2009 - **2010**

Inginerie Software pentru Comunicatii (ISC / RST)

Titular curs: **Eduard-Cristian Popovici**

Suport curs: *http://discipline.elcom.pub.ro/isc/*

Moodle: *http://electronica07.curs.ncit.pub.ro/course/category.php?id=4*

Continut curs

1. Introducere in ingineria software

1.1. Necesitatea unei abordari sistematice a dezvoltarii software

1.2. Abordari si metodologii larg utilizate in ingineria software

2. Introducere in limbajul UML

2.1. Definirea, rolul si istoricul limbajului de modelare unificat (UML)

2.2. Tipuri de diagrame UML. Organizarea ierarhica a diagramelor

3. Diagrame UML statice

3.1. Diagrame UML de clase

3.2. Diagrame UML de obiecte

3.3. Diagrame UML de pachete

3.4. Diagrame UML de componente

3.5. Diagrame UML de structuri compozite

Continut curs

4. Diagrame UML dinamice

- 4.1. Diagramele UML de caz de utilizare
- 4.1. Diagrame UML de comunicatie si de robustete
- 4.2. Diagrame UML de secventa si de sumar al interactiunilor
- 4.3. Diagrame UML de masini de stari
- 4.4. Diagrame UML de activitati
- 4.5. Diagrame UML de timp

5. Introducere in procesul de dezvoltare Rational unificat (RUP)

- 5.1. Organizarea iterativa a proiectelor
- 5.2. Fazele si activitatile procesului RUP

6. Introducere in managementul si organizarea proceselor de dezvoltare
7. Elemente de reutilizabilitate a software-ului. Pattern-uri de proiectare

3. Diagrame UML statice

3.1. Diagrame UML de clase

Orientarea spre obiecte

Caracteristicile de baza ale orientarii spre obiecte

A – abstactizare (**modelare**)

P – polimorfism (in ierarhiile bazate pe generalizare/specializare)

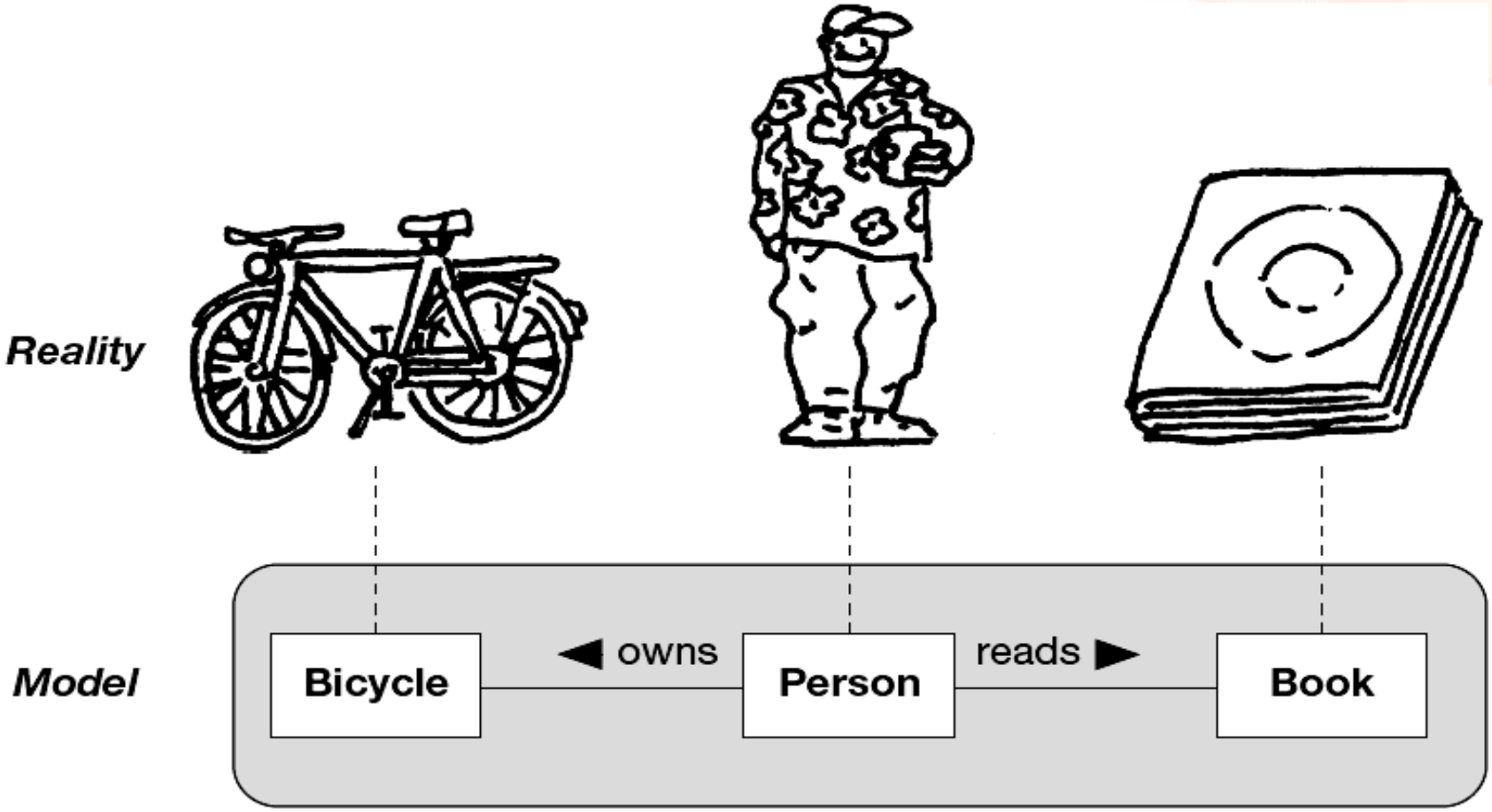
I – generalizare / specializare / mostenire (*inheritance*)

E – incapsulare (*encapsulation*) **a atributelor si operatiilor**

Alte caracteristici

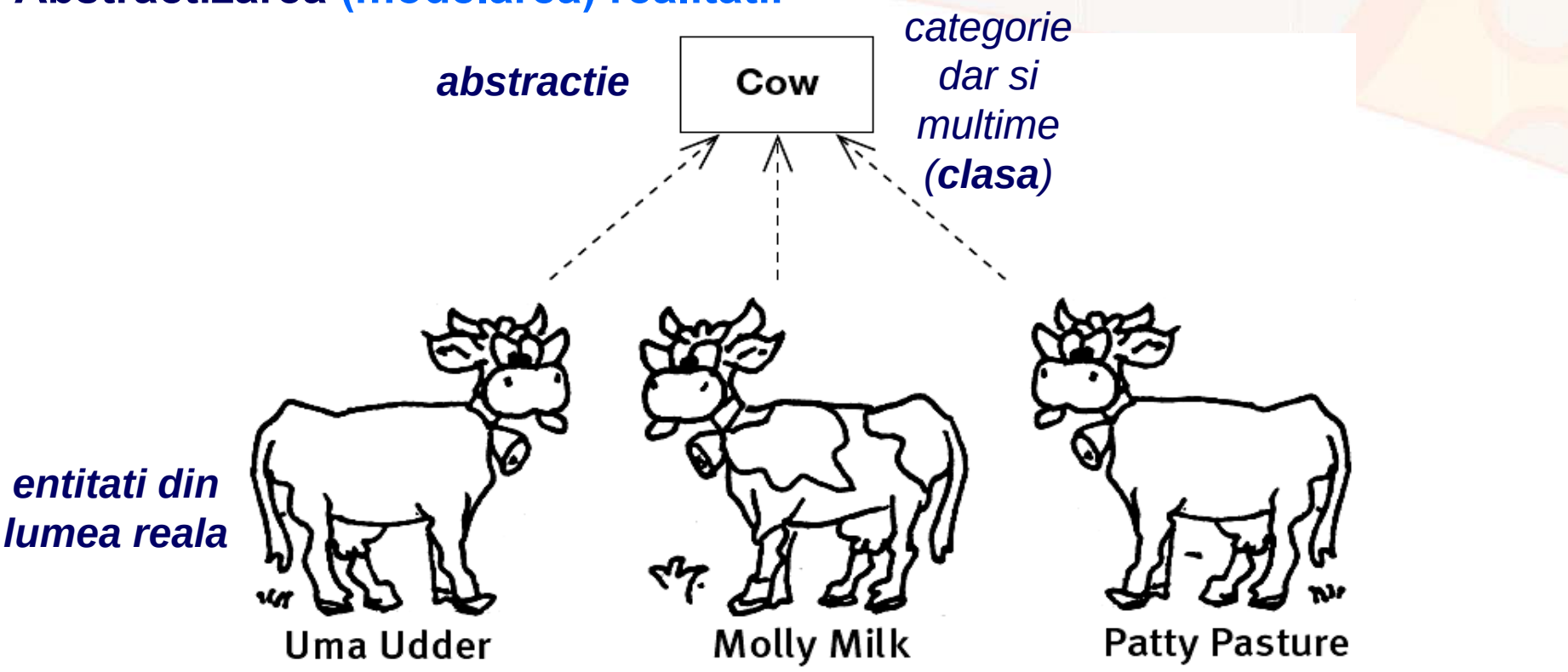
- obiectele au **stare** (ansamblul valorilor atributelor) si **comportament**
- obiectele **colaboreaza** prin schimburi de **mesaje** (invocari de operatii)
- **interfata** obiectului este **publica**
- **implementarea** (din spatele) interfetei si **starea** obiectului sunt **private** (inaccesibile direct, ascunse, protejate)

Abstractizarea (modelarea) realitatii



abstractii

Abstractizarea (modelarea) realitatii



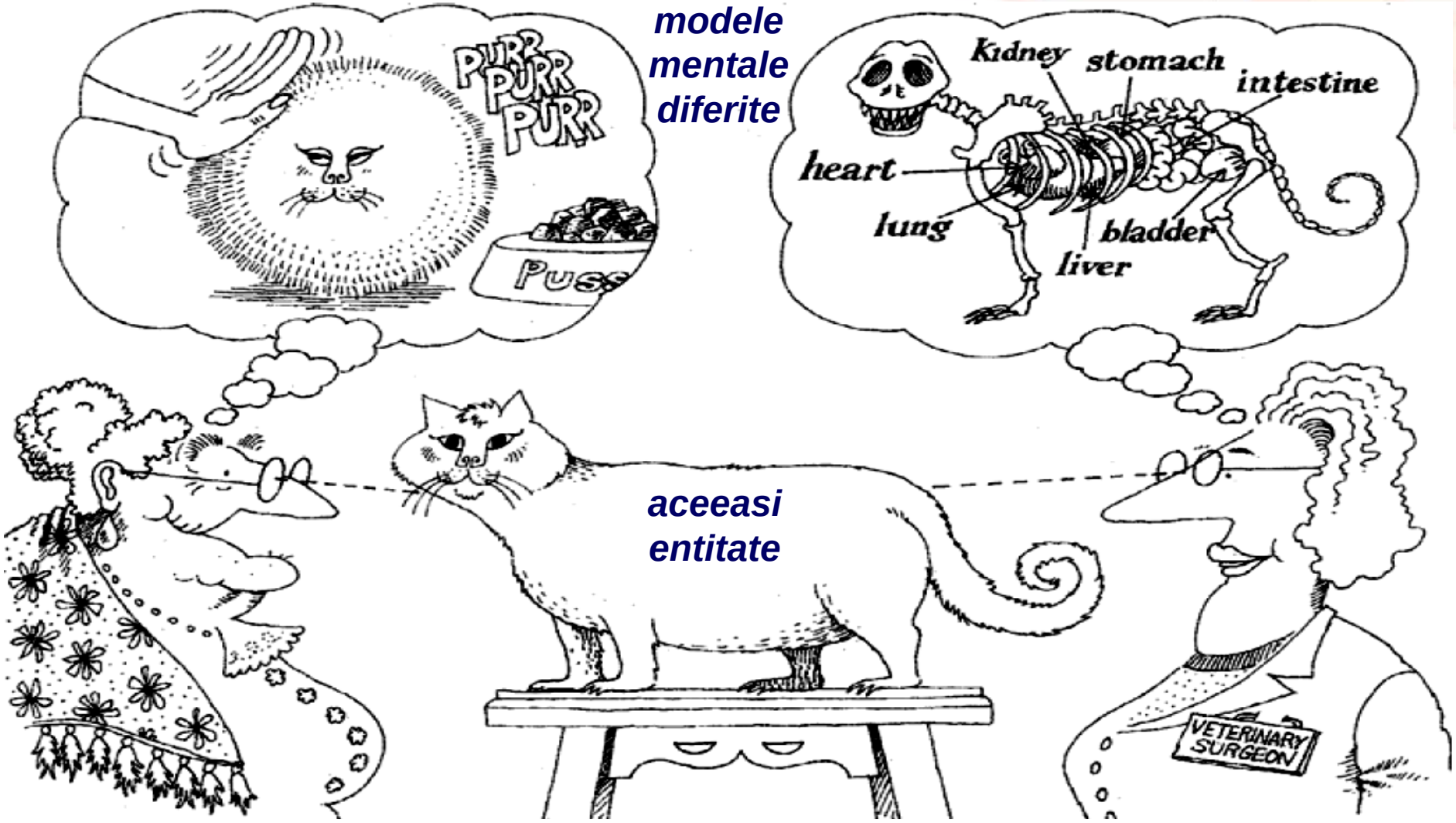
abstractizare a entitatii

abstractizare a obiectelor

Abstractizarea (modelarea) realitatii

detaliile considerate nesemnificative

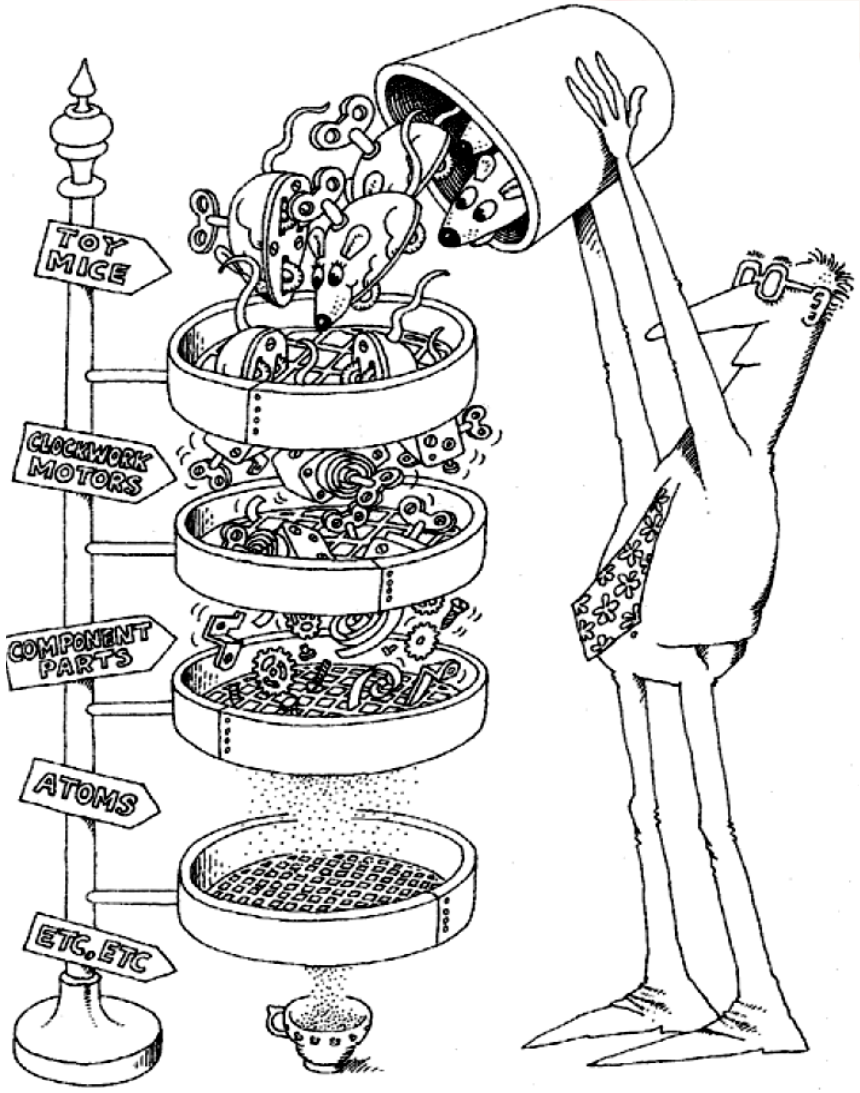
Abstractizarea realitatii este relativa (dependenta de context)



Abstractizarea formeaza ierarhii (diferenta fiind data de detalii)

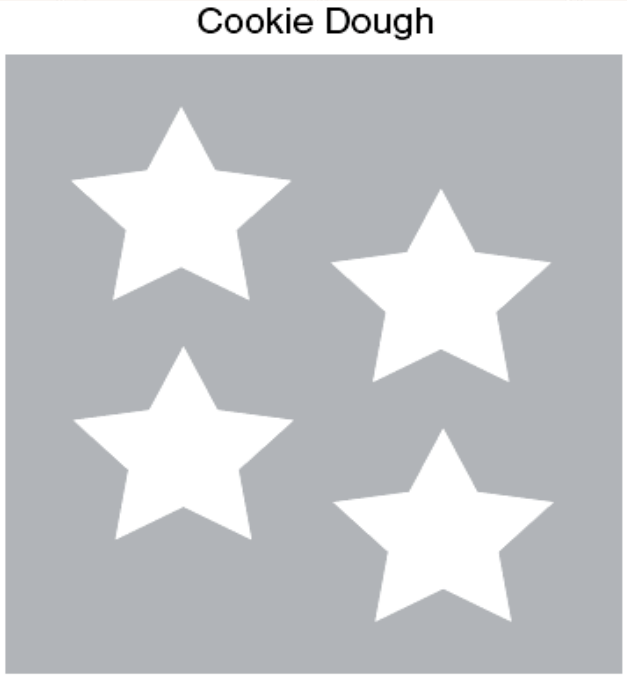
diferite niveluri de detaliu

(diferite niveluri de abstractizare)



