()";
        }
        void sleep() {
            cout << "Cat::sleep()";
        }
        void meow() {
            cout << "Cat::meow()";
        }
};
```

Exemplu clasă abstractă

Declarările de mai jos:

```
virtual void eat() = 0;
virtual void sleep() = 0;
```

Reprezintă **metode pur virtuale**. Acestea nu au implementare, pentru că, după cum am spus şi mai sus, am considerat că implementarea lor depinde de fiecare **clasă derivată** în parte.

3.2.3 Exemplu (int main)

Acum, să urmărim care sunt **declarările corecte** pentru structura claselor de mai sus.

```
int main() {
    Animal a; // eroare de compilare

    Dog d; // corect
    Cat c; // corect

    Animal* a1 = new Dog; // corect
    Animal* a2 = new Cat; // corect
    return 0;
}
```

Exemplu declarări pentru clase abstracte

4 Resurse

4.1 Introducere

- Cele 4 principii OOP

4.2 Polimorfism

- https://www.geeksforgeeks.org/polymorphism-in-c/
- https://beginnersbook.com/2017/08/cpp-polymorphism/
- https://www.geeksforgeeks.org/operator-overloading-c/
- RTTI: https://www.geeksforgeeks.org/g-fact-33/amp/
- https://www.bogotobogo.com/cplusplus/dynamic_cast.php

4.3 Abstractizare

- https://stackify.com/oop-concept-abstraction/
- https://www.geeksforgeeks.org/abstraction-in-c/
- Abstractizare vs Încapsulare
- OOP încapsulare şi abstractizare