Curs 2

Principiile Programării Orientate pe Obiecte

Programare Orientată pe Obiecte

Tehnici de programare
Programarea procedurală

- Modul în care este abordată programarea, din punct de vedere al descompunerii programelor
- Paradigme

Programarea procedurală

- prima modalitate de programare, încă frecvent folosită
- descompunerea programului în proceduri (funcţii) care sunt apelate în ordinea de desfăşurare a algoritmului
- sunt prevăzute posibilităţi de transfer a argumentelor către funcţii şi de returnare a valorilor rezultate
- limbajul Fortran: primul limbaj de programare procedurală .
- au urmat Algol60, Algol68, Pascal, iar C este unul din ultimele invenţii în acest domeniu.

Programarea modulară (structurată)

- accentul s-a deplasat de la proiectarea procedurilor către **organizarea datelor**, datorită creşterii dimensiunii programelor.
- stilul de programare este în continuare procedural
- datele şi procedurile sunt **grupate în module**, nu implică însă şi o asociere strictă între acestea
- **Modul**: o mulţime de proceduri corelate, împreună cu datele pe care le manevrează
- tehnică de ascundere a datelor *(data-hidin*g): posibilitatea de ascundere a unor informaţii definite într-un modul faţă de celelalte module.
- modularitatea şi ascunderea informaţiilor sunt caracteristici implicite în programarea orientată pe obiecte.

Programarea orientată pe obiecte

- programarea procedurală şi structurată: descriere a algoritmilor ca o secvenţă de paşi care duc de la datele iniţiale la rezultatul căutat.

- limbaje de programare orientate la o clasă concretă de probleme: sisteme de dirijare cu baze de date, modelare ş.a.

- a apărut necesitatea sporirii siguranţei programelor - interzicerea accesului neautorizat la date.

Programarea orientată pe obiecte

- dezvoltarea sistemelor *orientate pe obiecte*, bazate pe *programarea orientată pe obiecte* a cunoscut o amploare deosebită în anii 90

- programarea orientată pe obiecte presupune:
  1. determinarea şi descrierea claselor utilizate în program
  2. crearea exemplarelor de obiecte necesare
  3. determinarea interacţiunii dintre ele.

Modelul obiect

- Reprezintă aplicarea în domeniul programării a unei metode din tehnică (tehnologia orientată pe obiecte, care se bazează pe modelul obiect)
- Primele aplicaţii: limbajul Simula (a stat la baza Smaltalk), Object Pascal, C++, Clos, Ada, Eiffel
- Modelul obiect al unei aplicaţii implică patru principii importante:
  - **abstractizare;**
  - **încapsulare;**
  - **modularitate;**
  - **ierarhizare.**
- Modelul obiect: un concept unificator în ştiinţa calculatoarelor, aplicabil nu numai în programare, ci şi în arhitectura calculatoarelor, în proiectarea interfeţelor utilizator, în baze de date.

Programarea orientată pe obiecte

- ***Object-oriented programming****: metodă de programare în care programele sunt organizate ca şi colecţii de obiecte cooperante, fiecare dintre ele reprezentând o instanţă a unei clase, iar clasele sunt membre ale unei ierarhii de clase, corelate între ele prin relaţii de moştenire.*
- Se folosesc obiecte, nu algoritmi, ca unităţi constructive de bază.
- Fiecare obiect este o instanţă (un exemplar) al unei clase.
- Clasele sunt componente ale unei ierarhii de tip, fiind corelate între ele prin relaţii de moştenire.

Obs: Dacă lipseşte una din aceste caracteristici: ***programare prin abstractizarea datelor*** (o clasă este un tip de date abstract)

Limbaj de programare orientată pe obiecte

- Cerinţe:
  1. Suportă obiecte (instanţe ale unor clase), clasele fiind tipuri definite de utilizator (numite şi tipuri abstracte de date)
  2. Tipurile (clasele) pot moşteni atribute de la alte clase, numite clase de bază

- Dacă un limbaj nu suportă direct moştenirea între clase se numeşte limbaj de programare bazat pe obiecte *(object-based*), cum este, de exemplu, limbajul Ada.

