Programare orientată pe obiecte

1. Despre curs
2. Concepte şi paradigme în POO

Obiectivele cursului

- Să vă ofere cunoştinţe operaţionale în dezvoltarea software folosind orientarea pe obiecte, cu accent pe conceptele, paradigmele şi atitudinile necesare scrierii de cod orientat pe obiecte de calitate
- Să vă permită să vă dezvoltaţi îndemânarea în programare şi inginerie software, să învăţaţi să utilizaţi efectiv un limbaj de programare orientat pe obiecte relevant din punctul de vedere al industriei software

Despre POO

- Orele de curs: Marius Joldoş
  - Marius.Joldoş@cs.utcluj.ro
- Laborator: veţi afla cu cine
- Web:
  - http://users.utcluj.ro/~jim/OOPR (public)
  - http://t201:81/OOPR (accesibil numai in D3)
- 14 sesiuni de curs a 2 ore
- 14 sesiuni de laborator a 2 ore
- 5 credite
- Colocviu laborator, miniproiect, examen parţial si final

Rezultatele cursului (1)

- Privitoare la cunoştinţe/înţelegere
  - Fundamentele POO
  - Elementele principale în proiectarea, programarea, testarea şi documentarea soluţiilor OO
  - Metode de proiectare pentru programe Java de complexitate relativ redusă
  - Elemente de bază ale UML

Rezultatele cursului (2)

- Abilităţi intelectuale
  - Să înţelegeţi specificaţia unei probleme şi să o rezolvaţi printr-un program pentru calculator în paradigma programării orientate pe obiecte.
  - Să luaţi o specificaţie parţială şi să luaţi deciziile corespunzătoare asupra funcţionalităţii sistemului propus.
  - Să dezvoltaţi o specificaţie în UML dintr-o descriere în limbaj natural

Rezultatele cursului (3)

- Abilităţi practice
  - Să folosiţi eficient conceptele de programare OO în Java
  - Să dezvoltaţi programe de complexitate relativ redusă în Java
  - Să depanaţi, testaţi şi documentaţi soluţii folosind orientarea pe obiecte
  - Să dezvoltaţi applet-uri şi componente GUI relativ simple

Subiecte abordate

- Concepte şi paradigme în programarea orientată pe obiecte
- Abstracţiuni şi tipuri de date abstracte
- Caracteristicile limbajului Java
- Tipurile de date primitive şi structurile de control în Java
- Clase şi obiecte
- Interfeţe Java
- Excepţii şi tratarea lor
- Principalele API-uri şi clase predefinite în Java
- Reprezentarea în UML a claselor, obiectelor şi asociaţiilor; diagrame de clase şi obiecte
- Programarea bazată pe clase şi obiecte
- Moştenirea şi polimorfismul
- Programarea orientată pe obiecte
- Scurtă introducere în programarea condusă de evenimente
- Scurtă introducere în programarea applet-urilor şi a componentelor GUI

Evaluare. Referinţe

- Evaluare
  - Examen scris (E), evaluarea activităţii de la laborator (L), evaluarea unui mini-proiect (P).
  - Nota finală = 0.6E + 0.4(L+P)/2
- Referinţe
  - Tănasă, S., Olaru, C, Andrei, Şt., *Java de la 0 la expert*, Ed. Polirom, 2003
  - Călin Văduva, *Programarea în Java*, Ed. Albastră, 2004 (reeditare)
  - Bruce Eckel, *Thinking in Java*, 3rd edition, Prentice Hall, 2002
  - Documentaţia Java de la Sun Microsystems
  - Tutoriale UML introductive
  - Documentaţia BlueJ

Concepte şi paradigme în POO

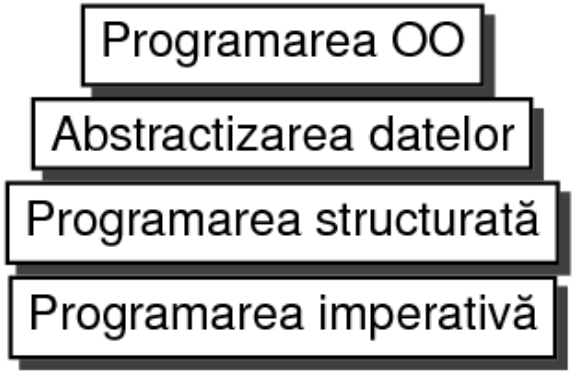
- Temele de astăzi
  - Paradigme de programare (programarea imperativă şi programarea structurată)
  - Abstractizarea datelor
    - Tipuri de date abstracte
  - Paradigma POO
  - Concepte POO
    - Obiecte şi clase
    - Încapsulare şi moştenire