Clasa si obiectele

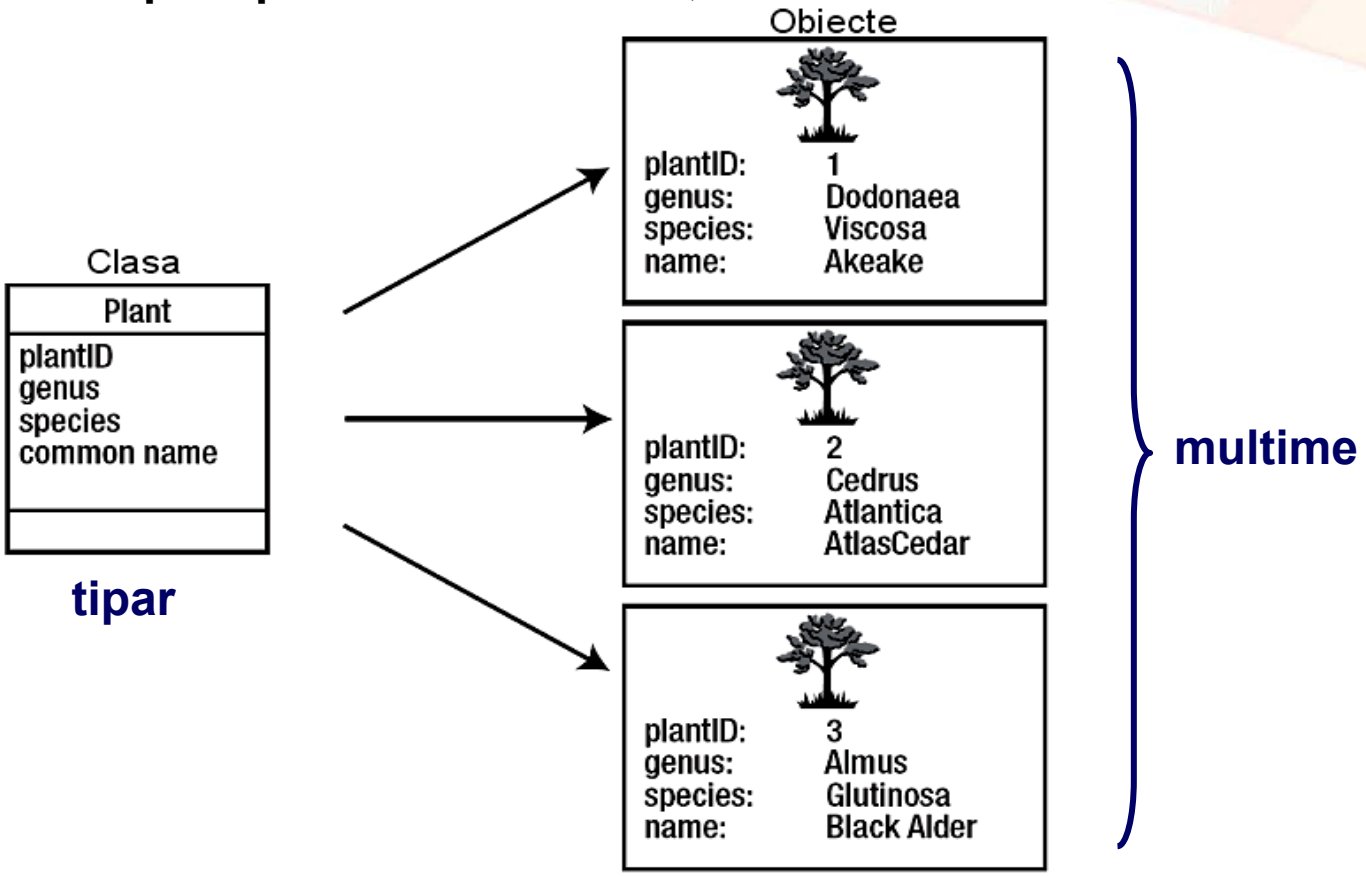
clasa *vazuta ca* ***model (proiect)d upa care se produc obiectele***



clasa *vazuta ca* ***multime de obiecte***

Clasa si obiectele

Obiectul este **exemplu specific** al unei clase, numit **instanta** a clasei



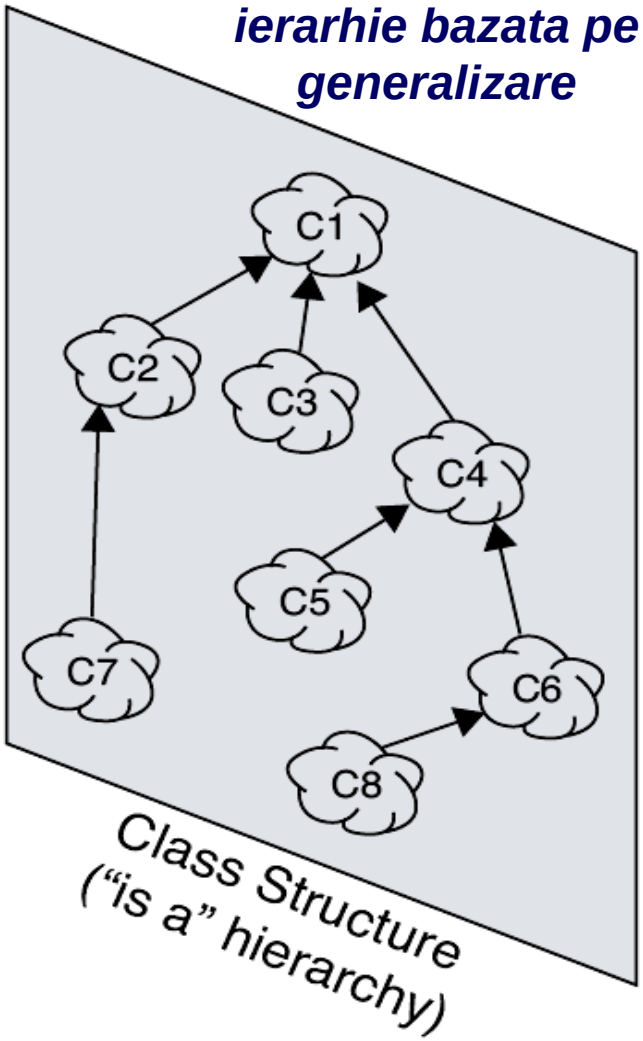
Clasa este

- **tipar** sau **sablon** dupa care sunt construite variabile numite **obiecte**
- **domeniu de definitie** (asemanator unei **multimi**) pentru obiecte

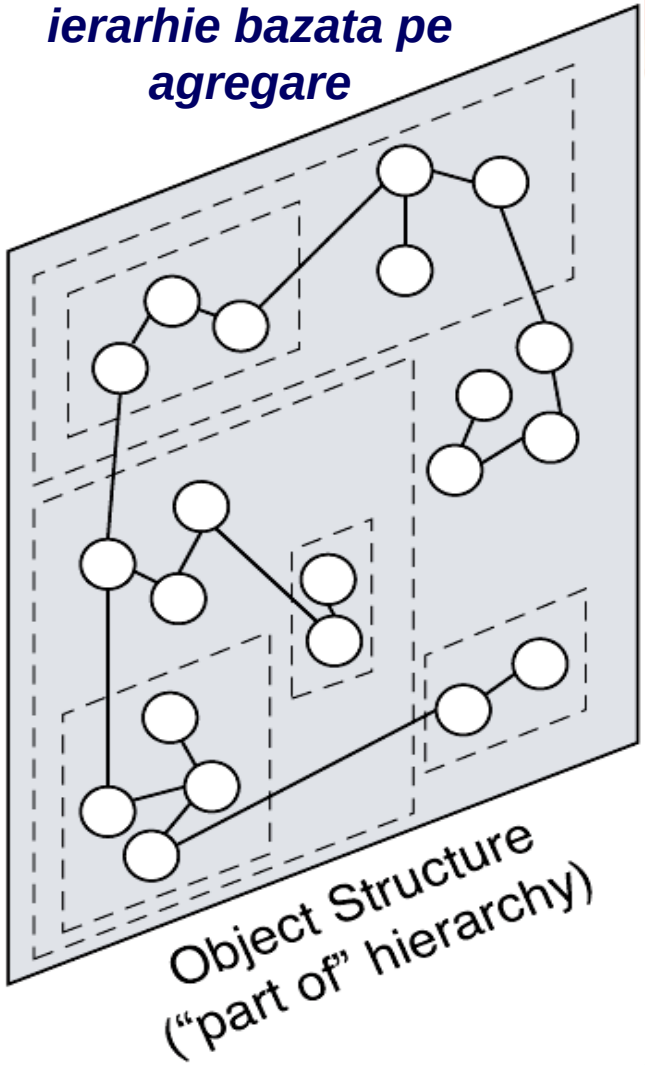
Ierarhii bazate pe **generalizare** si ierarhii bazate pe **agregare**

clasele *vazute ca* ***multimi*** *(de obiecte)* ***intre care exista includere***

ierarhie bazata pe generalizare

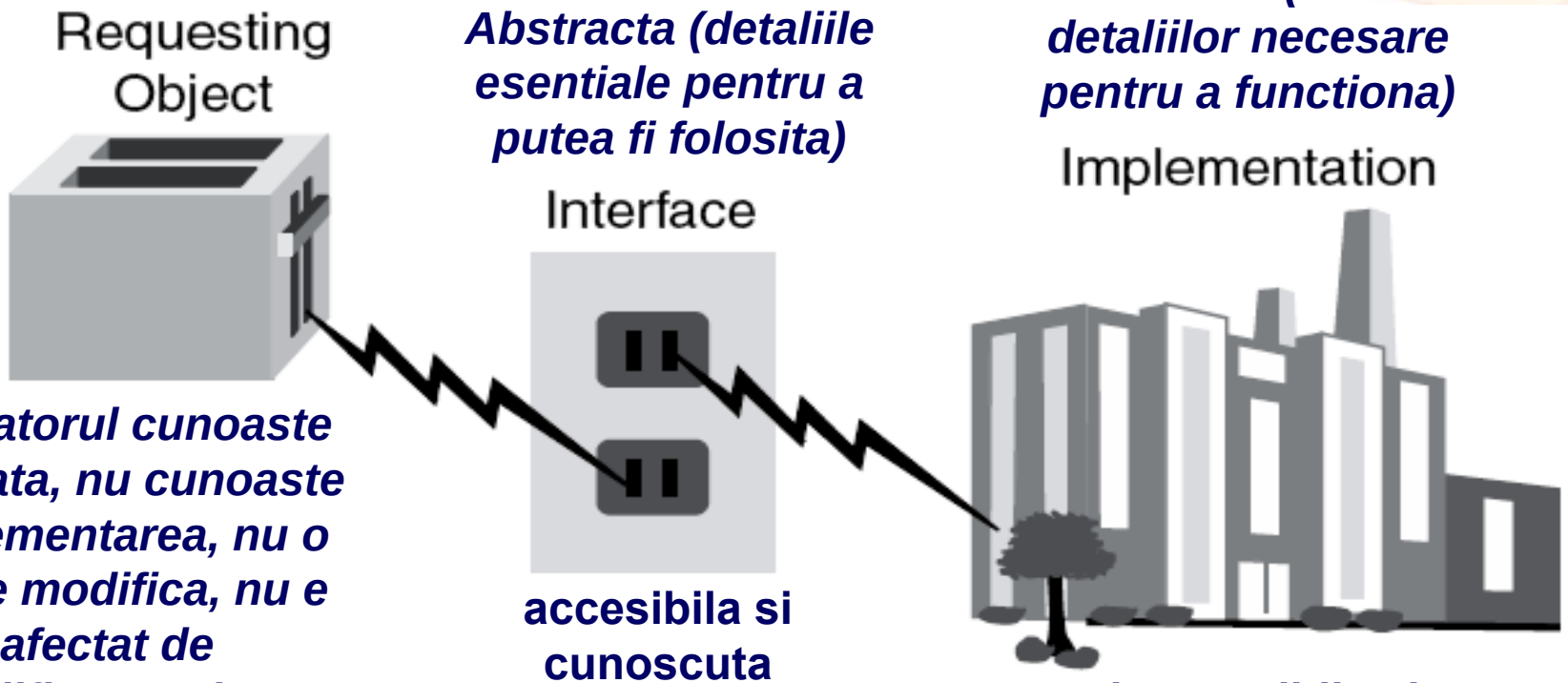


ierarhie bazata pe agregare



clasele *vazute ca* ***tipuri*** *de* ***obiecte (intre care exista legaturi)***

Interfata si implementarea

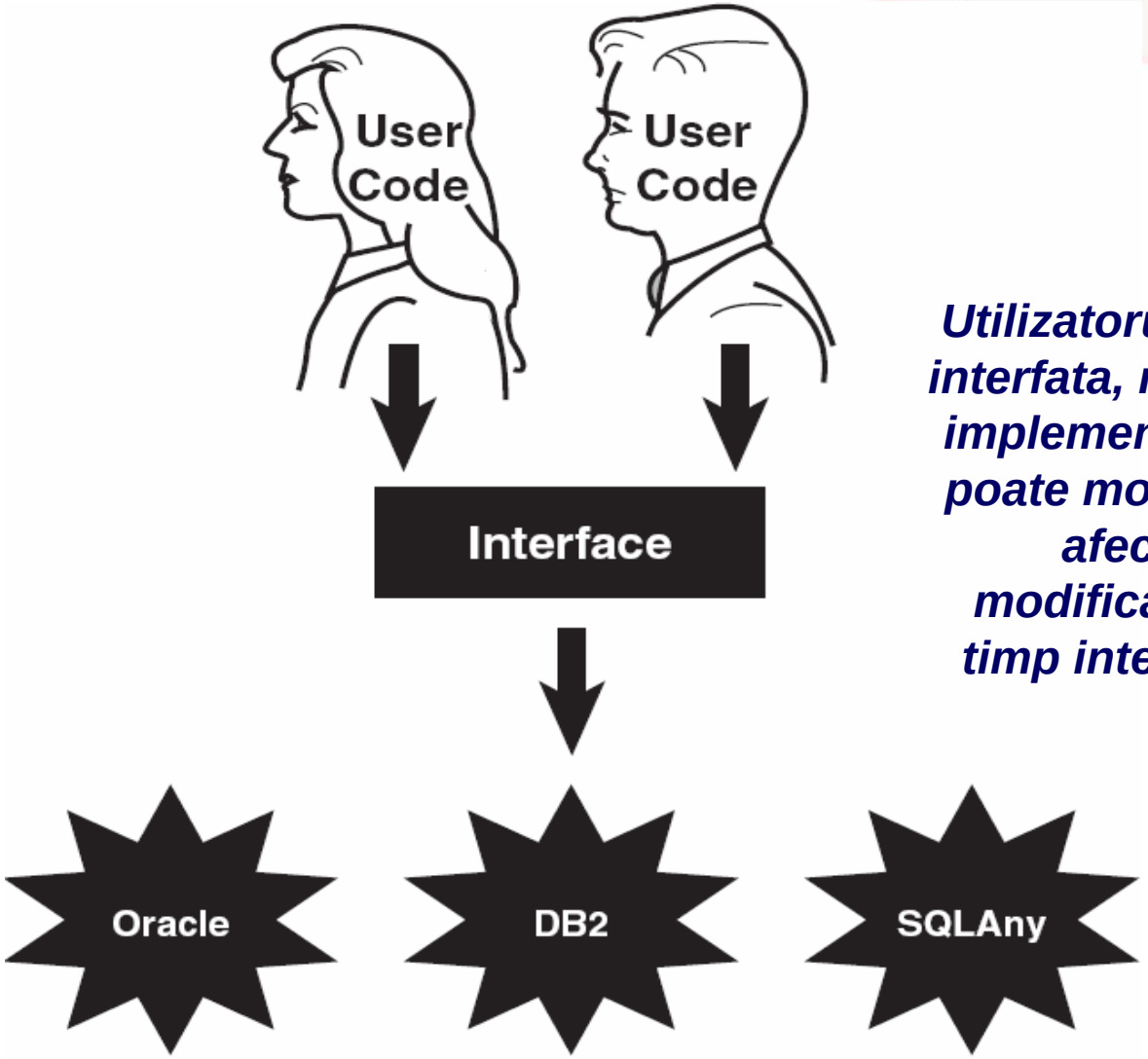


Utilizatorul cunoaste interfata, nu cunoaste implementarea, nu o poate modifica, nu e afectat de modificarea ei cat timp interfata e OK

accesibila si cunoscuta *(publica)*

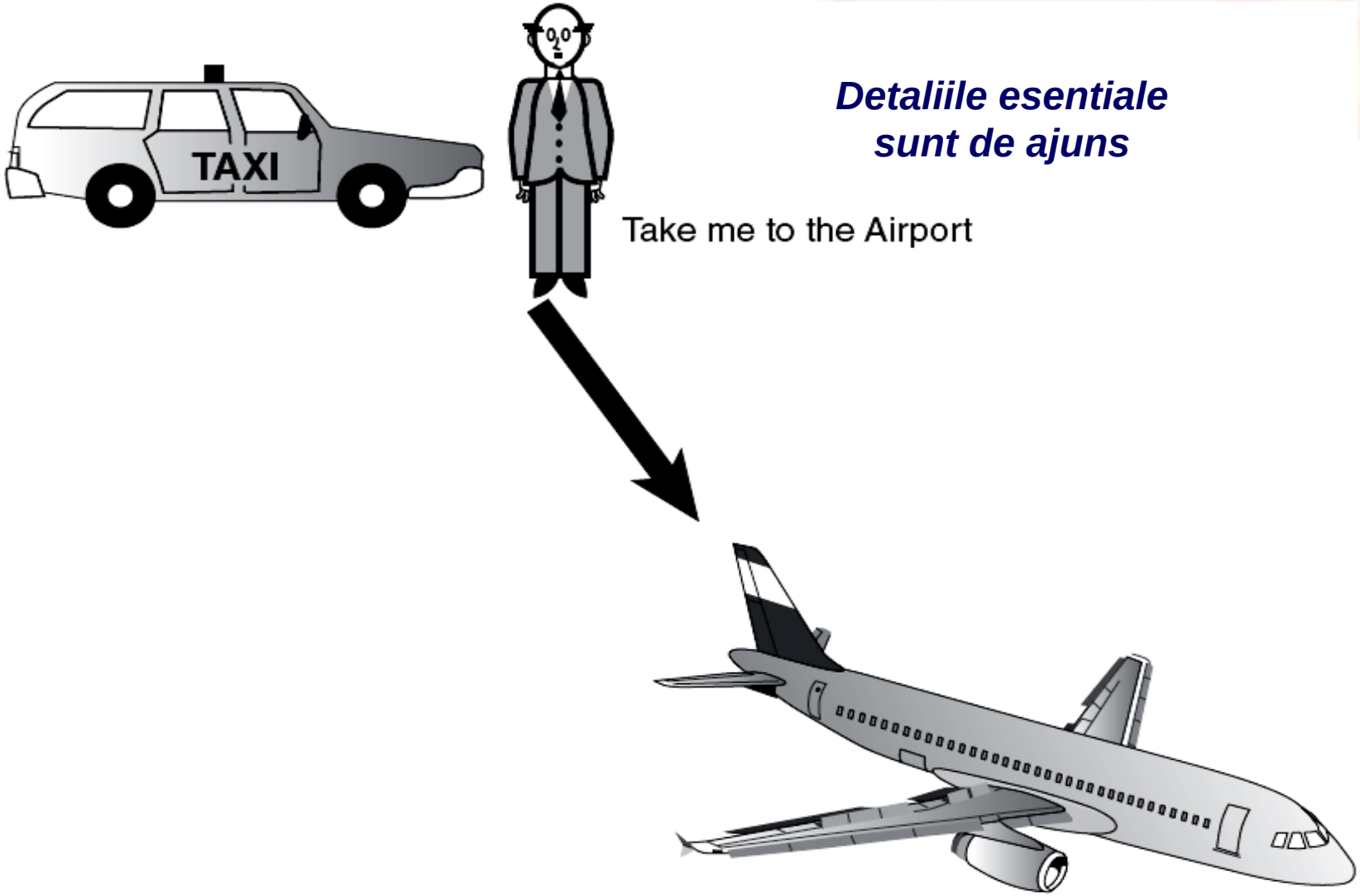
inaccesibila si necunoscuta *(private)*

Interfata si implementarile



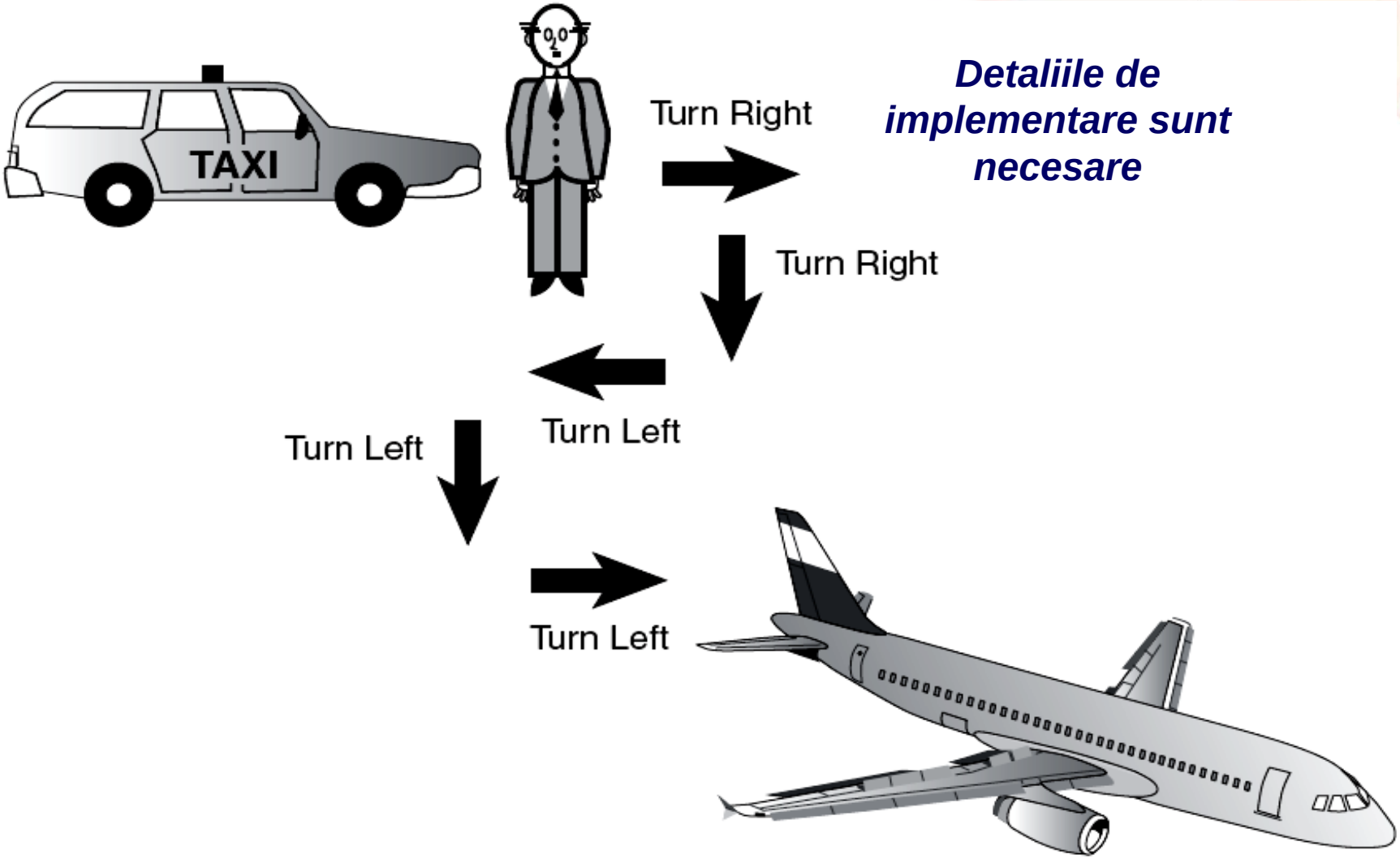
Utilizatorul cunoaste interfata, nu cunoaste implementarea, nu o poate modifica, nu e afectat de modificarea ei cat timp interfata e OK