Principii POO: Abstractizarea

- ignorarea unor aspecte ale informaţiei manipulate, adică posibilitatea de a se concentra asupra esenţialului
- identificarea similitudinilor între diferite entităţi, situaţii sau procese din lumea reală, concentrarea atenţiei asupra acestor aspecte comune şi ignorarea pentru început a detaliilor
- identificarea trăsăturilor caracteristice esenţiale ale unui obiect, care îl deosebesc de toate celelalte feluri de obiecte
- fiecare obiect în sistem are rolul unui “actor” abstract, care poate executa acţiuni, îşi poate modifica şi comunica starea şi poate comunica cu alte obiecte din sistem fără a dezvălui cum au fost implementate acele facilitaţi
- procesele, funcţiile sau metodele pot fi de asemenea abstracte

Principii POO: Încapsularea

- ascunderea de informaţii (data-hiding)
- obiectele nu pot schimba starea internă a altor obiecte în mod direct (ci doar prin metode puse la dispoziţie de obiectul respectiv)
- doar metodele proprii ale obiectului pot accesa starea acestuia
- procesul de compartimentare a elementelor unei abstractizări în două părţi: structura şi comportarea
- încapsularea separă comportarea (accesată prin interfaţă) de structură, definită prin implementare
- fiecare tip de obiect expune o interfaţă pentru celelalte obiecte care specifică modul cum acele obiecte pot interacţiona cu el

Principii POO: Modularizarea

- este procesul de partiţionare a unui program în componente individuale (module)
- permite reducerea complexităţii programului prin definirea unor graniţe bine stabilite şi documentate în program.
  - modularizarea constă în partiţionarea programului în module care pot fi compilate separat, dar care au conexiuni cu alte module ale programului.
  - modulele servesc ca şi containere în care sunt declarate clasele şi obiectele programului.

Principii POO: Ierarhizarea

- Modalitatea de a ordona abstractizările (tipurile abstracte de date).
- Ierarhiile pot să denote relaţii de tip sau relaţii de agregare.
- Relaţiile de tip sunt definite prin moştenirile între clase, prin care o clasă (clasa derivată) moşteneşte structura sau comportarea definită în altă clasă (clasa de bază)
- Relaţiile de agregare specifică compunerea unui obiect din mai multe obiecte mai simple.
- Obs: în limbajele de programare procedurală agregarea se realiza prin structuri de tip înregistrare (record în Pascal, struct în C, etc).

Principii POO: Moştenirea

- permite definirea şi crearea unor clase specializate plecând de la clase (generale) care sunt deja definite
- permite construirea unor clase noi, care păstrează caracteristicile şi comportarea, deci datele şi funcţiile membru, de la una sau mai multe clase definite anterior, numite clase de bază, fiind posibilă redefinirea sau adăugarea unor date şi funcţii noi.
- o clasă ce moşteneşte una sau mai multe clase de bază se numeşte clasa derivată.
- posibilitatea refolosirii lucrurilor care funcţionează
- organizează şi facilitează polimorfismul şi încapsularea
- ” Anumite obiecte sunt similare dar în acelaşi timp diferite”.

Principii POO: Moştenirea

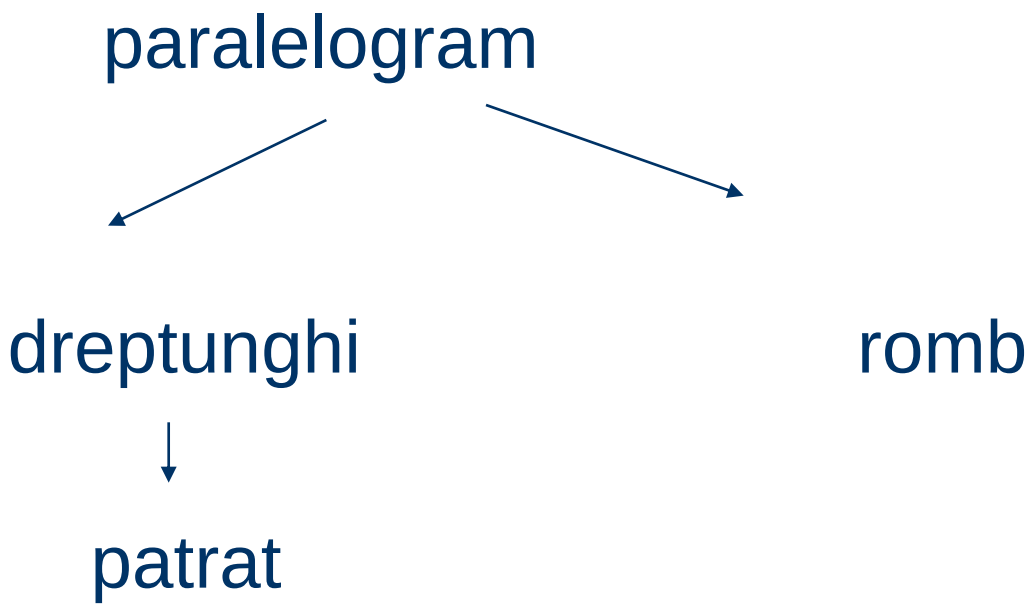
- Proprietatea de *moştenire: proprietatea* claselor prin care o clasă nou construită poate prelua datele şi metodele clasei mai vechi.