Paradigme de programare

- **Paradigmă** (dict.): Un set de supoziţii, concepte, valori şi practici care constituie o viziune a realităţii pentru comunitatea care le adoptă, în special într-o disciplină intelectuală.
- **Paradigmă de programare**
  - Un model care descrie esența şi structura computației
  - Oferă (şi determină) viziunea pe care o are programatorul asupra execuţiei programului

    Exemple:
    - în POO, programatorii pot concepe programele ca fiind o colecţie de obiecte care interacţionează
    - în programarea funcţională un program poate fi conceput ca fiind o secvenţă de evaluări de funcţii, fără stări

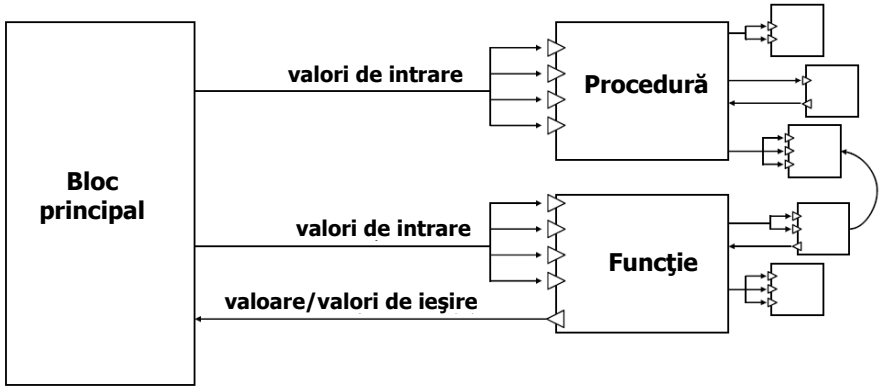
Paradigme de programare



Programarea imperativă

- În modelul tradițional von Neumann, un calculator constă dintr-o unitate centrală de prelucrare şi memorie şi el efectuează secvențe de instrucțiuni atomice care accesează, operează asupra valorilor stocate locații de memorie adresabile individual şi le modifică. Aici, o **computație este o serie de operații aritmetice şi de efecte laterale, cum sunt atribuirile sau transferurile de date care modifică starea unității de stocare, intrarea sau ieşirea**
- Ne referim la acest ca ***paradigmă imperativă sau procedurală.***
- Este de subliniat importanța atribuirilor şi a variabilelor pe post de containere pentru paradigma imperativă
- Exemple de limbaje: Fortran, Pascal, C, Ada.

Programarea structurată



Programarea structurată

- Abstractizarea operaţiilor
  - Structura unui modul
    - Interfaţa
      - Date de intrare
      - Date de ieşire
      - Descrierea funcţionalităţii
    - Implementarea
      - Date locale
      - Secvenţe de instrucţiuni
  - Sintaxa limbajului
    - Organizarea codului în blocuri de instrucţiuni
      Definiţii de funcţii şi proceduri
    - Extinderea limbajului cu noi operaţii
      Apeluri la proceduri şi funcţii noi

Beneficiile programării structurate

- Uşurează dezvoltarea software
  - Evită repetarea realizării aceluiaşi lucru
  - Munca de programare este descompusă în module independente
  - Proiectare Top-down: descompunerea în subprobleme
- Facilitează întreţinerea software
  - Codul este mai uşor de citit
  - Independenţa modulelor
- Favorizează reutilizarea software

Programarea structurată. Exemplu

```
int main()
{
  double u1, u2, m;
  u1 = 4;
  u2 = -2;
  m =sqrt (u1*u1 + u2*u2);
  printf("%lf\n", m);
  return 0;
}
```

```
double module(double u1, double u2)
{
  double m;
  m =sqrt (u1*u1 + u2*u2);
  return m;
}
int main()
{
  printf("%lf\n", module(4, -2));
  return 0;
}
```