Interfata de nivel inalt, abstracta



Detaliile esentiale sunt de ajuns

Interfata de nivel jos, mai putin abstracta

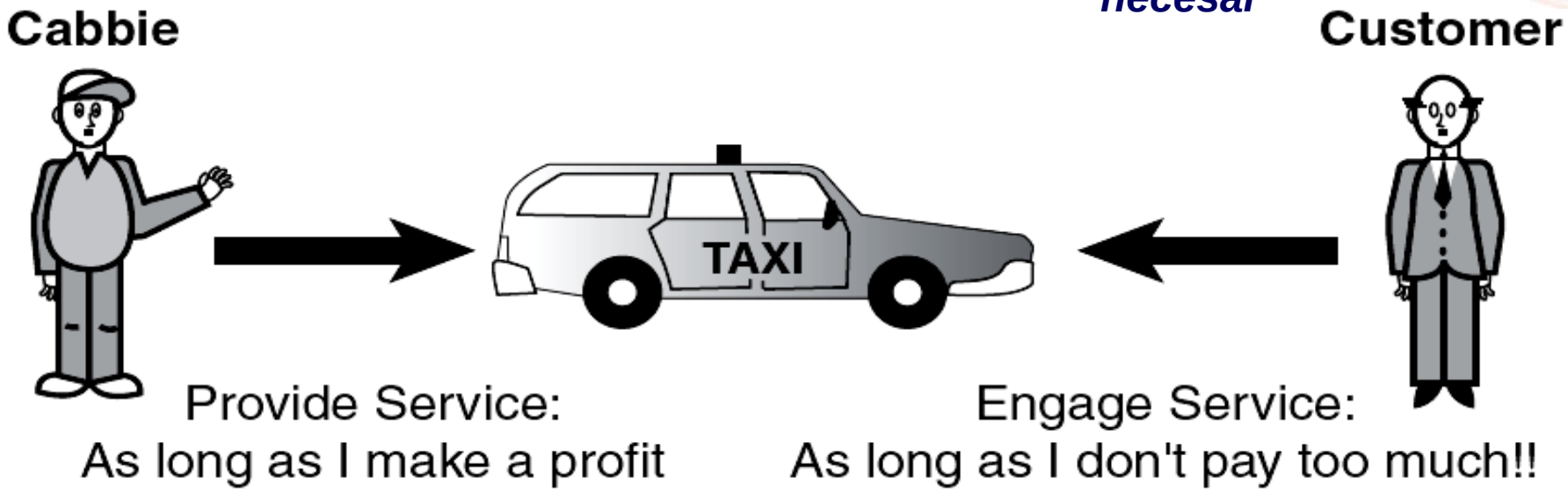


Detaliile de implementare sunt necesare

Servicii oferite (*provided*) si servicii necesare/dorite (*required*)

Serviciul oferit

Serviciul cerut / necesar

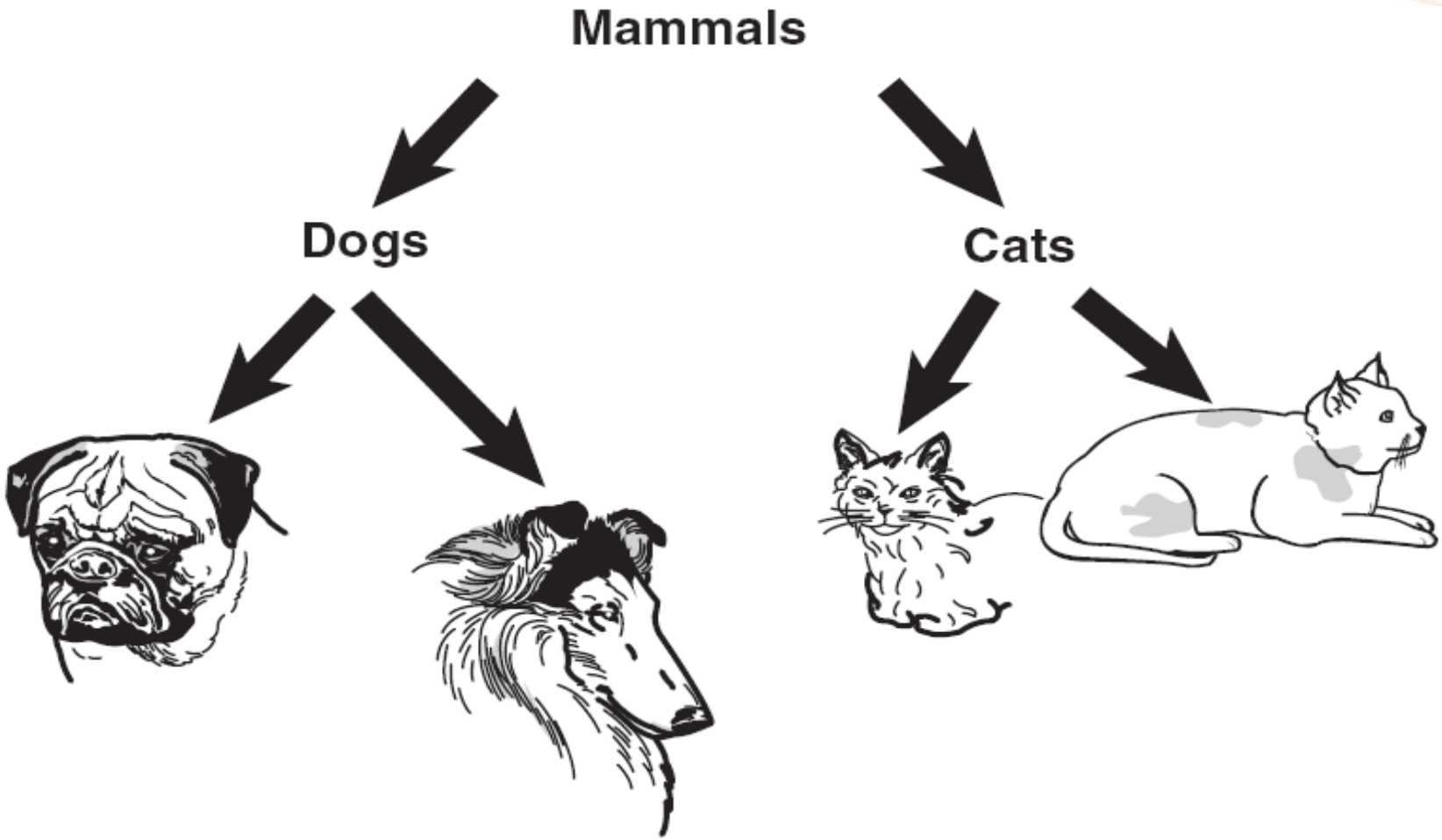


Interfata oferita este ca un contract al comportamentului oferit *(implementatorul e obligat sa il ofere)*

Interfata ceruta trebuie sa fie compatibila cu contractul (cu care se face comparatia, facand abstractie de implementare)

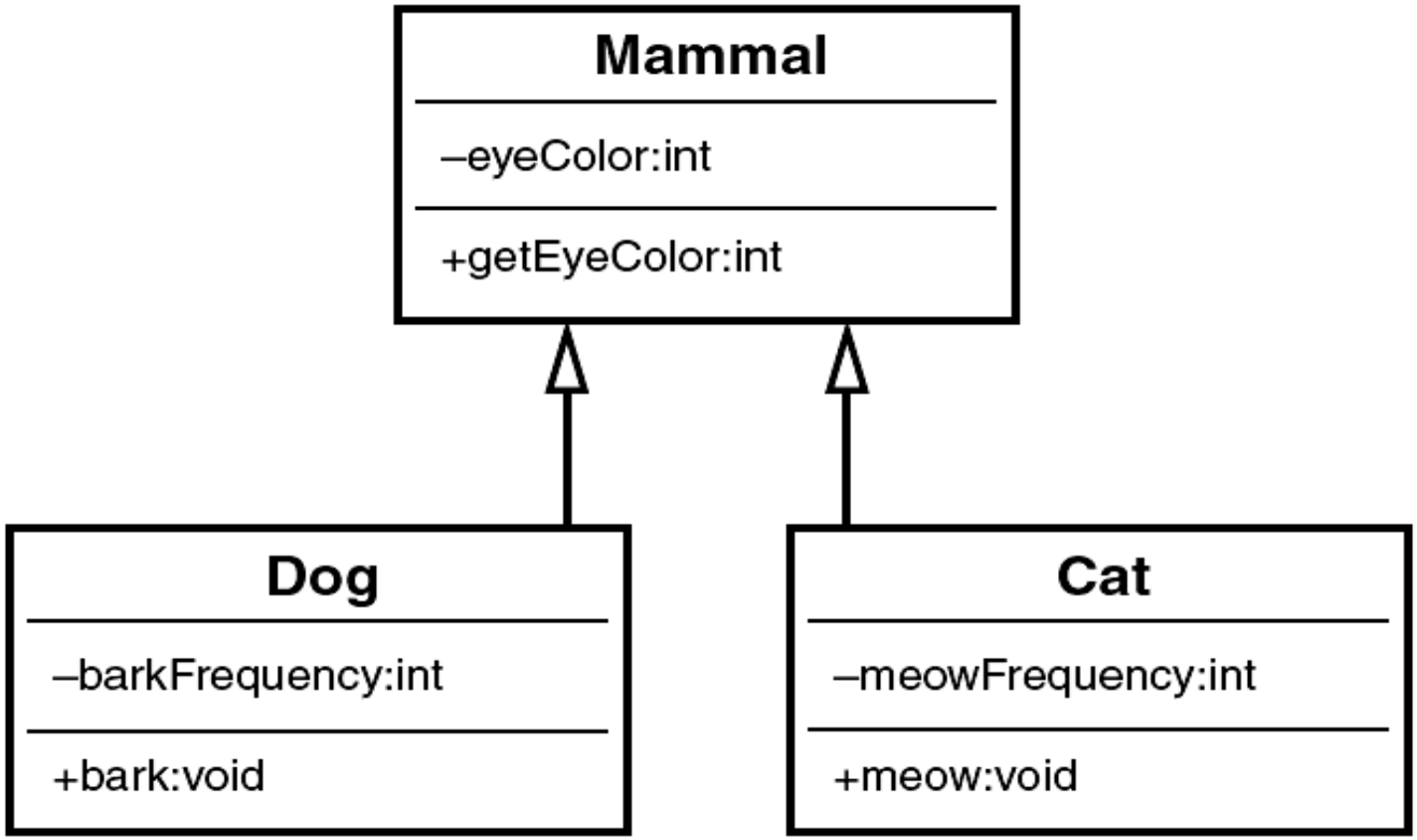
Ierarhii de clase de obiecte bazate pe generalizare

Ierarhie de clase de obiecte din lumea reala



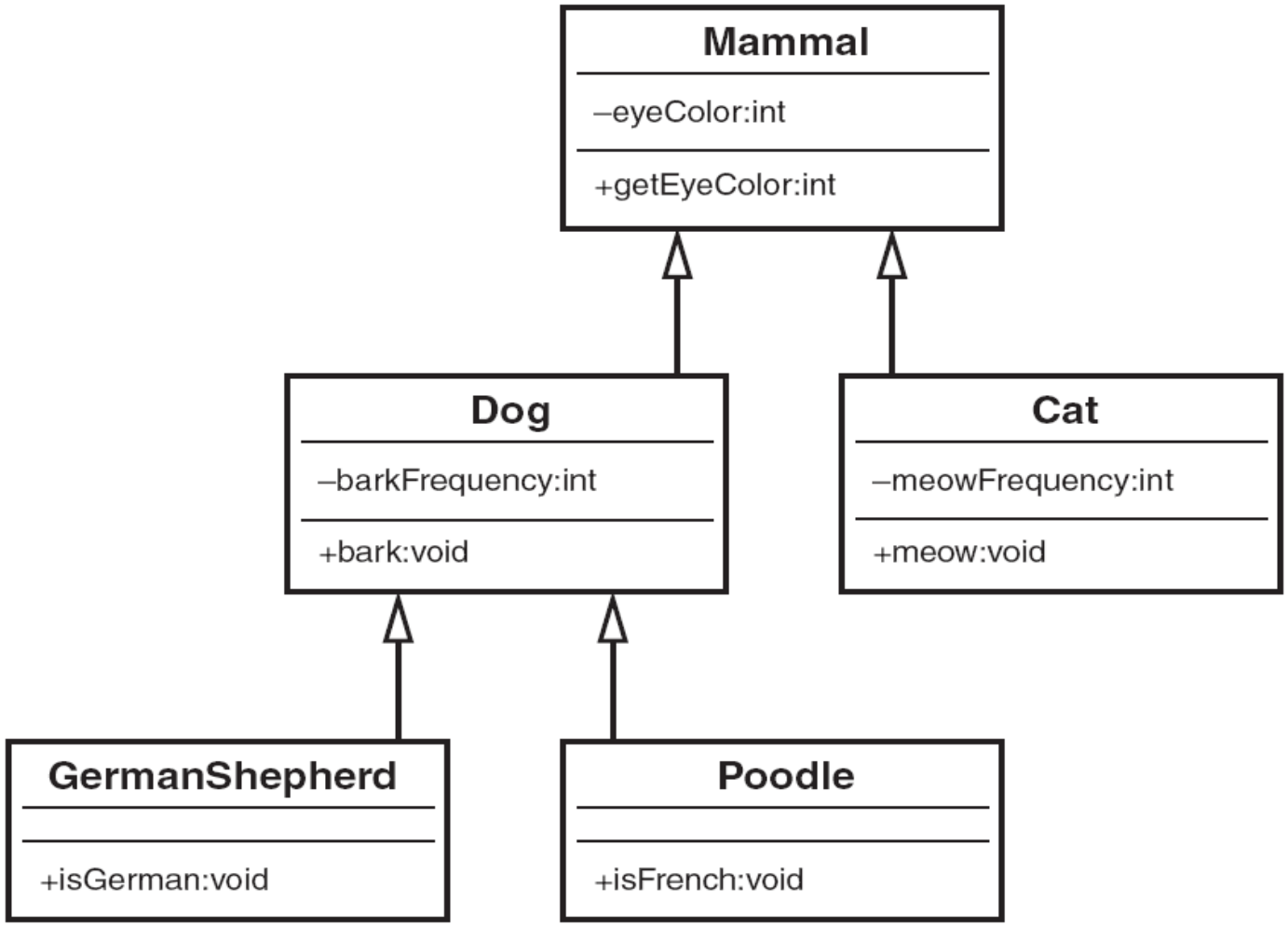
Ierarhii de clase de obiecte bazate pe generalizare

Ierarhie de clase de obiecte software reprezentate vizual in limbajul UML



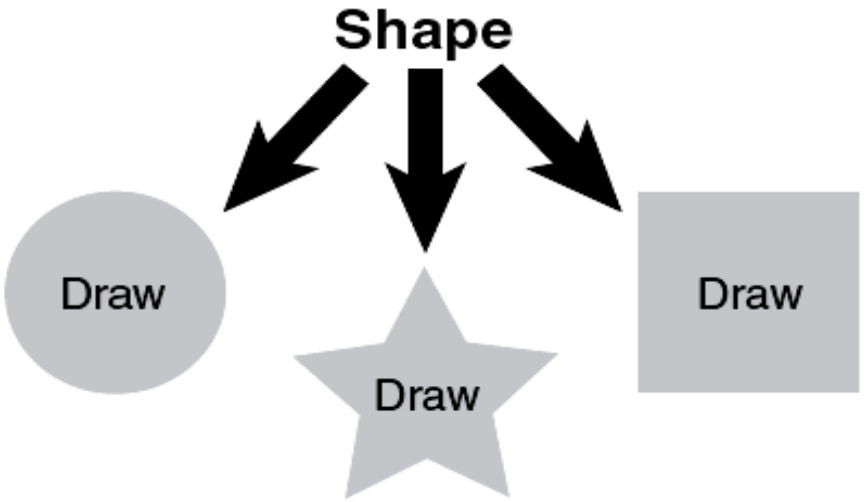
Ierarhii de clase de obiecte bazate pe generalizare

Extinderea ierarhiei de clase prin specializare



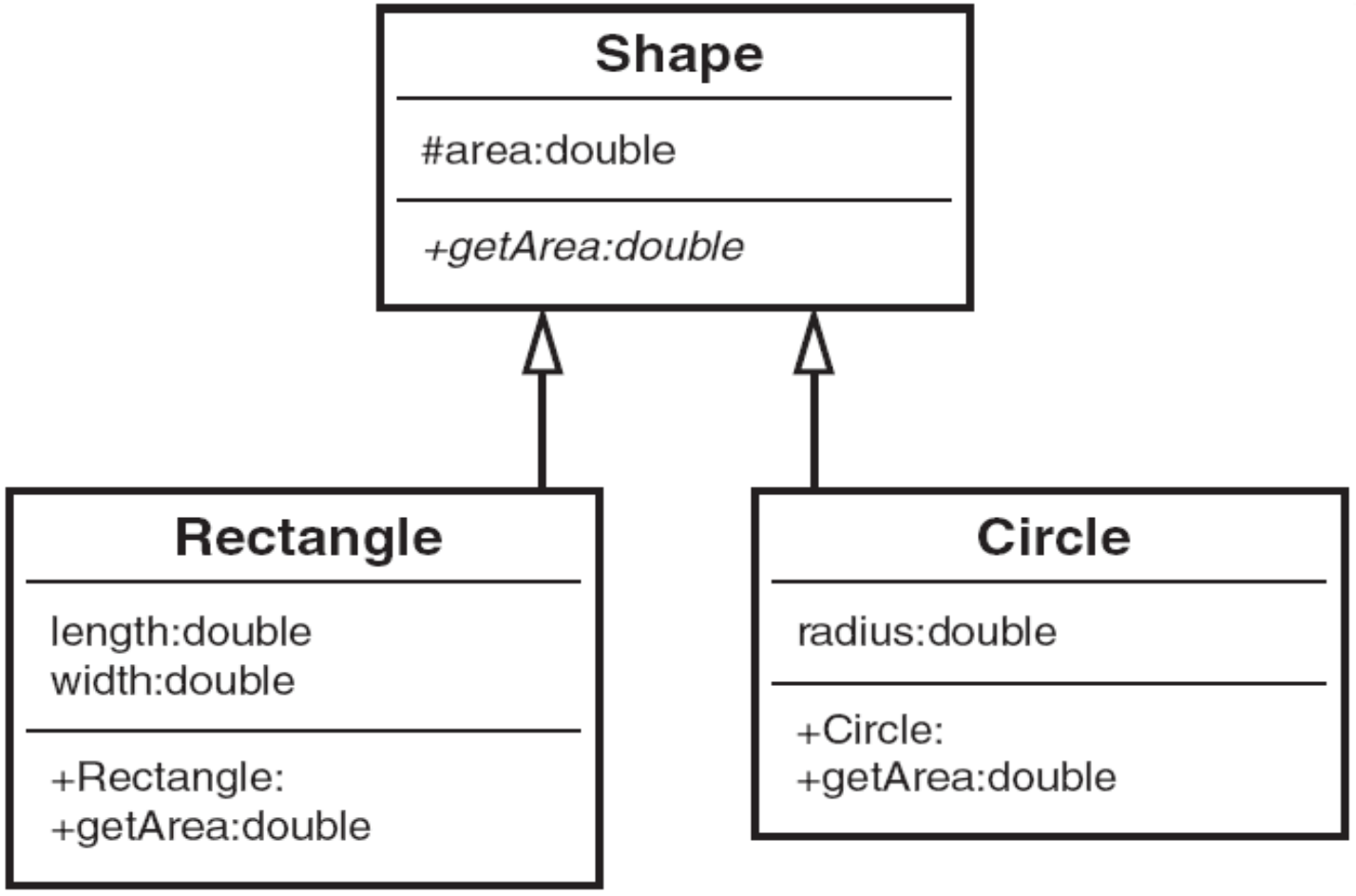
Ierarhii de clase de obiecte bazate pe generalizare