- Clasa *derivată* se află întotdeauna pe un nivel imediat inferior celui corespunzător clasei de *bază*.

- *În Java există doar moştenire simplă,* o ierarhie de clase în care fiecare clasă derivată are o singură clasă de bază.

Principii POO: Moştenirea

Exemplu.



- Clasa *dreptunghi* este o clasă derivată (subclasă) a clasei *paralelogram*, iar clasa *paralelogram* este o clasă de bază (supraclasă) a clasei *dreptunghi*
- Astfel, o ierarhie de concepte conduce la o *ierarhie între clasele* care implementează conceptele ierarhice respective.

Principii POO: Polimorfismul, supraîncărcarea

- Mai multe funcţii pot avea acelaşi nume în acelaşi domeniu de definiţie, dacă se pot diferenţia prin numărul sau tipul argumentelor de apel.
- O funcţie este polimorfică dacă se poate executa cu acelaşi efect asupra unor valori de tipuri diferite (ex. operatorul & din C)
- Un alt mecanism este supraîncărcarea funcţiilor(function overloading).
- O funcţie este supraîncărcată dacă execută operaţii diferite în contexte diferite (ex. operatorul '+' din Java)
- Se poate aplica doar funcţiilor.
- Supradefinirea (overriding) oferă posibilitatea de a redefini metode pentru clasele derivate, metodele au acelaşi tip şi aceeaşi parametri.

Principii POO: Polimorfismul, supraîncărcarea

- Dacă în acelaşi domeniu sunt definite mai multe funcţii cu acelaşi nume, la fiecare apel se selectează funcţia corectă prin compararea tipurilor argumentelor reale de apel cu tipurile argumentelor formale ale funcţiei.
  - double abs(double);
  - int abs(int);
  - abs(1); // apeleaza abs(int)
  - abs(1.0); // apeleaza abs(double)

- ***Nu este admis ca funcţiile să difere doar prin tipul returnat!***
- Două funcţii declarate cu acelaşi nume se referă la aceeaşi funcţie dacă sunt în acelaşi domeniu şi au număr şi tipuri identice de argumente.

Concluzii POO

- Programele: o colecţie de obiecte, unităţi individuale de cod care interacţionează unele cu altele, în loc de simple liste de instrucţiuni sau de apeluri de proceduri

- Obiectele POO sunt de obicei reprezentări ale obiectelor din viaţa reală

- Programele sunt mai uşor de înţeles, de depanat şi de extins decât programele procedurale (mai ales în cazul proiectelor software complexe şi de dimensiuni mari, care se gestionează făcând apel la ingineria programării).

Tip abstract de date

- mulţime de date care au aceeaşi reprezentare şi pentru care este definit setul de operaţii care se pot executa asupra elementelor mulţimii respective.

- are două părţi:
  - o parte care defineşte reprezentarea datelor
  - o parte care defineşte operaţiile asupra datelor respective.

Noţiunea de clasă

- O *clasă* defineşte un tip abstract de date.

Definiţie clasă:

class nume{

lista_elementelor_membru

}

Lista elementelor membru poate conţine:

- declaraţii de date;
- implementări de funcţii;
- prototipuri de funcţii abstracte.

- Datele declarate printr-o definiţie de clasă se numesc *date membru*
- Funcţiile definite sau pentru care este prezent numai prototipul în definiţia clasei, se numesc *funcţii membru* sau *metode.*
- Atât datele cât şi metodele pot avea modificatori de acces

Modificatorii de acces

- ***Modificatorii de acces*** sunt cuvinte rezervate ce controlează accesul celorlalte clase la membrii unei clase. Specificatorii de acces pentru variabilele şi metodele unei clase sunt: public, protected, private şi cel implicit (la nivel de pachet).

Specificator	Clasa	Subcls*	Pachet	Oriunde
Private	X			
Implicit	X		X	
Protected	X	X	X	
Public	X	X	X	X

***subclasă din alt pachet**

Clasă

Exemplu:

```
class Complex {
        // date membru
    float real;
    float imag;
       // functii membru publice
    public void atribuire(float x, float y) {
          real = x;  imag=y;
    }
    public double retreal() {
          return real;
    }
    public void afiscomplex(){
          System.out.println(real+”+“+imag+”*i”);
    }
}
```

Obiecte

- Un *obiect* este o dată de un tip definit printr-o clasă. Se spune că obiectul este o *instanţiere* a clasei respective.