Abstractizarea datelor

- Abstractizarea datelor: impunerea unei separări clare între *proprietăţile abstracte* ale unui *tip de dată* şi *detaliile concrete* ale *implementării* lui
  - Proprietăţi abstracte: acelea care sunt vizibile codului client care foloseşte tipul de dată – *interfaţa* cu tipul de dată
  - Implementarea concretă este păstrată in totalitate privată şi ea se poate într-adevăr schimba, spre exemplu pentru a incorpora îmbunătăţiri ale performanţelor în timp.

Tipuri abstracte de date

- Abstractizarea datelor + abstractizarea operaţiilor
  - Un tip de dată abstract:
    - **Structură de date** care stochează informaţii pentru a reprezenta un anumit concept
    - **Funcţionalitate:** set de operaţii care pot fi aplicate tipului de dată
  - Sintaxa limbajului
    - Modulele sunt asociate tipurilor de date
    - Sintaxa nu este neapărat nouă faţă de programarea modulară

Exemplu de tip de dată abstract în C

```
struct vector {
  double x;
  double y;
}
void construct (vector *v, double v1, double v2)
{
  v->x = v1;
  v->y = v2;
}
double module(vector v)
{
  double m;
  m = sqrt (v.x*v.x + v.y*v.y);
  return m;
}
```

```
int main()
{
  vector v;
  construct(&v, 4, 2);
  printf("%lf\n", module(v));
  return 0;
}
```


Extensibilitatea tipului de dată abstract

```
...

double product(vector v, vector w) {
  return v.x*w.x + v.y*w.y;
}
int main() {
  vector v;
  construct(&v, 4, 2);
  construct(&w, -1, 7);
  printf("%lf\n", product(v, w));
  return 0;
}
```

Beneficiile tipurilor de date abstracte

- Conceptele din domeniu sunt reflectate în cod
- *Încapsulare:* complexitatea internă, datele şi detaliile operaţiilor sunt ascunse
- Utilizarea tipului de dată este independentă de implementarea sa internă
- Oferă o mai mare modularitate
- Sporeşte uşurinţa întreţinerii şi reutilizării codului

Paradigma orientată pe obiecte

- Paradigma programării structurate a avut iniţial succes (1975-85)
  - Dar a început să eşueze la produse mai mari (> 50,000 LOC)
- PS avea probleme de întreţinere post-livrare (astăzi această întreţinere necesită, de la 70 la 80% din efortul total)
- Motivul: Metodele structurate sunt fie
  - orientate pe operaţii (analiza fluxului de date) fie
  - orientate pe atribute (d.e. dezvoltarea cu metoda Jackson)…
  - …dar nu amândouă

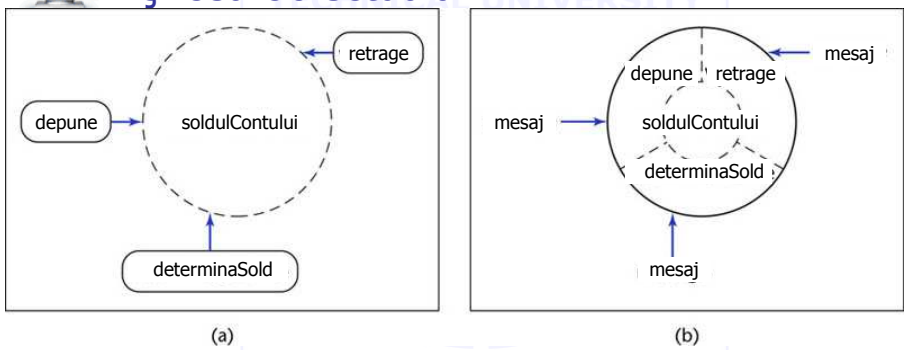
Paradigma orientării pe obiecte

- Paradigma folosită în limbaje: o simulare a domeniului unei probleme prin abstractizarea informaţiilor de comportament şi stare din obiecte din lumea reală
- Conceptele de obiecte, clase, transmitere de mesaje şi moştenire sunt cunoscute ca făcând parte din paradigma orientării pe obiecte.