Ierarhie de clase de obiecte din lumea reala

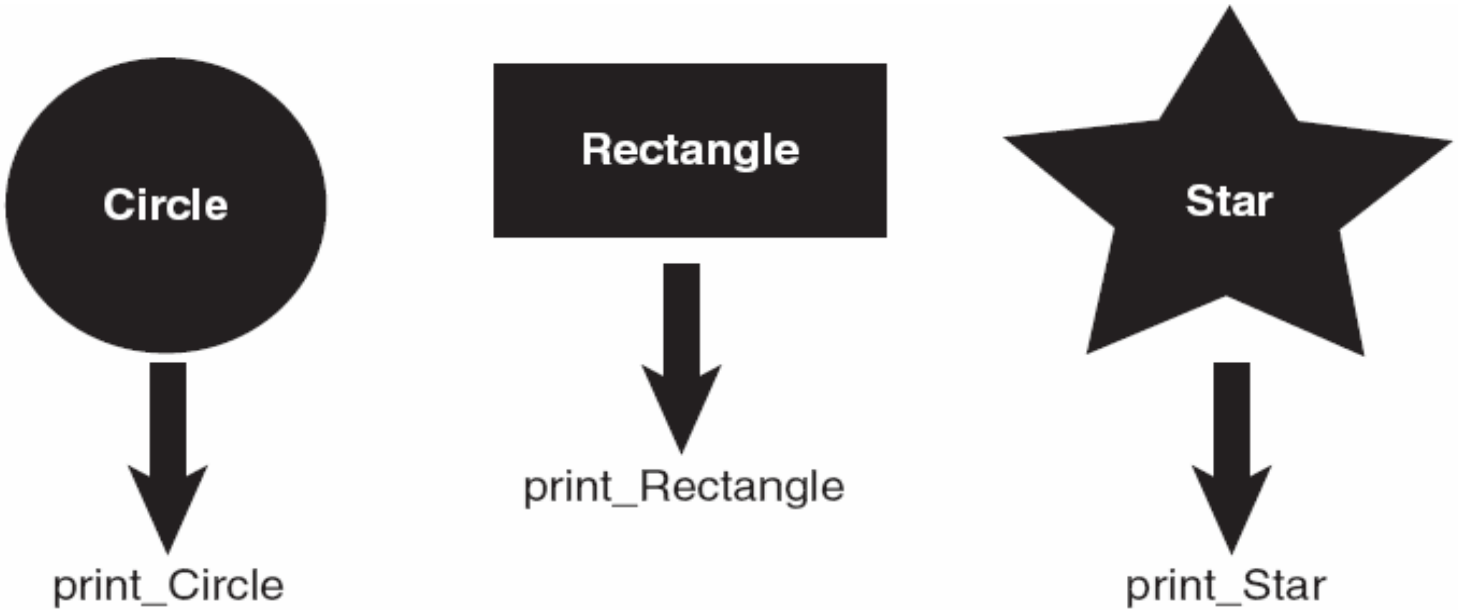


Ierarhii de clase de obiecte bazate pe generalizare

Ierarhie de clase de obiecte software reprezentate vizual in limbajul UML

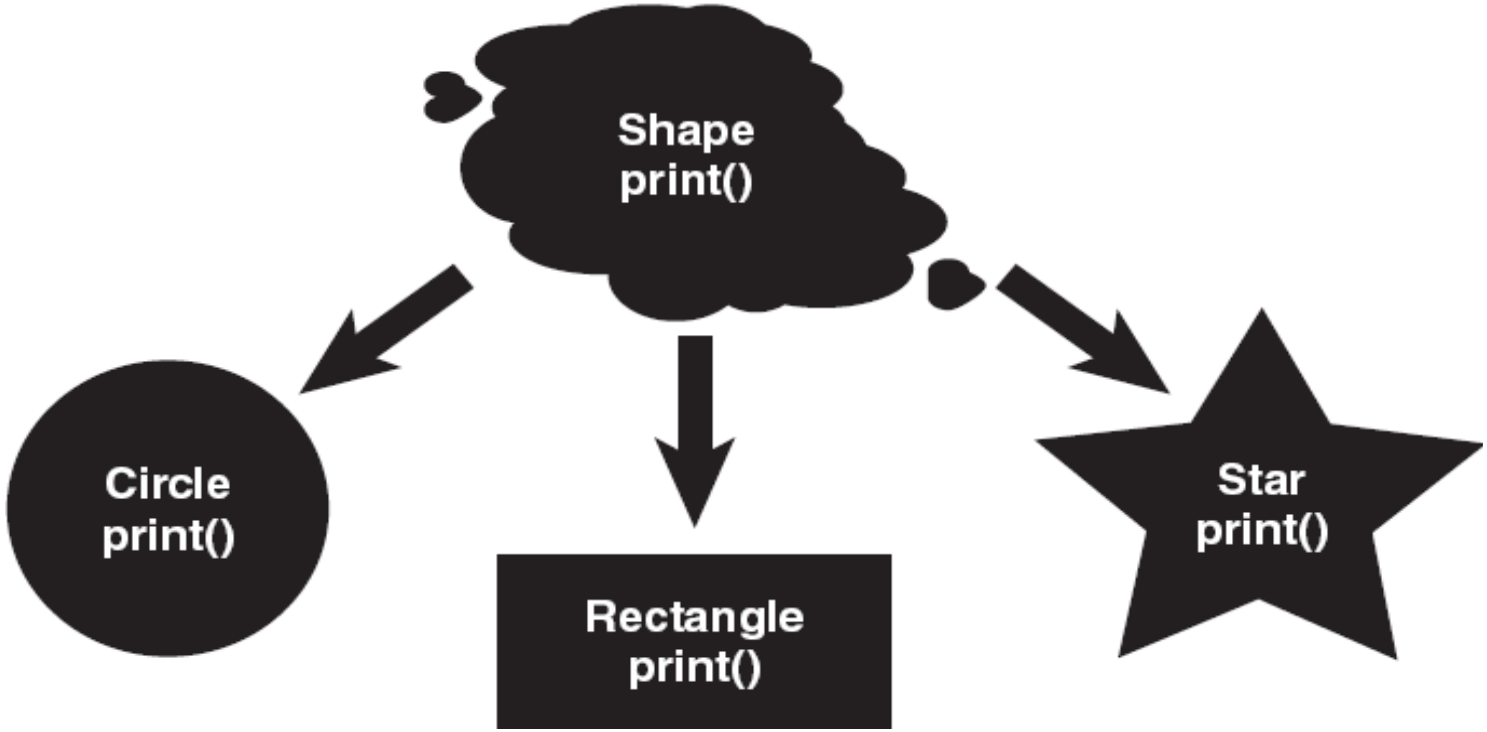


Variabilitatea din versiunea bazata pe orientare functionala

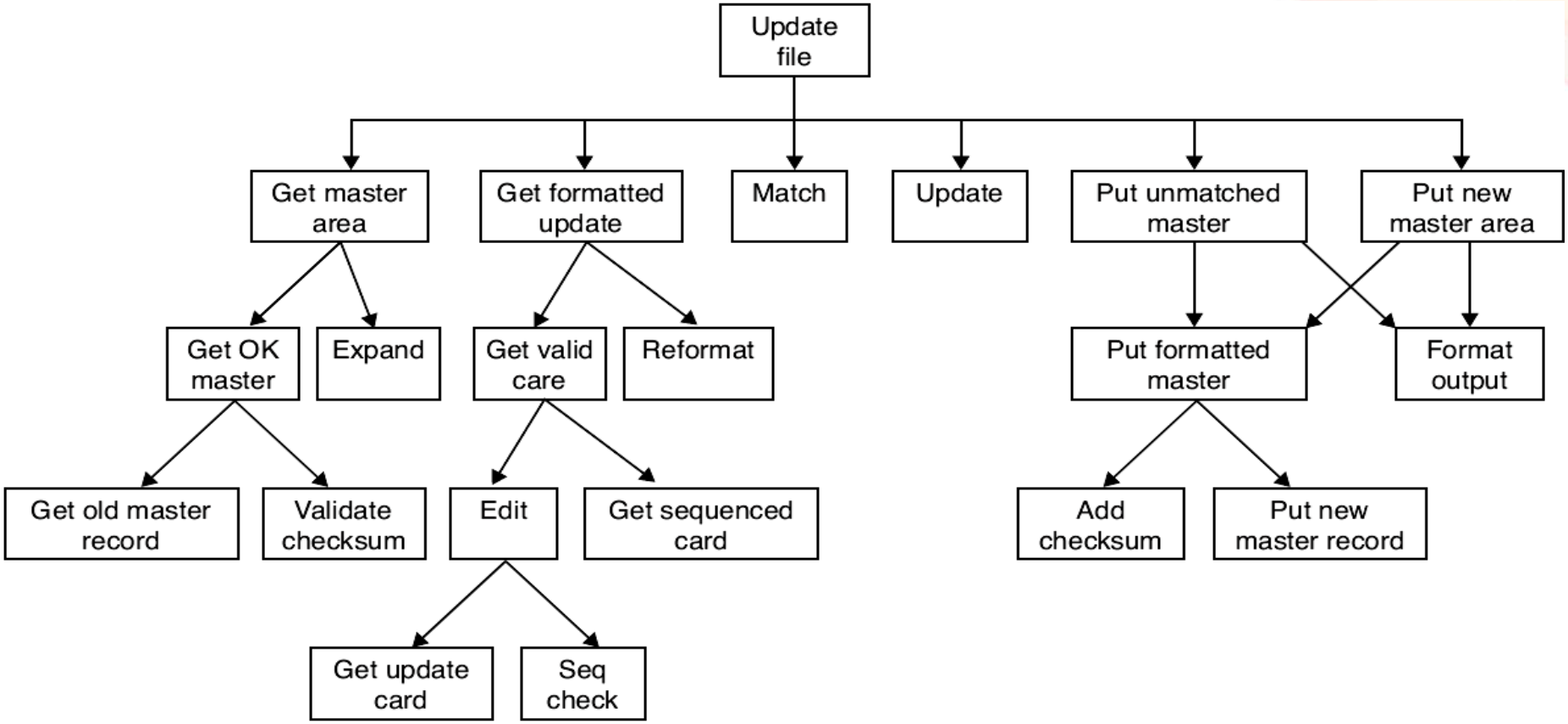


Polimorfismul din versiunea bazata pe orientare spre obiecte

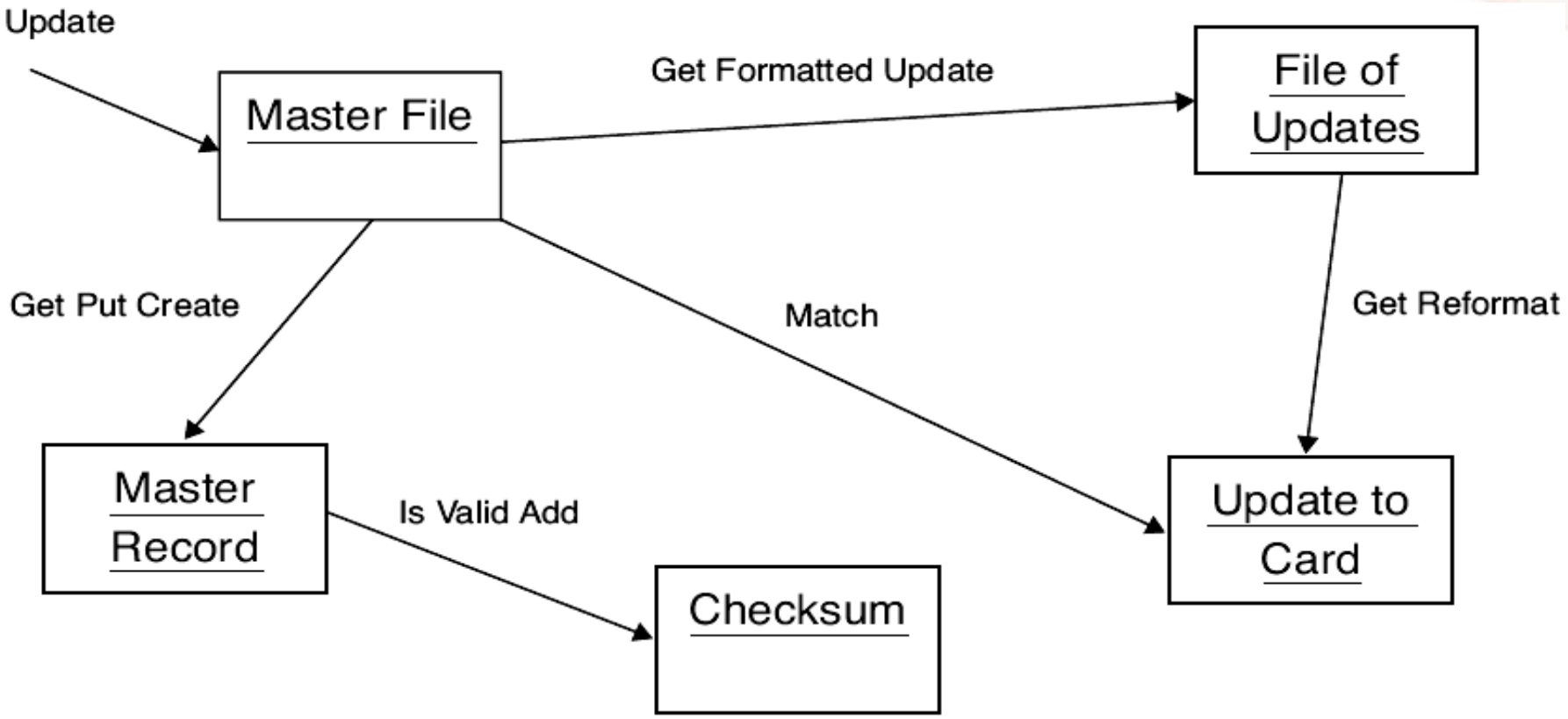
Clasele sunt responsabile pentru comportamentul lor



Descompunere si delegare functionala



Descompunere si delegare orientata spre obiecte



Diagrame UML de clase

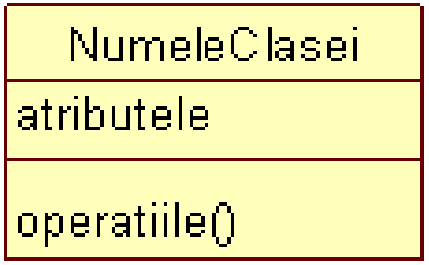
Seria E - Curs 2 – sapt. 2 / Curs 3 – sapt. 3

Diagramele de clase

- exprima la modul general **structura statica a unui sistem**
- în termeni de **clase** si de **relatii între aceste clase**

Clasa:

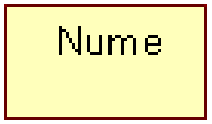
- **descriere abstracta**, condensata, **a unei multimi de obiecte** ale **domeniului aplicatiei**
- reprezentata **in UML** printr-un **dreptunghi cu 3 compartimente**
  - **numele** clasei care trebuie sa spuna **ce reprezinta clasa, NU ce face**
  - **atributele** clasei (elemente de date, variabile membru, campuri)
  - **operatiile** clasei (subprograme, functii membru, metode)



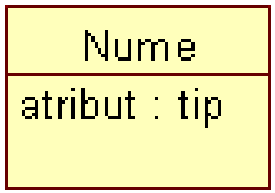
Diagramele de clase

Reprezentarea clasei:

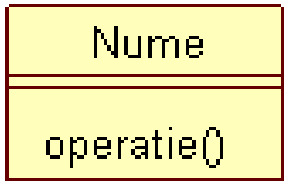
- **contine** întotdeauna **cel putin numele**



- când numele nu a fost stabilit, este recomandabil sa se figureze un nume generic usor de identificat, cum ar fi **DeDefinit**
- **celelalte compartimente pot fi suprimate daca nu au continut semnificativ** pentru contextul diagramei (**suprimarea e pur vizuala**, fara eliminarea cu adevarat a atributelor sau a operatiilor)
  - cand **clasa nu contine operatii** sau doar nu dorim sa le reprezentam ea are **doar doua compartimente**



  - cand **clasa nu contine atribute** sau doar nu dorim sa le reprezentam ea are **compartimentul din mijloc gol**



Diagramele de clase

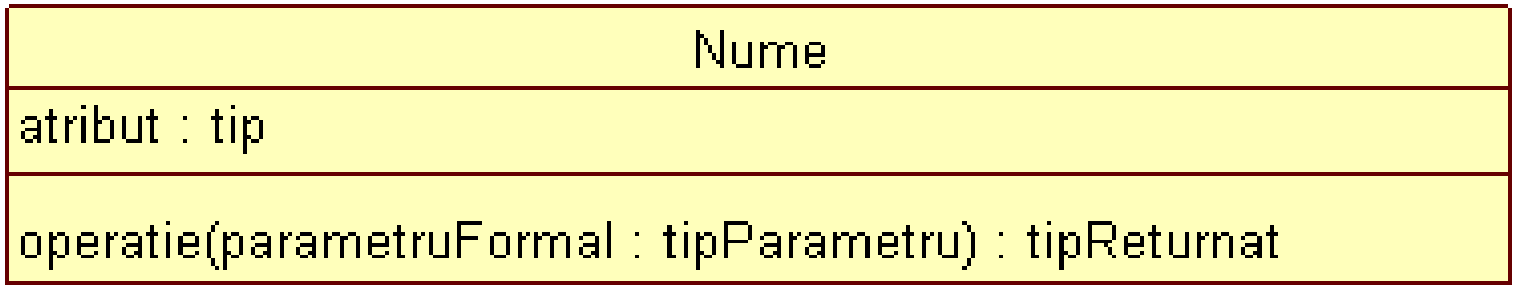
Reprezentarea clasei:

- **formatul complet** al declaratiei unui **atribut** UML este:

```
numeAtributUML : tipAtributUML := valoareInitiala
```

- **formatul complet** al declaratiei **(semnaturii) unei operatii UML** (cu un singur argument) este:

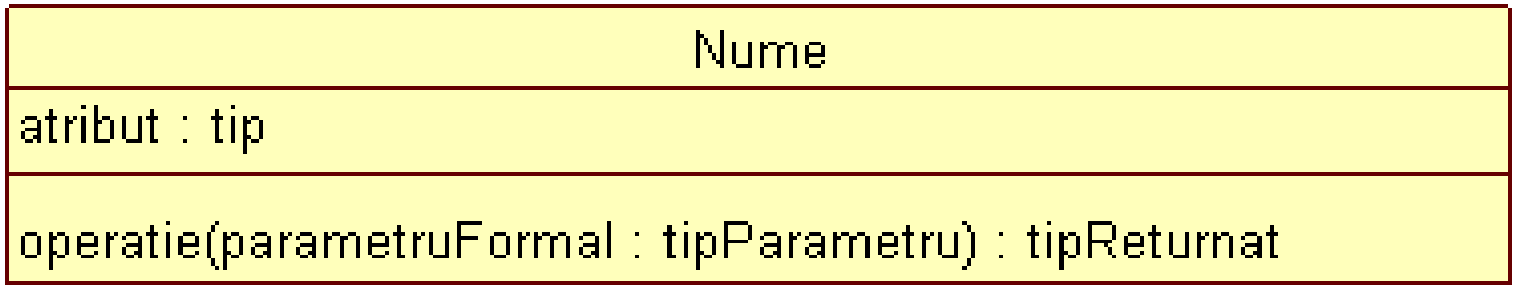
```
numeOperatieUML (numeArgumentUML : tipArgumentUML) : tipReturnatUML
```



Diagramele de clase

Codul Java echivalent

```
1   class Nume { // declaratie tip de date / structura de date
2
3      tip atribut; // declaratie atribut (camp Java)
4
5      // semnatura operatiei (metodei Java)
6      tipReturnat operatie (tipParametru  parametruFormal) {
7
8          // corpul operatiei (metodei Java)
9          // returneaza valoare de tipul tipReturnat
10     }
11  }
```

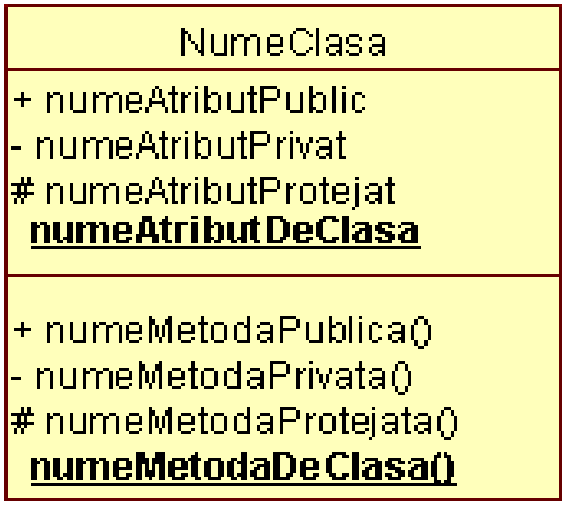


Diagramele de clase

UML defineste 3+1 **niveluri de vizibilitate pentru atribute si operatii**:

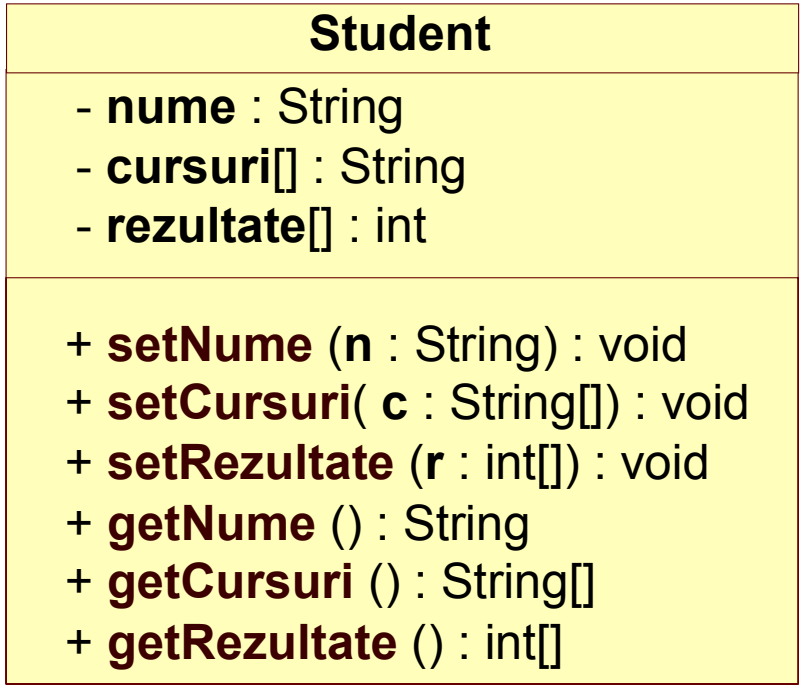
- ***public*** - element vizibil tuturor clientilor clasei (interfata pura, notat "**+**")
- ***private*** - element vizibil în interiorul clasei (implementare pura,notat "**-**")
- ***protected*** - element vizibil subclaselor clasei (notat "**#**")
- ***implementation*** - echivalentul **Java** al *package* (fara notatie)