- Formatul declaratiei unui obiect:

  nume_clasă nume_obiect;

- Instanţierea obiectelor se face folosind operatorul *new.*

  nume_obiect = new nume_clasă(..);

Obiecte

- Datele membru se alocă distinct la fiecare instanţiere a clasei. O excepţie o constituie datele membru care au clasa de memorare *static*, ea este o parte comună pentru toate instanţierile clasei şi există într-un singur exemplar.

- Funcţiile membru sunt într-un singur exemplar oricâte instanţieri ar exista. Legătura dintre funcţii membru şi obiectul pentru care se face apelul se realizează folosind operatorul punct.

Obiecte

- Exemplu de instanţieri pentru clasa *complex:*

```
Complex z;
z=new Complex();
```

- Atunci:

```
z.atribuire(0,0);
z.afiscomplex();
```

  - afişează numărul complex z (în cazul de faţă 0+0i).

Constructori

- Obiectele se generează şi se pot iniţializa la instanţiere cu ajutorul *constructorilor*
  - Funcţii membru ce au acelaşi nume cu numele clasei
  - Funcţii apelate automat la crearea obiectelor.
- Valorile de iniţializare se transferă constructorului şi ele joacă acelaşi rol ca parametrii efectivi de la apelurile funcţiilor obişnuite.
- Se pot defini mai mulţi constructori pentru o clasă. În acest caz ei au acelaşi nume, dar diferă prin numărul şi/sau tipurile parametrilor.

Constructori

- Dacă există mai mulţi constructori, atunci la iniţializare se utilizează regulile de la apelurile funcţiilor supraîncărcate.

- Funcţiile constructor nu întorc valori, dar nu sunt precedaţi de cuvântul *void.*

- Dacă clasa nu conţine constructori, se generează un constructor fără parametri, adică un *constructor implicit.* El are rolul numai de alocare a obiectelor clasei respective, fără a le iniţializa.

Constructori

Exemplu:

```
class Complex {
    double real;
    double imag;
     public Complex(double x, double y)
          {real = x; imag = y;}
    public Complex ( )
          {real = 0; imag = 0;}
 }
```

Exemple de instanţiere:

```
Complex z= new Complex();          // z = 0 + 0*i
Complex z1= new Complex(1,0);    // z1 = 1 + 0*
```

Realizarea încapsulării datelor

- accesul la datele sau funcţiile membre ale unei clase din orice punct al domeniului de definiţie al clasei s-ar putea rezolva simplu prin declararea de tip public a acestora
- o astfel de implementare nu respectă principiul încapsulării datelor şi se recomandă să fie evitată
- din punct de vedere al dreptului de acces la membrii clasei, o clasă bine definită permite încapsularea (sau ascunderea informaţiilor), prin care un obiect poate ascunde celor care-l folosesc modul de implementare, prin interzicerea accesului la datele şi funcţiile private sau protected.

Realizarea încapsulării datelor

- în general, respectând principiul încapsulării, datele membre sunt declarate private sau protected şi nu pot fi accesate direct (pentru citire sau scriere) din funcţii nemembre ale clasei.
- Pentru citirea sau modificarea unora dintre datele membre protejate în clasa respectivă se pot prevedea funcţii membre de tip public, care pot fi apelate din orice punct al domeniului de definiţie al clasei şi fac parte din interfaţa clasei.
- De exemplu, pentru clasa Complex , o implementare care respectă principiul încapsulării, dar, în acelaşi timp permite accesul la datele private ale clasei poate arăta astfel:

Realizarea încapsulării datelor

```
class Complex {
    private double real;
    private double imag;
     public Complex(double x, double y){
          real = x; imag = y;}
    public Complex ( ){
          real = 0; imag = 0;}
    public void set(double x, double y){
          real = x; imag = y; }
    public void setre(double x){
          real = x;}
    public void setim(double y){
          imag = y; }
    public double getre(){
          return real ;}
    public double getim() {
          return imag;}
```

Realizarea încapsulării datelor

```
    public void display(){
        System.out.println(real+"+"+ imag +"i");
    }
}

class test{
    public static void main(String arg[]){
        Complex c1=new Complex(), c2=new
                                        Complex(1,1);
        c1.set(7.2, 9.3);
        c1.display(); // afiseaza 7.2+9.3i
        c1.setre(1.3);
        c1.setim(2.8);
        c1.display(); // afiseaza 1.3+2.8i
    }
}
```