Paradigma orientării pe obiecte

- POO consideră că atât atributele cât şi operaţiile au importanţă egală
- O viziune simplistă a unui obiect poate fi:
  - Obiect = componentă software care incorporează atât atributele cât şi operaţiile care se pot efectua asupra atributelor şi care suportă moştenirea.
- Exemplu:
  - Cont bancar
    - Date: soldul contului
    - Acţiuni: depune, retrage, determină soldul

Comparaţie între paradigma structurată şi cea obiectuală



- Ascunderea informaţiei
- Proiectarea dirijată de responsabilităţi
- Impact asupra întreţinerii şi dezvoltării

Ascunderea informaţiei

- În versiunea orientată pe obiecte
  - Linia continuă din jurul lui `soldulContului` arată că în afara obiectului nu se ştie cum este implementat `soldulContului`
- În versiunea clasică
  - Toate modulele au detalii privind implementarea lui `soldulContului`

Punctele tari ale paradigmei OO

- Cu ascunderea informaţiei, întreţinerea post-livrare este mai sigură
  - Şansele apariţiei erorilor regresive sunt reduse (în software nu se repetă erori cunoscute)
- Dezvoltarea este mai uşoară
  - Obiectele au în general corespondente fizice
    - Acest lucru simplifică modelarea (un aspect cheie al paradigmei OO)

Punctele tari ale paradigmei OO (2)

- Obiectele bine proiectate sunt unităţi independente
  - Tot ce se referă la obiectul real modelat este în obiect — *încapsulare*
  - Comunicarea se face prin schimb de *mesaje*
  - Această independenţă este augmentată prin *proiectarea dirijată de responsabilitate*

Punctele tari ale paradigmei OO (3)

- Un produs clasic constă d.p.d.v conceptual dintr-o singură unitate (deşi poate fi implementată ca un set de module)
  - Paradigma OO reduce complexitatea deoarece produsul constă, în general, din unităţi independente
- Paradigma OO promovează reutilizarea
  - Obiectele sunt entităţi independente

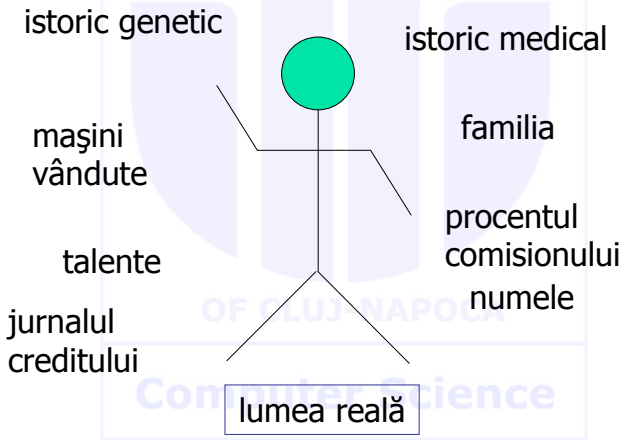
Programarea orientată pe obiecte

- Oferă suport sintactic pentru tipurile de date abstracte
- Oferă facilităţi asociate cu ierarhiile de clase
- Schimbă punctul de vedere: programele sunt apendice ale datelor
- Introduc un concept nou: *obiect* = tip de dată abstract cu *stare* (atribute) şi *comportament* (operaţii)

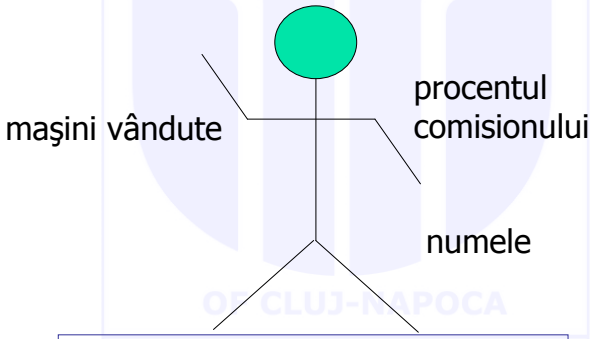
Concepte OOP

- Ne ocupăm doar de datele care prezintă interes pentru problema noastră.
- Acest proces de filtrare a detaliilor neimportante ale obiectului astfel încât să rămână doar caracteristicile importante este cunoscut sub numele de *abstractizare*.