Atributele si operatii vizibile în toate expresiile lexicale ale clasei, sunt numite ***static***, **de clasa**, si sunt reprezentate prin **sublinierea numelor**



Diagramele de clase

Exemplul unei clase Student in UML



Diagramele de clase – codul clasei Student in Java

```
/**
 * Incapsuleaza informatiile si comportamentul unui Student.
 *
 */
public class Student {
  // Campuri (atribute) private (inaccesibile codurilor exterioare)
  private String nume;
  private String[] cursuri;
  private int[] rezultate;

  // Metode (operatii) publice   (accesibile tuturor codurilor exterioare)
  // Metoda stabilire nume
  public void setNume(String n)          {  nume = n;  }

  // Metoda stabilire cursuri
  public void setCursuri(String[] c)     {  cursuri = c;  }

  // Metoda stabilire rezultate
  public void setRezultate(int[] r)      {  rezultate = r;  }

  // Metoda obtinere nume
  public String getNume()                {  return (nume);  }

  // Metoda obtinere cursuri
  public String[] getCursuri()           {  return (cursuri);  }

  // Metoda obtinere rezultate
  public int[] getRezultate()            {  return (rezultate);  }
}
```

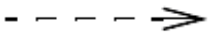
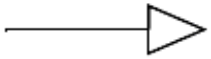
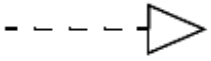
Informatii ascunse = stare ascunsa (campuri private)

Interfata publica = servicii oferite (semnaturi metode)

Implementare ascunsa = comportament ascuns (coduri interne ale metodelor)

Relatii intre clase.
Asociere, agregare, compunere

Tipuri de relatii intre clase

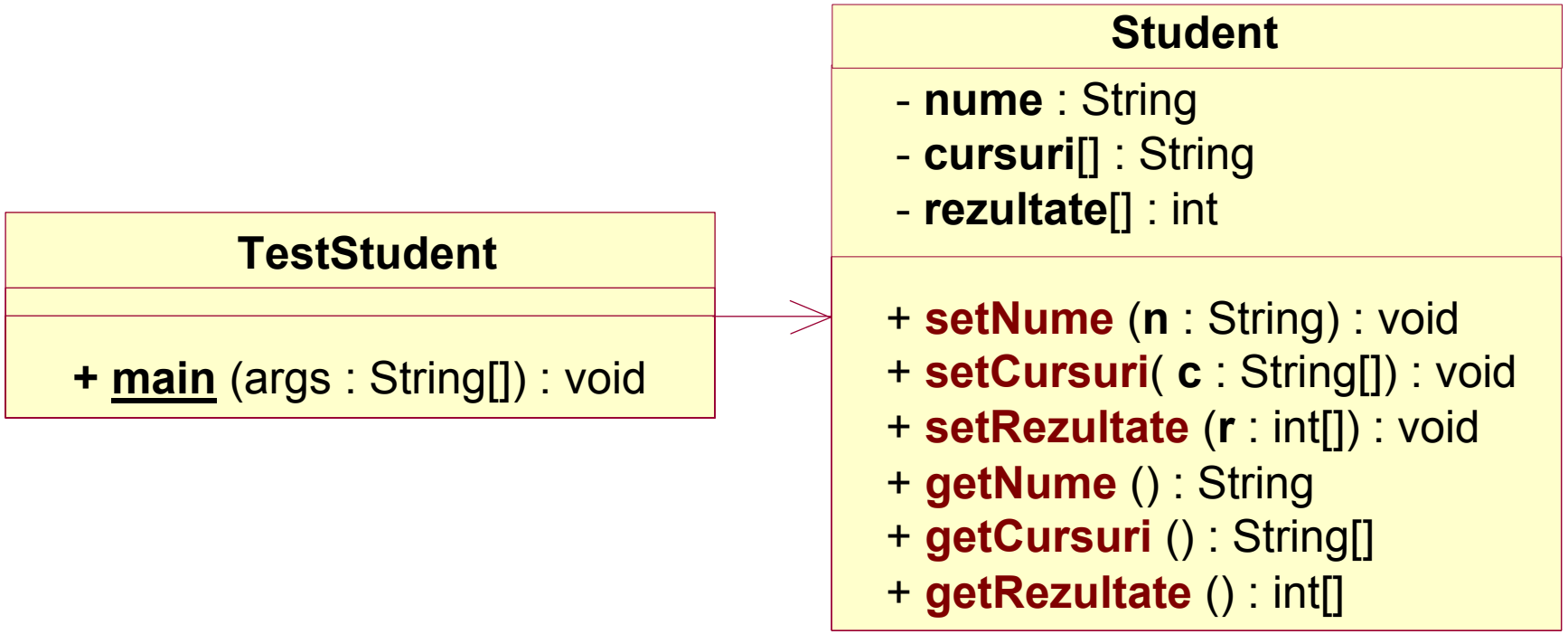
Relationship	*Function*	*Notation*
association	A description of a connection among instances of classes	
dependency	A relationship between two model elements	
generalization	A relationship between a more specific and a more general description, used for inheritance and polymorphic type declarations	
realization	Relationship between a specification and its implementation	
usage	A situation in which one element requires another for its correct functioning	

Exemplu de clasa utilizator pentru clasa tinta anterioara

```
public class TestStudent {

  // Metoda de test. Punct de intrare in program.
  public static void main(String[] args) {

     // Crearea unui nou Student, fara informatii
     Student st1 = new Student();

     // Initializarea campurilor noului obiect
     st1.setNume("Xulescu Ygrec");
     String[] crs = {"CID", "AMP"};
     st1.setCursuri(crs);
     int[] rez = {8, 9};
     st1.setRezultate(rez);

     // Utilizarea informatiilor privind Studentul
     System.out.println("Studentul " + st1.getNume() + ":");
     for (int i=0; i<rez.length; i++)
         System.out.println("- are nota " + st1.getRezultate()[i]
                     + " la disciplina " + st1.getCursuri()[i]);
  }
}    // Rezultatul:  Studentul Xulescu Ygrec:
     //              - are nota 8 la disciplina CID
     //              - are nota 9 la disciplina AMP
```

Relatii intre clase

Exemplu de asociere intre clase



Exemplu de clasa

```
public class Point {
   // atribute (variabile membru)
   private int x;
   private int y;

   // operatie care initializeaza atributele = constructor Java
   public Point(int abscisa, int ordonata) {
      x = abscisa;
      y = ordonata;
   }

   // operatii care modifica atributele = metode (functii membru)
   public void moveTo(int abscisaNoua, int ordonataNoua) {
      x = abscisaNoua;
      y = ordonataNoua;
   }
   public void moveWith(int deplasareAbsc, int deplasareOrd) {
      x = x + deplasareAbsc;
      y = y + deplasareOrd;
   }
   // operatii prin care se obtin valorile atributelor = metode
   public int getX() {  return x;  }
   public int getY() {  return y;  }
}
```

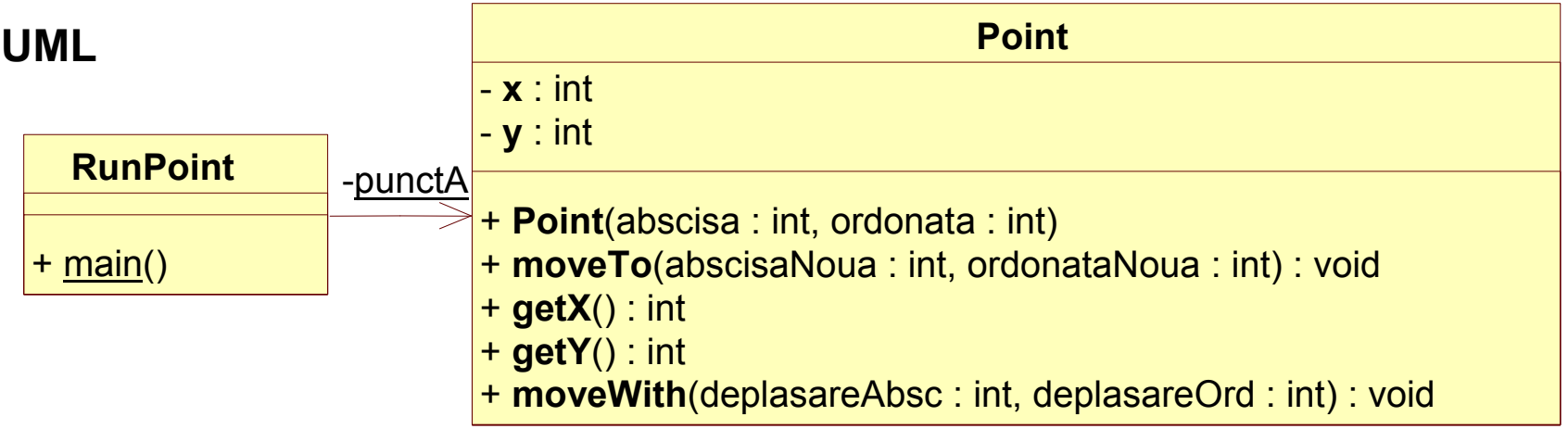
Declaratii (specificare) **atribute**

Semnaturi (declaratii, specificari) **operatii** + **Implementari** (corpuri) **operatii**

Exemplu de clasa utilizator pentru clasa anterioara

```
// clasa de test pentru clasa Point
public class RunPoint {
   private static Point punctA;                    // atribut de tip Point

   public static void main(String[] args) {        // declaratie metoda
      // corp metoda
      punctA = new Point(3, 4);  // alocare si initializare atribut punctA

      punctA.moveTo(3, 5);          // trimitere mesaj moveTo() catre punctA

      punctA.moveWith(3, 5);        // trimitere mesaj moveWith() catre punctA
   }
}
```

In UML



Exemplu de clasa tinta

```
public class DatePersonale  {

   // Campuri ascunse
   private String nume;
   private String initiale;
   private String prenume;
   private int anNastere;

   // Constructori
   public DatePersonale(String n, String i, String p, int an) {
      nume = new String(n); // copiere„hard”a obiectelor primite parametri
      initiale = new String(i);  // adica se copiaza obiectul camp cu camp
      prenume = new String(p);   // nu doar referintele ca pana acum
      anNastere = an;
   }

   // Interfata publica si implementarea ascunsa
   public String getNume() {      return (nume);   }
   public String getPrenume() {   return (prenume);   }
   public int getAnNastere() {    return (anNastere);   }

   public String toString() {    // forma „String” a campurilor obiectului
      return (nume+ " " +initiale+ " " +prenume+ " (" +anNastere+ ")");
   }
}
```

Exemplu de clasa tinta

```
public class SituatieCurs  {

   // Campuri ascunse
   private int nota = 0;                    // initializare implicita
   private String denumire;

   // Constructor
   public SituatieCurs(String d) { denumire = new String(d); } // copiere
                                 // „hard”, se initializeaza doar denumire

   // Interfata publica si implementarea ascunsa
   public void notare(int n) {     nota = n; }     // se adauga nota
   public int nota() {             return(nota); } // se returneaza nota

   public String toString() {              // forma „String” a campurilor
      if (nota==0)
         return ("Disciplina " + denumire + " nu a fost notata");
      else
         return("Rezultat la disciplina " + denumire + ": " + nota);
   }
}
```

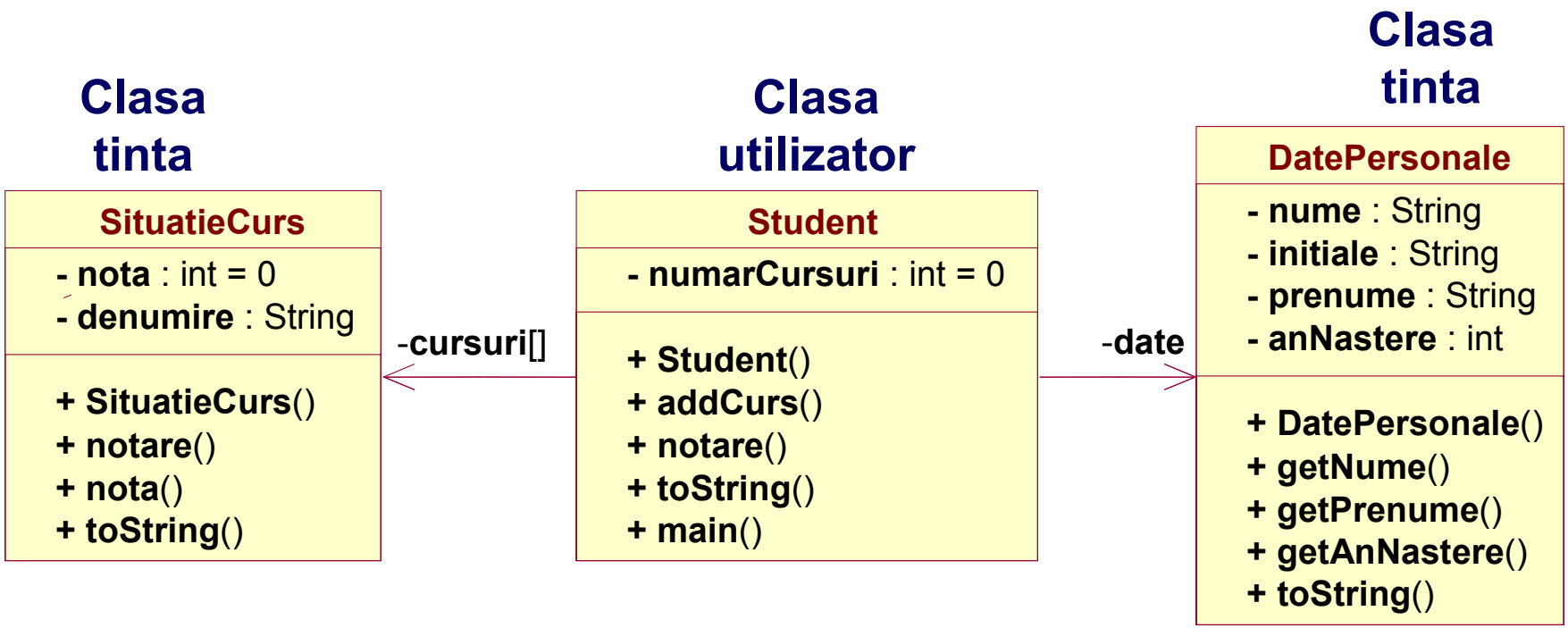
Exemplu de clasa utilizator pentru clasele tinta anterioare

```
public class Student { // rescrisa pentru a putea utiliza clasele anterioare
  // Campuri ascunse
  private DatePersonale date;
  private SituatieCurs[] cursuri;
  private int numarCursuri = 0;      // initializare implicita
  // Constructori
  public Student(String nume, String initiale, String prenume, int anNastere){
     date = new DatePersonale(nume, initiale, prenume, anNastere);
     cursuri = new SituatieCurs[10]; // se initializeaza doar date si cursuri
  }
  // Interfata publica si implementarea ascunsa
  public void addCurs(String nume) { // se adauga un nou curs
     cursuri[numarCursuri++] = new SituatieCurs(nume);
  }
  public void notare(int numarCurs, int nota) {
     cursuri[numarCurs].notare(nota); // se adauga nota cursului specificat
  }
  public String toString() {          // forma „String” a campurilor
     String s = "Studentul " + date + " are urmatoarele rezultate:\n";
     for (int i=0; i<numarCursuri; i++)
        s = s + cursuri[i].toString() + "\n";
     return (s);
  }
}
```

Exemplu de clasa utilizator pentru clasele tinta anterioare

In UML

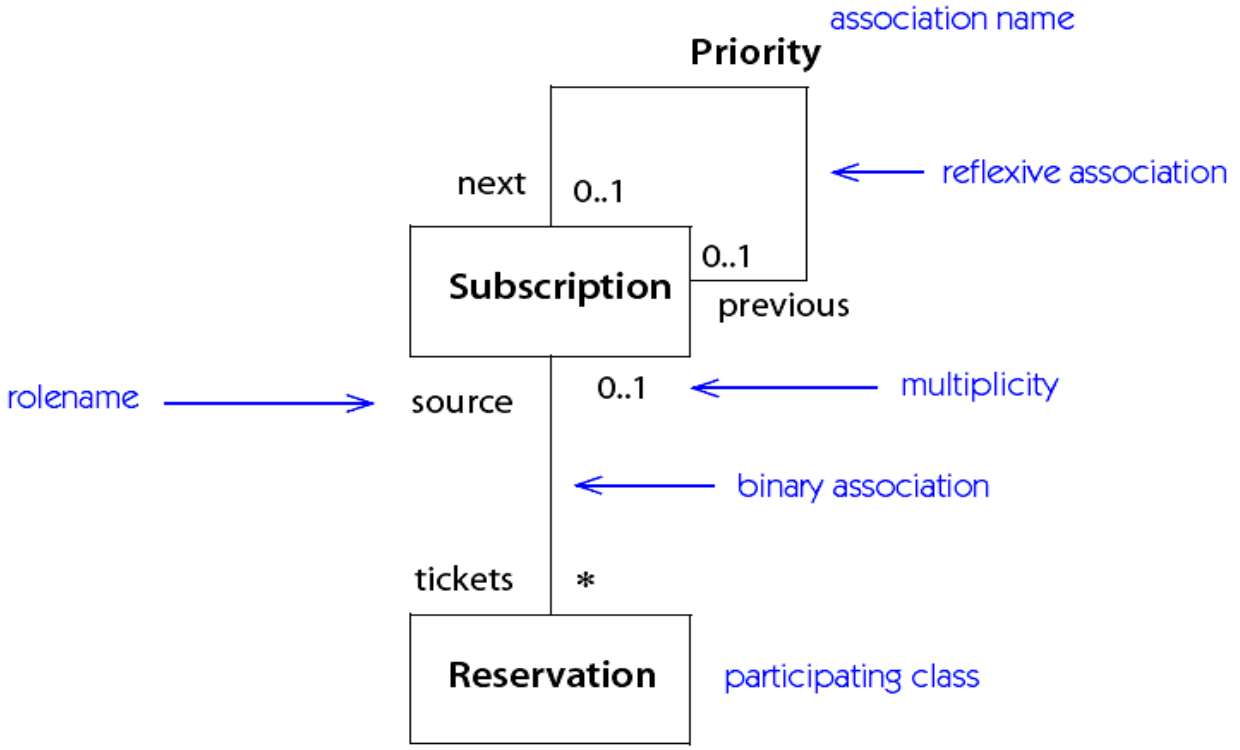
- se observa ca **atributele** numite **cursuri** si **date** sunt **reprezentate ca asocieri intre clase**



Relatii intre clase

Asocierile

- sunt **relatii structurale intre clase** de obiecte
- majoritatea sunt **binare, conectand 2 clase**
- sunt reprezentate prin **linii care unesc clasele asociate**



Relatii intre clase

Codul Java echivalent

```
1    public class Subscription {
2
3       // referinte catre alte obiecte de acelasi tip
4       Subscription next;
5       Subscription previous;
6
7       // referinta catre "colectie" de obiecte de alt tip
8       Reservation[] tickets;
9       // sau  ArrayList<Reservation> tickets;  etc.
10   }
```

```
1    public class Reservation {
2
3       // referinta catre obiect de alt tip
4       Subscription source;
5    }
```

Relatii intre clase

Rolul descrie **modul si numele sub care o clasa vede o alta clasa** dincolo de o asociere

Multiplicitatea este o **informatie detinuta de rol** sub forma unei expresii naturale:

Simbol	Multiplicitate	Numar de obiecte
1	Unu si numai unu (exact unu)	**{ 1 }**
0..1	Zero sau unu	**{ 0, 1 }**
M..N	De la M la N (intregi naturali)	**{ M, M+1, …, N-1, N }**
*****	De la 0 la mai multe (neprecizat)	**{ 0, 1, …, K }**
0..*	De la 0 la mai multe (neprecizat)	**{ 0, 1, …, K }**
1..*	De la 1 la mai multe (neprecizat)	**{ 1, 2, …, K }**

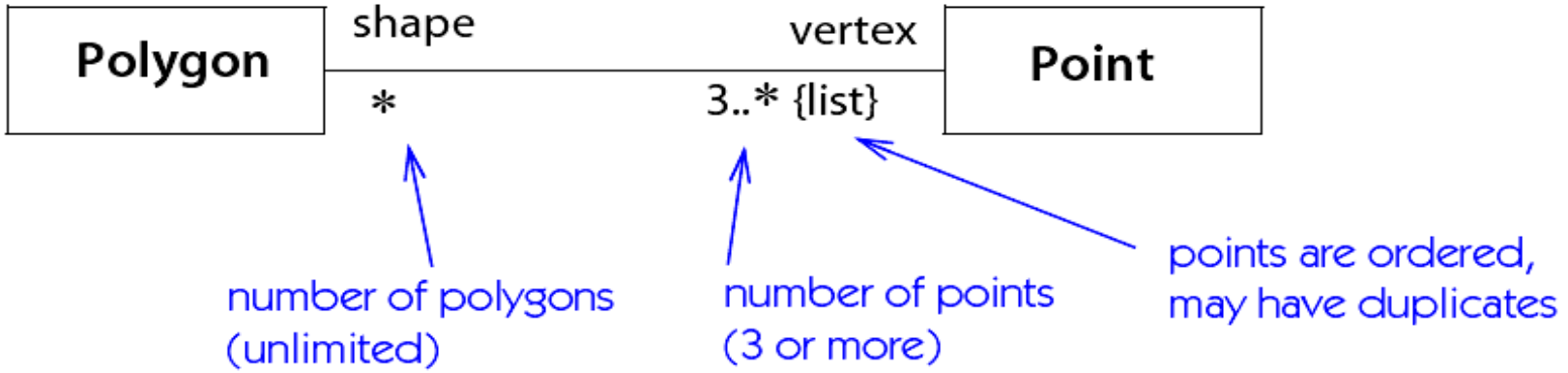
Constrangeri:

set	unordered, unique elements (default)
bag	unordered, nonunique elements
ordered set	ordered, unique elements
list (or sequence)	ordered, nonunique elements

Relatii intre clase

Exemple de multiplicitate

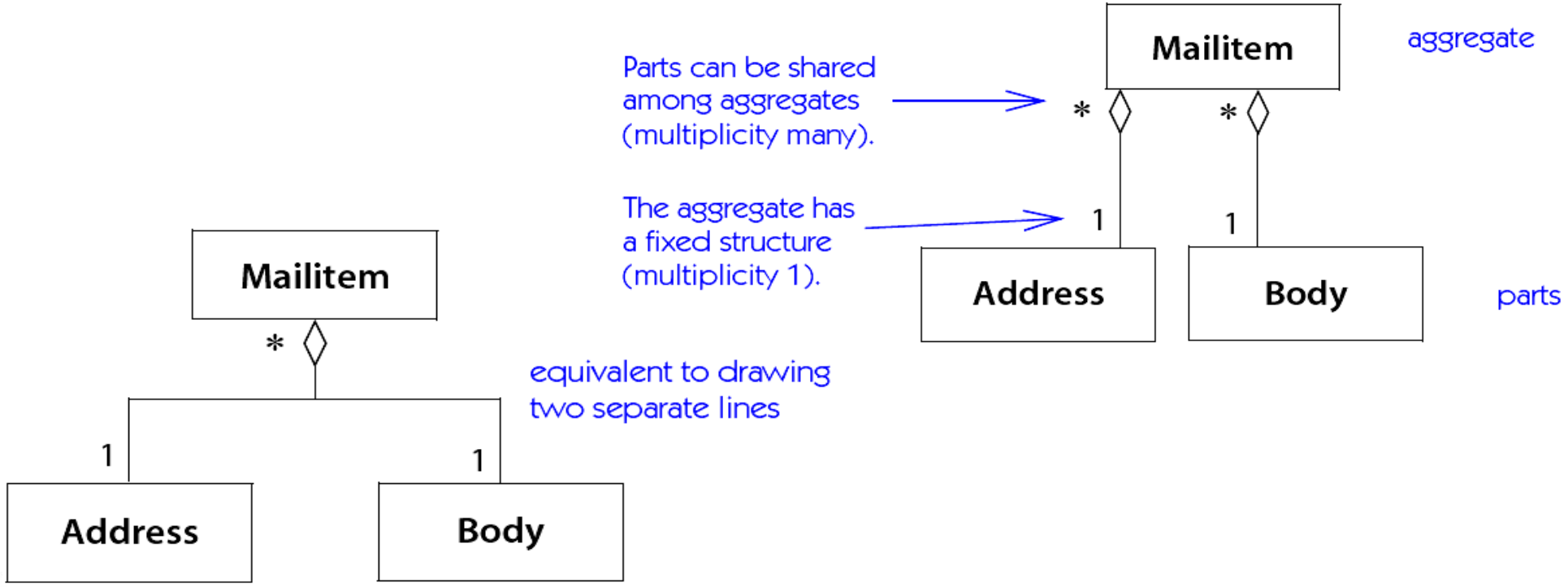
0..1	optional value
1	exactly one
0..*	any number of elements, unordered, unique
* {list}	any number of elements, ordered, duplicates
1..* {ordered}	one or more, ordered, unique
1..6 {bag}	between 1 and 6, unordered, duplicates



Relatii intre clase

Agregarea

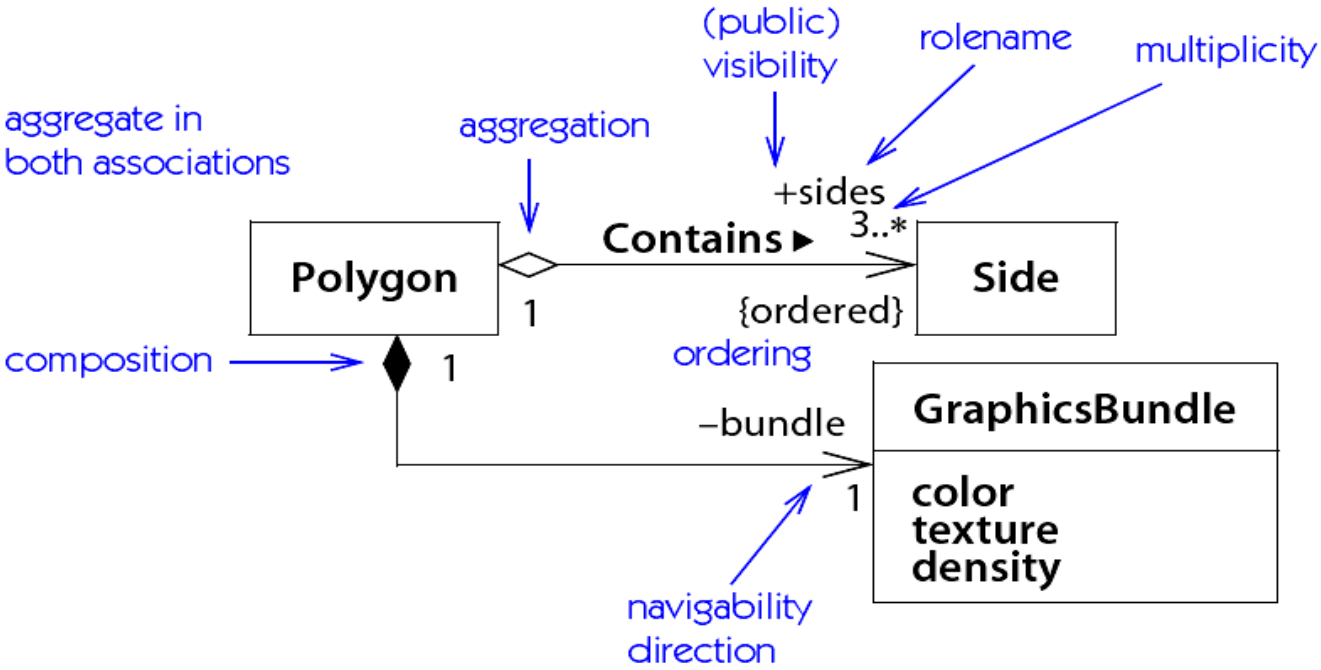
- **asociere asimetrica in care una dintre extremitati joaca un rol predominant** in raport cu celelata extremitate
- se reprezinta prin **adaugarea unui mic romb rolului agregatului**.
- oricare ar fi multiplicitatea, ea priveste **doar un singur rol al unei asocieri**



Relatii intre clase

Compunerea

- **caz particular de agregare prin continere fizica**
- reprezentata in diagrame printr-un **romb de culoare neagra**



Compunerea si atributele sunt **echivalente semantic** (din punct de vedere al semnificatiei) adica **reprezentarile lor grafice** sunt **interschimbabile**

Relatii intre clase

Codul Java echivalent

```
1    public class Polygon {
2
3       // referinte catre alte obiecte = atribute
4       Side[] sides;              // parte agregata
5       GraphicsBundle bundle;     // componenta
6    }
```

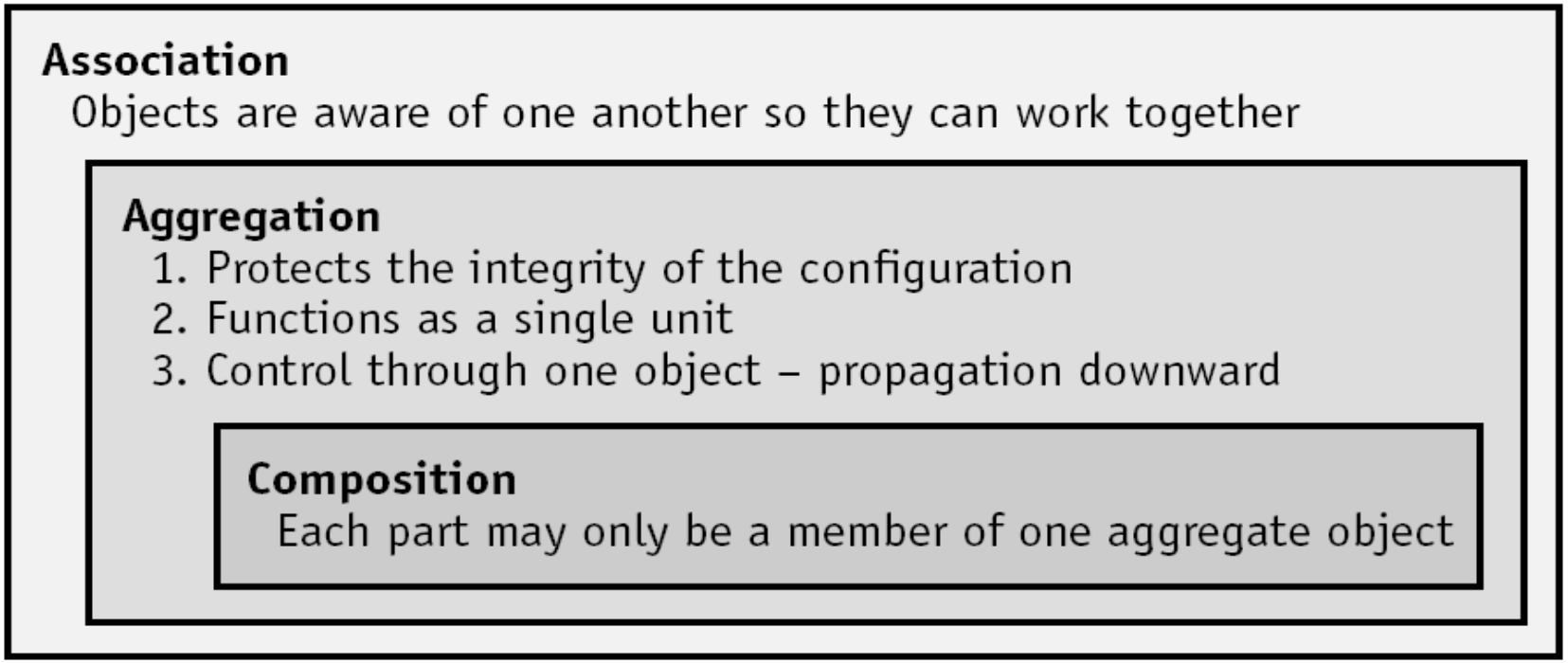
```
1    public class GraphicsBundle {
2
3       // referinte catre alte obiecte = atribute
4       ColorType color;
5       TextureType texture;
6       DensityType density;
7    }
```

Relatii intre clase

Asocierea are doua cazuri speciale

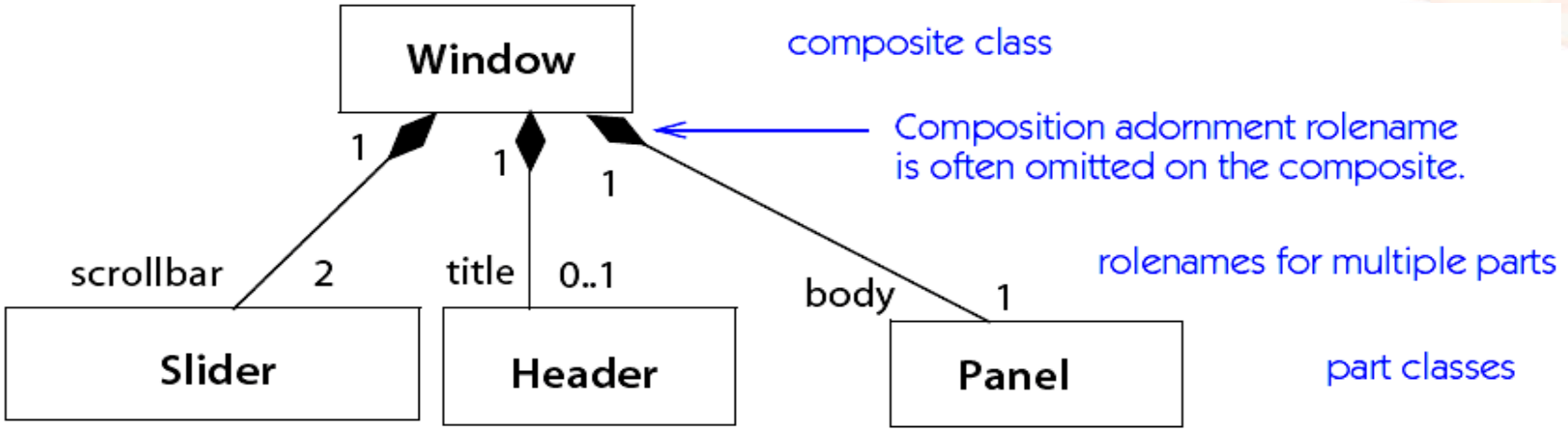
- **agregarea** (relatia de **subordonare a unei clase de catre o clasa** numita **agregat**)

- **compunerea** (forma **puternica** de agregare, de tip **intreg-parte**, in care **partile**, numite **componente**, **apartin strict intregului**, numit **composit**)

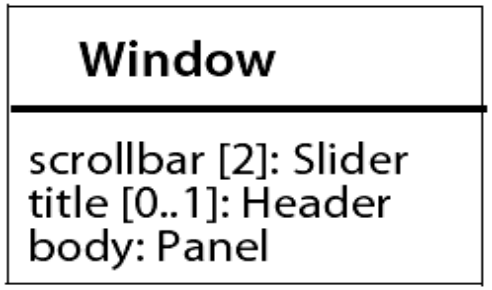


Relatii intre clase

Compunerea



poate fi echivalata cu atribute

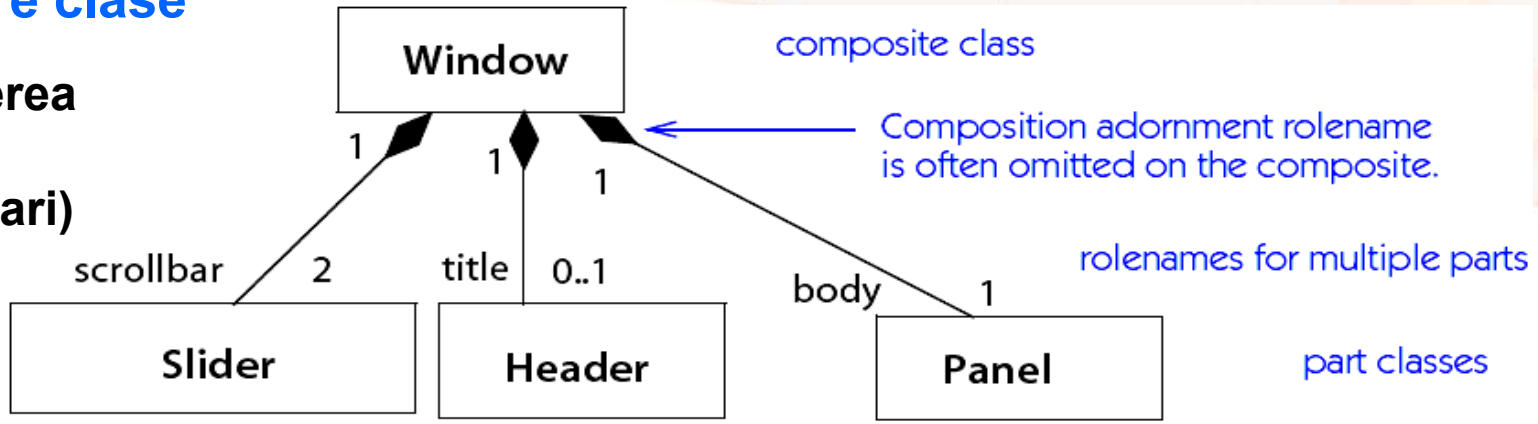


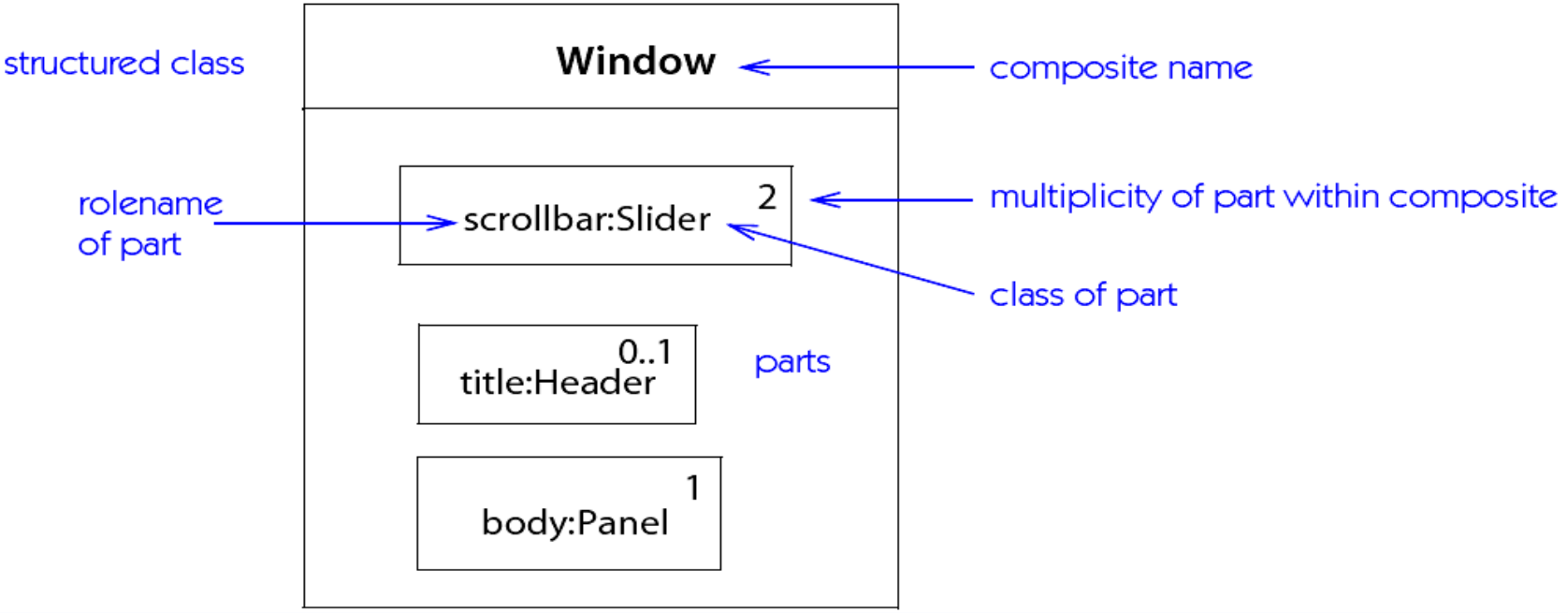
Avoid attributes for objects unless they are unshared and implementation-based.