Abstractizare

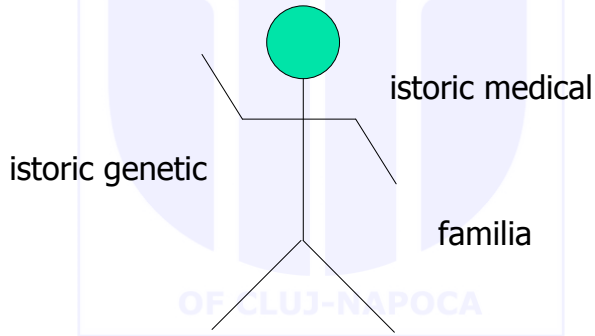


Abstractizare



Abstractizarea unei PersoanaVinzator pentru un Sistem de urmărire a vânzărilor

Abstractizare



Abstractizarea unui Pacient dintr-o baze de date medicală

Ce sunt obiectele software?

- Blocurile de construcţie a sistemelor software
  - un program este o colecţie de obiecte care interacţionează
  - obiectele cooperează pentru a finaliza o sarcină
  - pentru aceasta, ele comunică trimiţându-si "mesaje" unul altuia
- Obiectele modelează lucruri *tangibile*
  - persoană
  - bicicletă
  - ...

Ce sunt obiectele software?

- Obiectele modelează *lucruri conceptuale*
  - întâlnire
  - dată calendaristică
- Obiectele modelează *procese*
  - aflarea drumului printr-un labirint
  - sortarea unui pachet de cărţi de joc
- Obiectele au
  - capabilităţi: ce pot face, cum se comportă
  - proprietăţi: trăsături (caracteristici) care descriu obiectele

Capabilităţile obiectelor: acţiuni

- Obiectele au *capabilităţi* (*comportamente*) care le permit să efectueze acţiuni specifice
  - obiectele sunt deştepte — ele "ştiu" cum să facă anumite lucruri
  - un obiect face ceva doar dacă un alt obiect îi spune să-şi folosească una dintre capabilităţi
- Capabilităţile pot fi:
  - *constructori*: stabilesc starea iniţială a proprietăţilor obiectului
  - comenzi: modifică proprietăţile obiectului
  - interogări: furnizează răspunsuri bazate pe proprietăţile obiectului

Capabilităţile obiectelor: acţiuni

- Exemple: *borcanele cu gem* sunt capabile să efectueze acţiuni specifice
  - constructor: să fie creat
  - comenzi: adaugă gem, goleşte-te
  - interogări: răspunde dacă este închis sau deschis capacul, dacă borcanul este plin sau gol

Proprietăţile obiectului: starea

- *Proprietăţile* determină cum acţionează un obiect
  - unele proprietăţi pot fi constante, iar altele variabile
  - proprietăţile însele sunt obiecte — pot şi primi mesaje
  - capacul *borcanului cu gem* şi gemul în sine sunt obiecte
- Proprietăţile pot fi:
  - atribute: lucruri care ajută la descrierea unui obiect
  - componente: lucruri care sunt "parte a" unui obiect
  - asocieri: lucruri despre care ştie un obiect, dar care nu sunt parte a acelui obiect

Proprietăţile obiectului: starea

- *Stare*: colecţie a tuturor proprietăţilor obiectului; se schimbă dacă o proprietate se schimbă
  - unele nu se schimbă, d.e. volanul unei maşini
  - altele se schimbă, d.e. culoarea maşinii
- Exemplu: proprietăţile *borcanelor cu gem*
  - atribute: culoare, material, miros
  - componente: capac, container, etichetă
  - asocieri: un *borcan cu gem* poate fi asociat cu încăperea în care se află