Attributes are a form of composition

Relatii intre clase

Compunerea (diferite reprezentari)



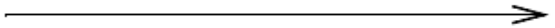
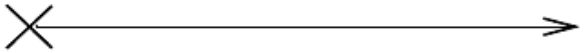
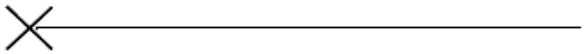
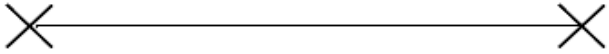


Relatii intre clase

Asocierile descriu **reteaua de relatii structurale care exista intre clase** si care **dau nastere legaturilor intre obiecte**, instante ale acestor clase

Legaturile pot fi vazute ca niste canale de navigatie intre obiecte

Tipuri de navigabilitate

Symbol	Explicit notation	Implicit notation
	unspecified	navigable both ways
	navigable right unspecified left	navigable right only
	navigable right only	navigable right only
	unspecified right nonnavigable left	navigable right only
	navigable both ways	navigable both ways
	nonnavigable both ways	nonnavigable both ways

Relatii intre clase.
Generalizare, specializare si mostenire

Clasa vazuta ca o generalizare (abstractizare) a obiectelor

Clasa

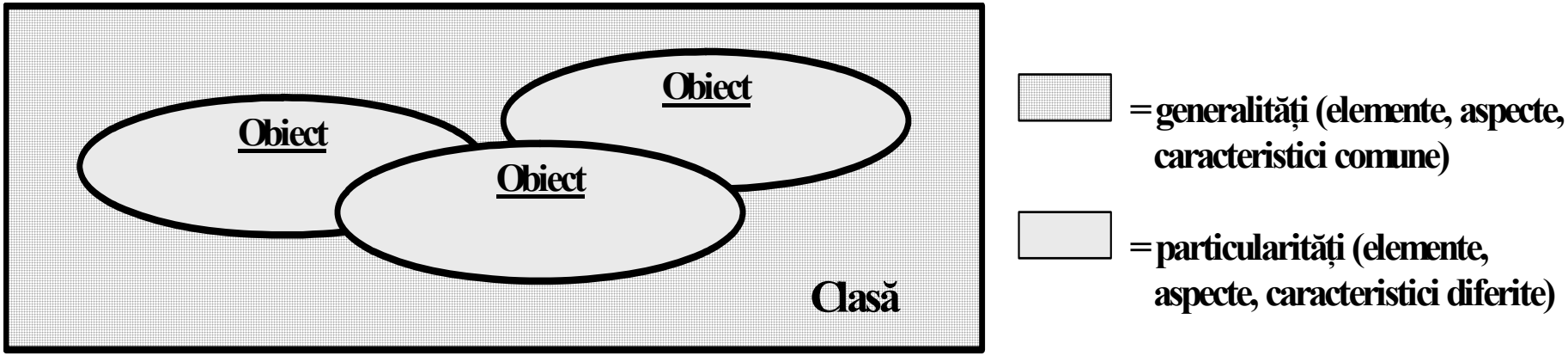
- **contine <u>generalitatile</u>** (abstractiile generale)

Obiectele

- **contin <u>particularitatile</u>** (detaliile particulare)

Clasa vazuta ca

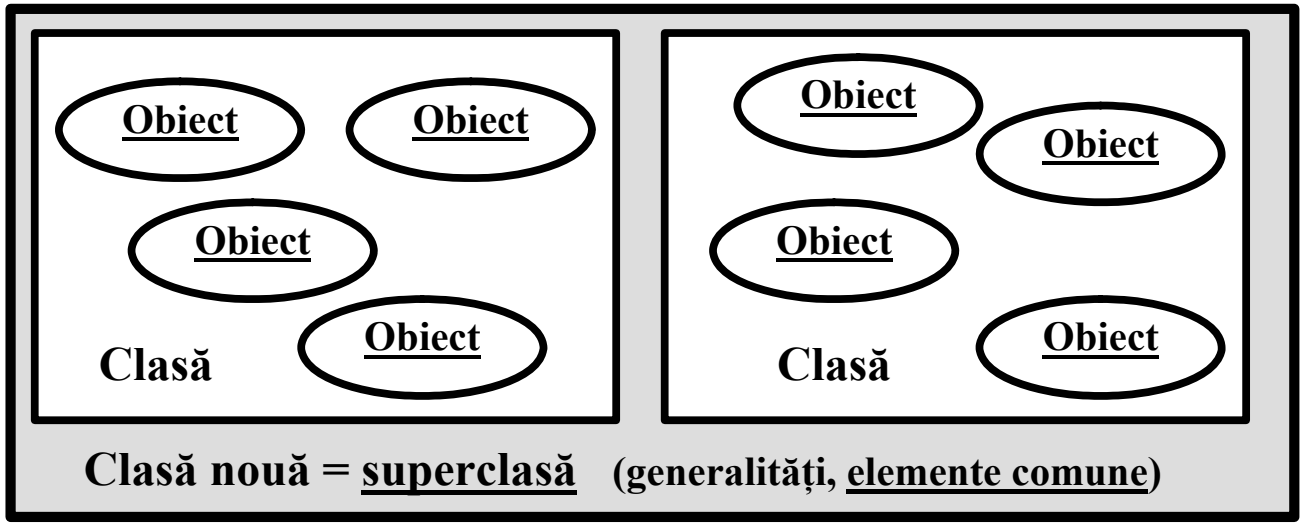
- **multime de obiecte**



Relatii intre clase

Generalizarea

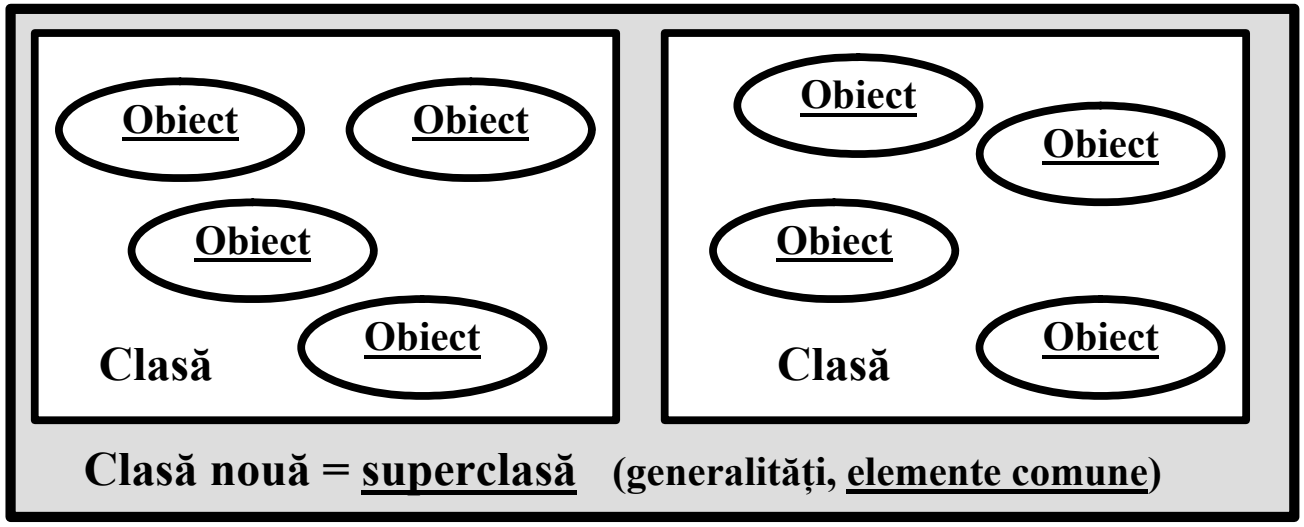
- inseamna **extragerea elementelor comune** (atribute, operații şi constrângeri) **ale unui ansamblu de clase**
  - într-o **nouă clasă** mai **generală**, denumită **superclasă** (parinte)
- superclasa este o **abstracție a subclaselor** (descendentilor) ei
- rezulta o **ierarhie de clase** bazata pe generalizare in care
  - **arborii** de clase sunt **construiți pornind de la frunze**



Relatii intre clase

Generalizarea

- este utilizată cand elementele modelului au fost identificate
  - pentru a **obține o descriere generica a soluțiilor**
- semnifică "**este un (fel de)**"
  - un obiect dintr-o subclasa **este un (fel de)** obiect din superclasa
- privește clasele vazute ca **multimi** (**NU produce legaturi** intre obiecte)
  - **subclasa** este o **submultime** de **obiecte ale superclasei**



Relatii intre clase

Generalizarea actioneaza in OO la **doua niveluri**:

- **clasele** sunt generalizari ale **ansamblurilor de obiecte**
  - un **obiect** este de **felul specificat** de o **clasa**
- **superclasele** sunt generalizari de **clase**
  - **obiectele** de **felul specificat in clasa** sunt si de **felul specificat in superclasa**

Limbajele "orientate spre obiecte" (OO)

- ofera **ambele mecanisme** de generalizare
- de ex. Java, C++, C#

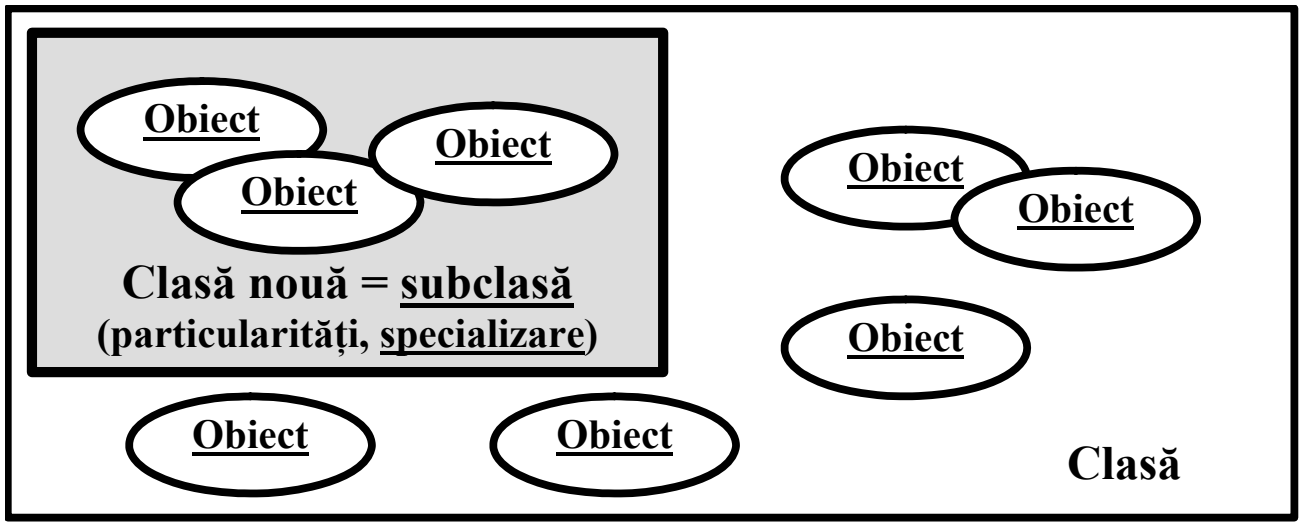
Limbajele care ofera doar constructii numite **obiecte** (eventual **clase**) se pot numi

- **limbaje "care lucreaza cu" obiecte** (si eventual **clase**)

Relatii intre clase

Specializarea

- inseamna **capturarea particularităților / elementelor distincte** ale unui **ansamblu de obiecte ale unei clase existente**
  - noile caracteristici fiind reprezentate **într-o nouă clasă** mai **specializată**, denumită **subclasă**
- este utilă pentru **extinderea coerentă a unui ansamblu de clase**
  - **noile cerințe** fiind încapsulate în **subclase** care **extind coerent** si uniform **funcțiile existente**

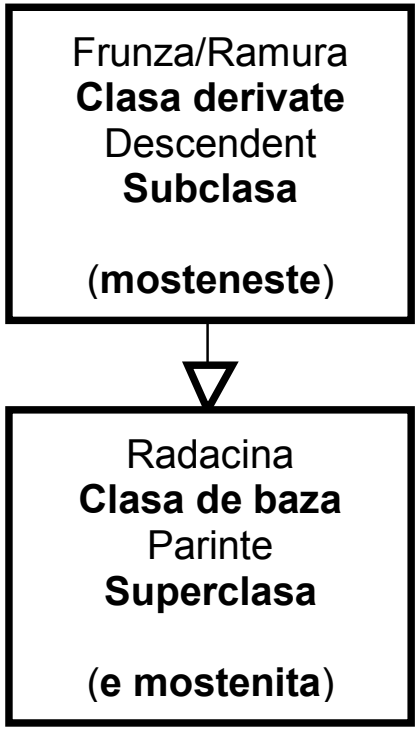


Relatii intre clase

Moştenirea – mecanism suport pentru generalizare/specializare

- **tehnică de generalizare** oferită de limbajele OO pentru a construi o clasă pornind de la una sau mai multe alte clase

- **partajând atributele si operațiile** într-o **ierarhie de clase**



Relatii intre clase

Studiu de caz: orice clasa Java care nu extinde prin mostenire explicit o alta

- **extinde implicit** clasa **Object** (radacina ierarhiei de clase Java)

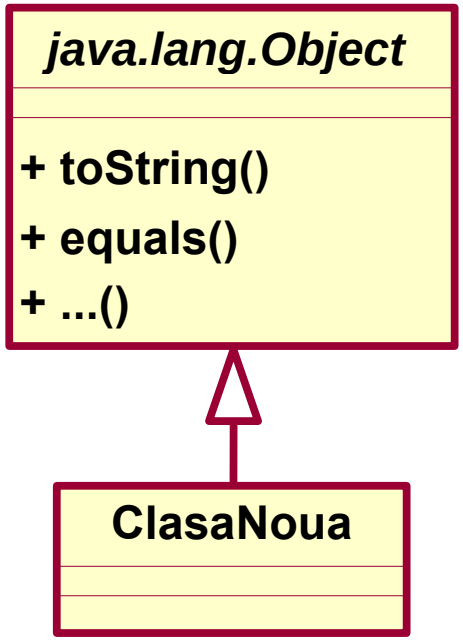
  - care **contine metodele necesare tuturor obiectelor** Java (*equals()*, *toString()*, etc.)

Declaratia Java:

```
class NumeClasa {
    // urmeaza corpul clasei ...
```

este **echivalenta cu**:

```
class NumeClasa extends Object {
    // urmeaza corpul clasei ...
```

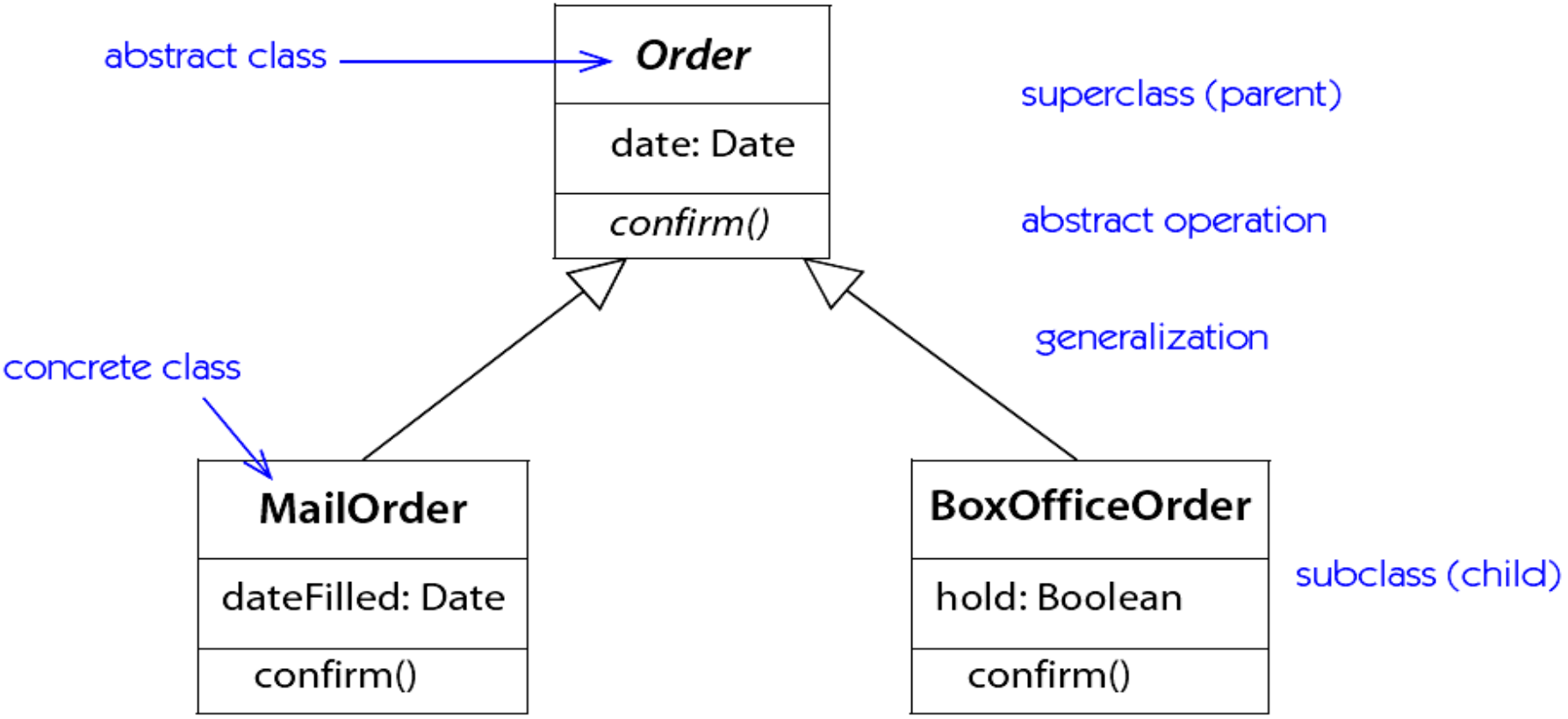


Relatii intre clase.
Clase abstracte. Interfete si implementarea interfetelor

Relatii intre clase

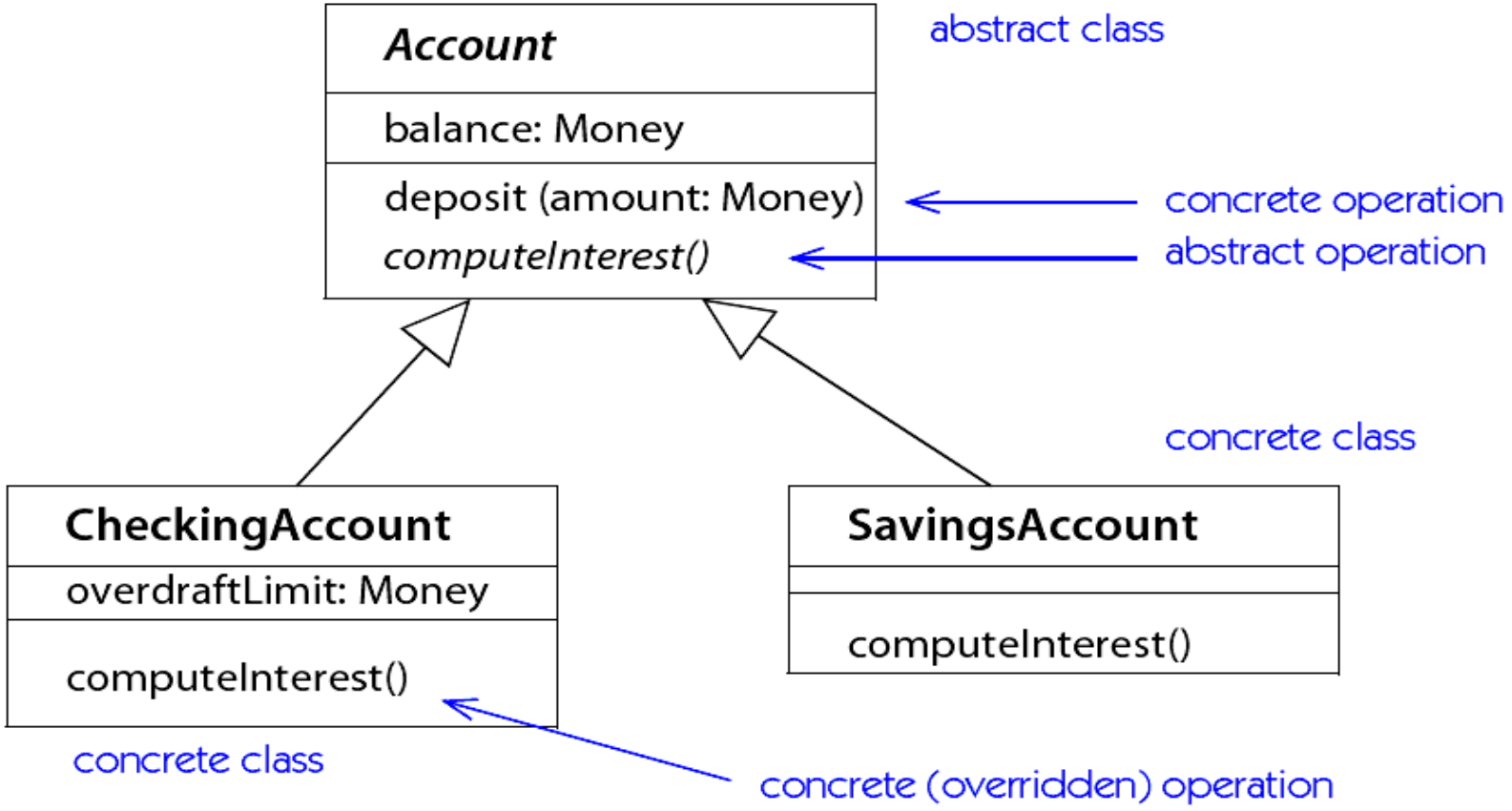
Clase si metode abstracte

- **clasele** numite **abstracte** nu sunt direct instantiabile, adica **nu pot da nastere unor obiecte**, ci pot servi doar ca clase mai generale