Clase şi instanţe

- Concepţia noastră curentă: fiecare obiect corespunde direct unui *anumit* obiect din realitate, d.e., un atom sau un automobil anume
- Dezavantaj: este mult prea nepractic să lucrăm cu obiecte în acest fel deoarece
  - ele pot fi infinit de multe
  - nu dorim să descriem fiecare individ separat, deoarece ei au multe lucruri în comun
- Clasificarea obiectelor scoate în evidenţă ce este comun între mulţimi de obiecte similare
  - mai întâi să descriem ce este comun
  - apoi să "ştampilăm" oricâte copii

Clase ale obiectelor

- *Clasa unui obiect*
  - categoria obiectului
  - defineşte capabilităţile şi proprietăţile comune unei mulţimi de obiecte individuale
    - toate *borcanele cu gem* se pot deschide, închide şi goli
  - defineşte un şablon pentru crearea de *instanţe de obiect*
    - unele *borcane cu gem* pot fi din plastic, pot fi colorate, de o anumită mărime etc.

Clase ale obiectelor

- Clasele implementează capabilităţile ca *metode*
  - secvenţe de instrucţiuni în Java
  - obiectele cooperează trimiţând mesaje altor obiecte
  - fiecare mesaj "invocă o metodă"
- Clasele implementează proprietăţile ca *variabile instanţă*
  - locaţie de memorie alocată obiectului, care poate păstra o valoare care se poate schimba

Instanţe de obiect

- *Instanţele obiectelor* sunt obiecte individuale
  - realizate din şablonul clasei
  - o clasă poate reprezenta un număr nedefinit de instanţe de obiect
  - realizarea unei instanţe de obiect constituie *instanţierea* obiectului respectiv
- Prescurtare:
  - clasă: clasa obiectului
  - instanţă: instanţa obiectului (a nu se confunda cu variabilele instanţă)

Instanţe de obiect

- Instanţe diferite ale, d.e., clasei **`BorcanCuGem`** pot avea:
  - culoare şi poziţie diferită
  - diverse tipuri de gem în interior
- Astfel că, *variabilele instanţă* ale lor au valori diferite
  - Notă: instanţele de obiect conţin variabile instanţă — două moduri de folosire diferite, dar înrudite, a cuvântului *instanţă*
- Instanţele individuale au identităţi individuale
  - aceasta permite altor obiecte să trimită mesaje unui obiect dat
  - fiecare instanţă este unică, chiar dacă are aceleaşi capabilităţi
    - gândiţi-vă la clasa studenţilor de la acest curs

Mesaje pentru comunicarea între obiecte

- Nici o instanţă nu este o insulă — ea trebuie să comunice cu altele pentru a-şi realiza sarcina
  - proprietăţile le permit să ştie despre alte obiecte
- Instanţele trimit mesaje una alteia pentru a invoca o capabilitate (adică, pentru a executa o sarcină)
  - metoda reprezintă codul care implementează mesajul
  - spunem "apelează metoda" în loc de "invocă capabilitatea"

Mesaje pentru comunicarea între obiecte

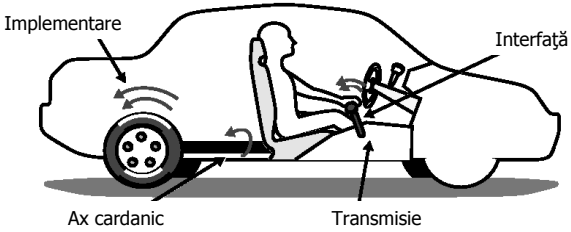
- Fiecare mesaj necesită:
  - *un emiţător (expeditor)*: obiectul care iniţiază acţiunea
  - *un receptor*: instanţa a cărei metode este invocată
  - *numele mesajului*: numele metodei apelate
  - opţional *parametri*: informaţii suplimentare necesitate de metodă pentru a opera
    - mai multe mai târziu
- Receptorul poate (dar nu este nevoit) sa trimită un răspuns
  - vom trata *tipurile returnate* în detaliu, mai târziu

Încapsulare

- Un automobil *încapsulează* multă informaţie
  - chiar literal, prin complexitatea construcţiei sale
- Dar nu este nevoie să ştii cum funcţionează o maşină pentru a o conduce
  - Volanul şi schimbarea vitezelor constituie interfaţa
  - motorul, transmisia, axul cardanic, roţile, . . . , sunt implementarea (ascunsă)

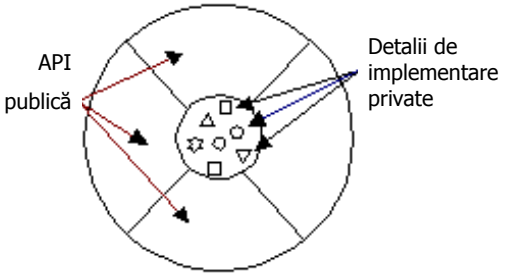
Încapsulare

- Asemănător, nu e nevoie să ştim cum funcţionează un obiect pentru a-i trimite mesaje
- Dar, este nevoie să ştim ce mesaje înţelege (adică, care îi sunt capabilităţile)
  - clasa instanţei determină ce mesaje îi pot fi trimise



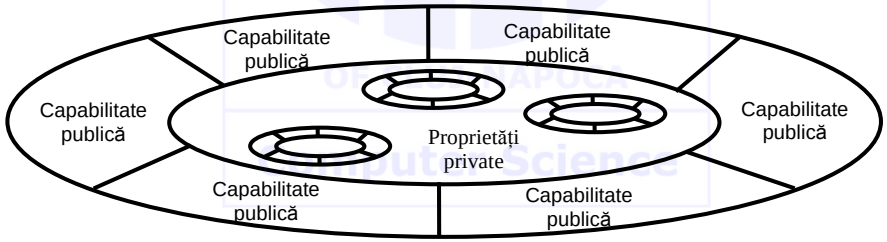
Încapsularea

- Închiderea datelor într-un obiect
  - Datele nu pot fi accesate direct din afară
- Oferă securitatea datelor



Vederile unei clase

- Obiectele separă *interfaţa* de *implementare*
  - obiectul este "cutie neagră"; ascunde funcţionarea şi părţile interne
  - interfaţa protejează implementarea împotriva utilizării greşite

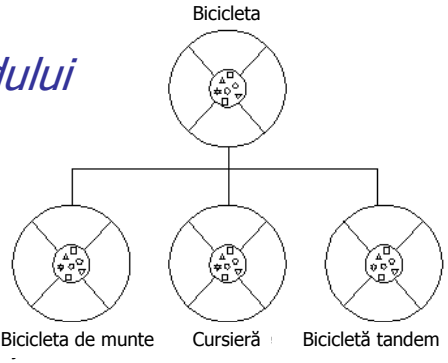


Vederile unei clase

- Interfaţa: vedere *publică*
  - permite instanţelor să coopereze unele cu altele fără a şti prea multe detalii
  - ca un *contract*: constă dintr-o listă de capabilităţi şi documentaţie pentru cum să fie folosite
- Implementarea: vedere *privată*
  - proprietăţile care ajută capabilităţile să-şi îndeplinească sarcinile

Moştenirea

- O clasă *(subclasă)* poate *moşteni* atribute şi metode dintr-o altă clasă (superclasă)
- Subclasele furnizează comportament specializat
- Oferă *reutilizarea codului*
- Evită *duplicarea* datelor



Polimorfism

- Abilitatea de a lua multe forme
- *Aceeaşi metodă* folosită într-o superclasă poate fi *suprascrisă* în subclase pentru a da o *funcţionalitate diferită*
- D.e. Superclasa 'Poligon' are o metodă numită, aflaSuprafata

  aflaSuprafata în subclasa 'Triunghi' → a=x*y/2

  aflaSuprafata in subclasa 'Dreptunghi' → a=x*y

Concepte prezentate

- Paradigme în programare
  - structurată
  - orientată pe obiecte
- Abstractizare
  - tip de dată abstract
- Obiect
  - capabilităţi(comportamente): acţiuni
    - constructor
    - comenzi
    - interogări
  - proprietăţi: stare
    - atribute
    - componente
    - asocieri
  - instanţă a unei clase
- Comunicarea între obiecte: mesaje
- Încapsulare
- Clasă
  - şablon pentru crearea obiectelor
  - capabilităţile obiectelor: implementate ca metode
  - proprietăţile obiectelor: variabile instanţă
  - interfaţă – vedere publică
  - implementare – vedere privată
- Moştenire
  - superclasă, subclasă
- Polimorfism