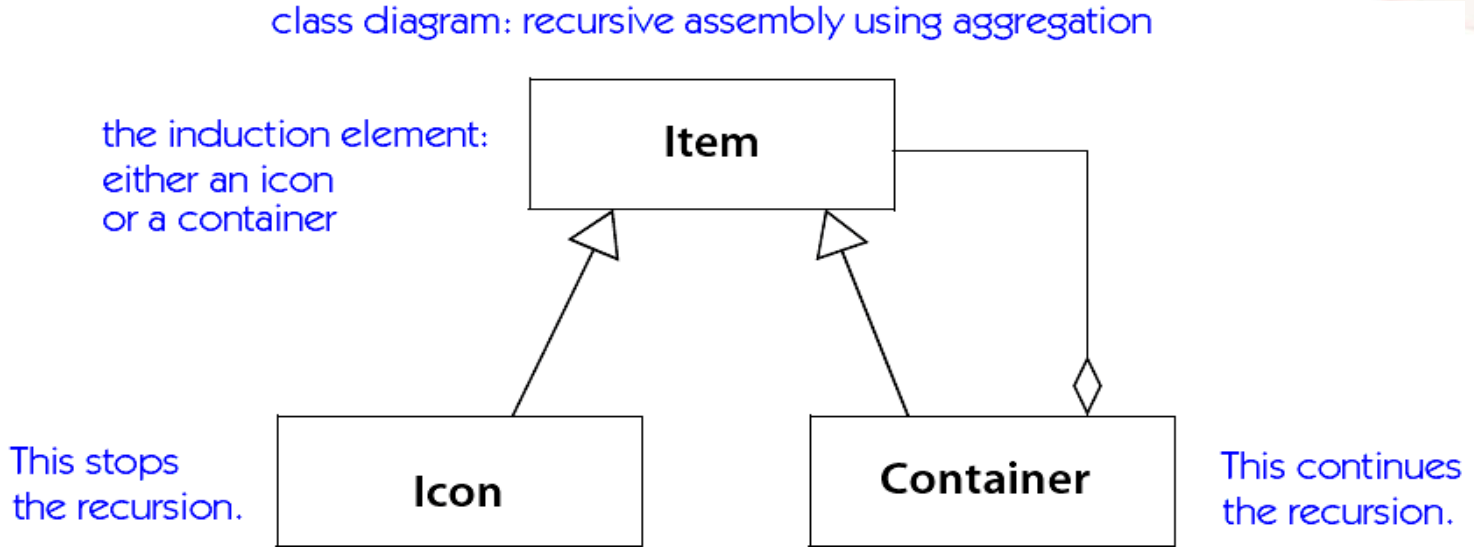
Relatii intre clase

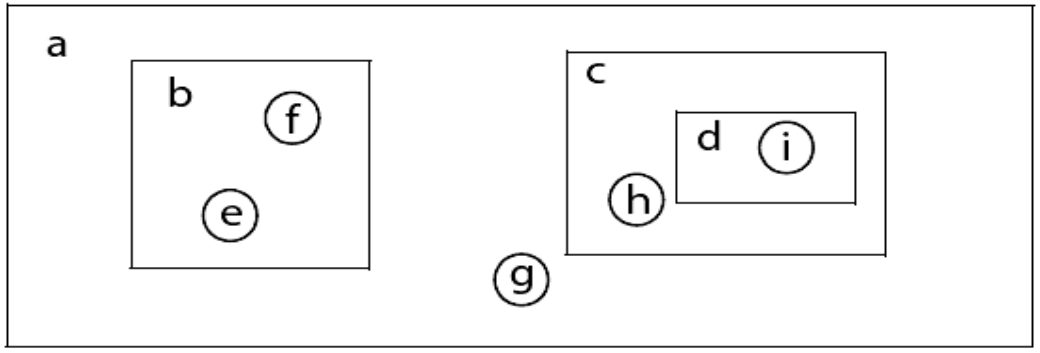
Clase si metode abstracte



Relatii intre clase

<u>Utilizarea generalizarii si a agregarii</u> pentru reprezentarea agregarilor recursive (*pattern-ul **Composite***) **– exemplul containerelor de elemente ale interfetelor grafice**





Schita unei interfete grafice posibil de modelat prin reprezentarea compunerilor recursive

Relatii intre clase

Codul Java echivalent

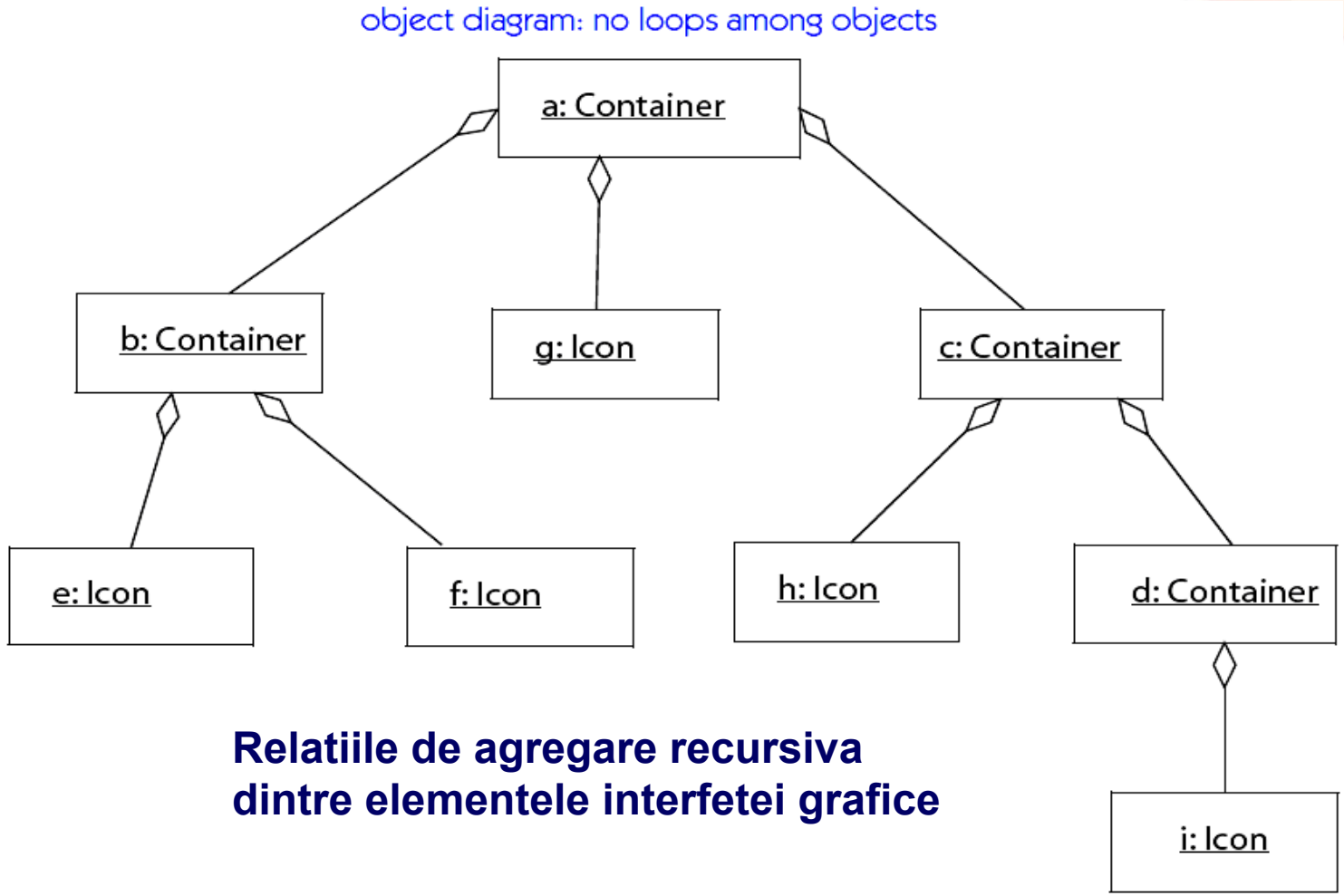
```
1    public abstract class Item {
2       // corpul clasei Item
3    }
```

```
1    public class Container extends Item {
2
3       ArrayList<Item> items;          // parti agregate
4
5       public void add (Item item) {  // legarea partilor
6          items.add (item);
7       }
8    }
```

```
1    public class Icon extends Item {
2       // corpul clasei Icon
3    }
```

Relatii intre clase

Diagrama de obiecte care prezinta agregarile recursive



Relatiile de agregare recursiva dintre elementele interfetei grafice

Relatii intre clase

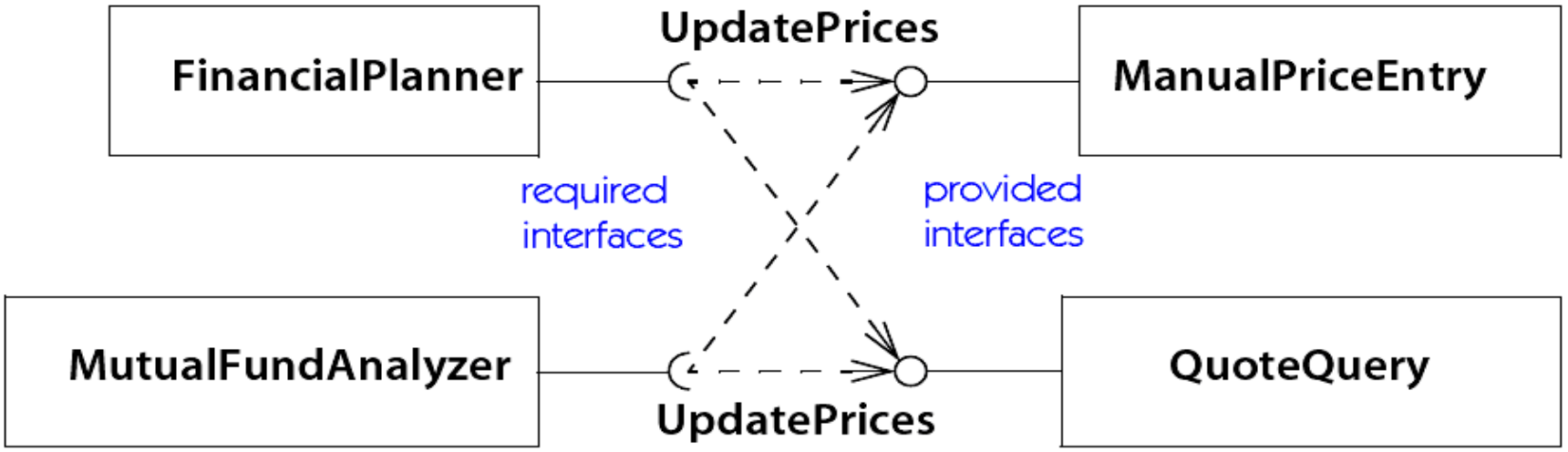
Codul Java echivalent

```
Container a = new Container(); // containerul radacina
Container b = new Container(); // componenta container radacina
Container c = new Container(); // componenta container radacina
Container d = new Container();
Icon e = new Icon();
Icon f = new Icon();
Icon g = new Icon(); // componenta a containerului radacina
Icon h = new Icon();
Icon i = new Icon();
a.add(b); // adaugarea componentelor in containerul radacina
a.add(c); // adaugarea componentelor in containerul radacina
a.add(g); // adaugarea componentelor in containerul radacina
b.add(e);
b.add(f);
c.add(d);
c.add(h);
d.add(i);
```

Relatii intre clase

Interfata

- **tip de date** folosit pentru a **descrie comportamentul vizibil al unei clase**, componente sau al unui pachet, fiind **un stereotip** de tip.
- cea **oferita** e reprezentata ca un **mic cerc legat** printr-un segment **de elementul care ofera serviciile** de interfata
- cea **necesara** e reprezentata ca un **mic semicerc legat** printr-un segment **de elementul care cere serviciile** de interfata



Relatii intre clase

Codul Java echivalent

```
1    public interface UpdatePrices {
2       public void update();  // metoda neimplementata
3    }                         // (semnatura metoda)
```

```
1    public class ManualPriceEntry implements UpdatePrices {
2       public void update() {  // implementare metoda
3       }
4    }
```

```
1    public class QuoteQuery implements UpdatePrices {
2       public void update() {  // implementare metoda
3       }
4    }
```

Relatii intre clase

Codul Java echivalent

```
1    public class FinancialPlanner {
2
3       private UpdatePrices[] updaters = new UpdatePrices[2];
4
5       public void someMethodOfFinancialPlanner() {
6
7          // stocare uniforma
8          updaters[0] = new QuoteQuery();
9          updaters[1] = new ManualPriceEntry();
10
11         // apelare polimorfica uniforma
12         for (int i=0; i<updaters.length; i++) {
13            updaters[i].update();
14         }
15      }
16   }
```

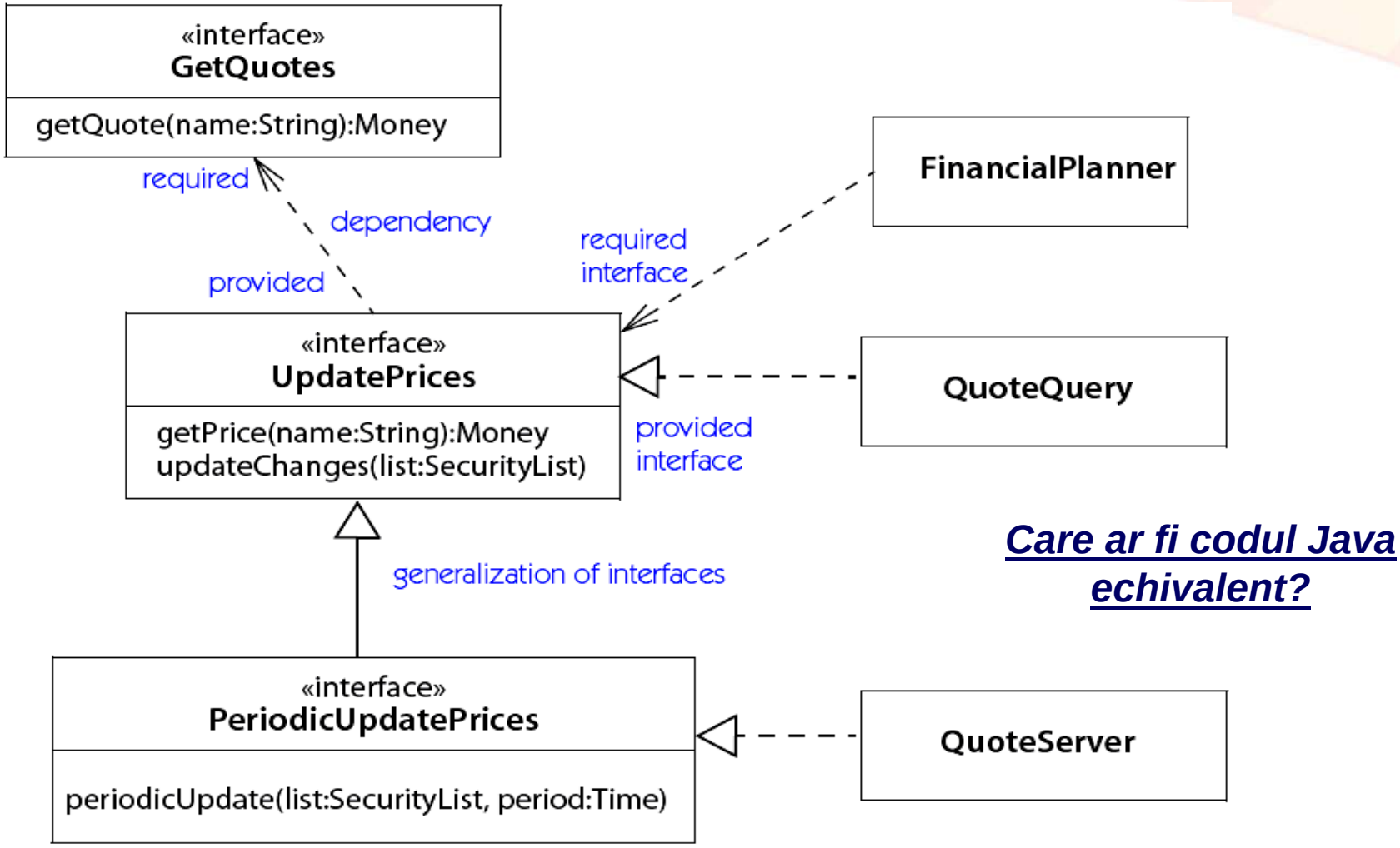
Relatii intre clase

Codul Java echivalent

```
1    public class MutualFundAnalyser {
2
3       private UpdatePrices[] updaters = new UpdatePrices[2];
4
5       public void someMethodOfMutualFundAnalyser() {
6
7          // stocare uniforma
8          updaters[0] = new QuoteQuery();
9          updaters[1] = new ManualPriceEntry();
10
11         // apelare polimorfica uniforma
12         for (int i=0; i<updaters.length; i++) {
13            updaters[i].update();
14         }
15      }
16   }
```

Relatii intre clase

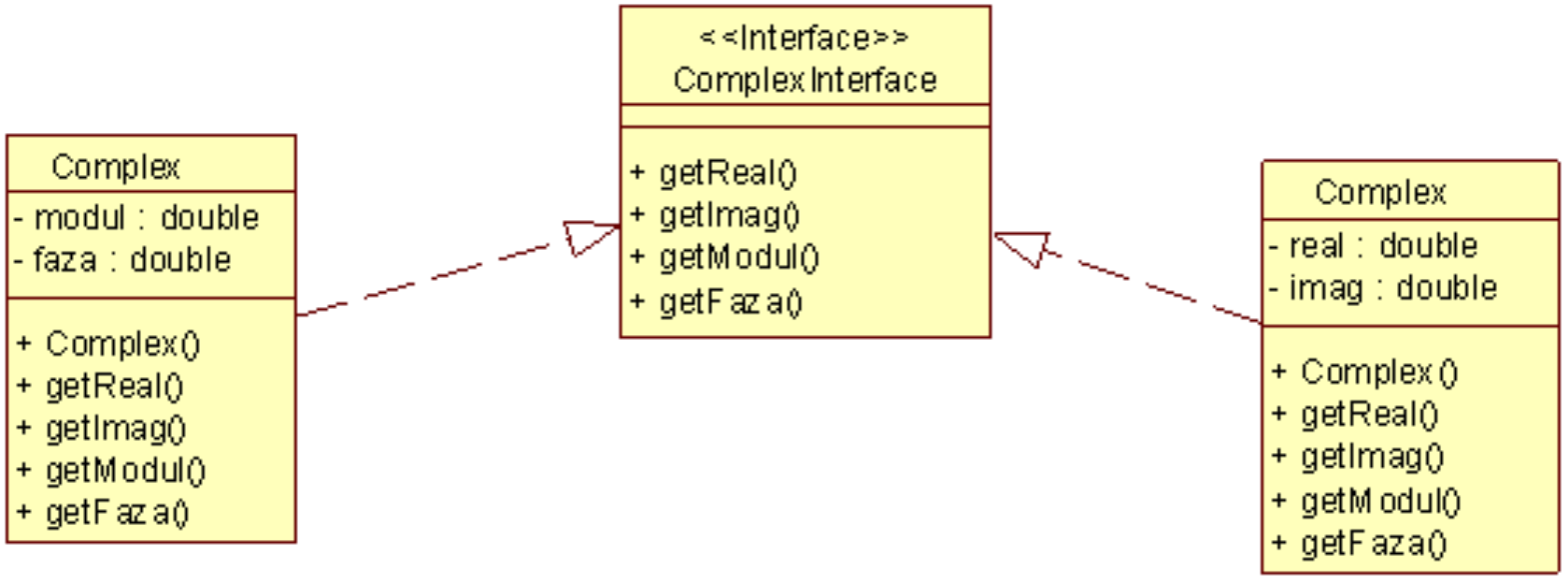
Interfata poate fi reprezentata si ca o **clasa cu** stereotipul **<<interface>>**



Care ar fi codul Java echivalent?

Exemplu de interfata Java (un fel de clasa complet abstracta)

```
// Interfata (colectie de metode neimplementate)
// care reprezinta un contract privind interfata publica
public interface ComplexInterface {

  // Metode abstracte, neimplementate
  public abstract double getReal();
  public abstract double getImag();
  public abstract double getModul();
  public abstract double getFaza();
}
```



Exemplu de clasa care implementeaza o interfata Java

```
// Clasa concreta, din care pot fi create direct obiecte, care
// implementeaza (concretizeaza) interfata ComplexInterface
public class Complex implements ComplexInterface {
  // Atribute private (ascunse, inaccesibile din exteriorul clasei)
  private double real;
  private double imag;
  // Constructori (cu nume supraincarcat) si operatii (metode)
  public void Complex(float real, float imag) {
      this.real = real;
      this.imag = imag;
  }
  public void Complex(double modul, double faza) {
      this.real = modul * Math.cos(faza);
      this.imag = modul * Math.sin(faza);
  }
  public double getReal() {       return this.real;     }
  public double getImag() {       return this.imag;    }
  public double getModul() {
        return Math.sqrt(this.real*this.real + this.imag*this.imag);
  }
  public double getFaza() {
        return Math.atan2(this.real, this.imag);
  }
}
```

Exemplu de clasa care implementeaza o interfata Java

```
// Clasa concreta, din care pot fi create direct obiecte, care
// implementeaza (concretizeaza) interfata ComplexInterface
public class Complex implements ComplexInterface {
  // Atribute private (ascunse, inaccesibile din exteriorul clasei)
  private double modul;
  private double faza;
  // Constructori (cu nume supraincarcat) si operatii (metode)
  public void Complex(float real, float imag) {
      this.modul = Math.sqrt(real*real + imag*imag);
      this.faza = Math.atan2(real, imag);
  }
  public void Complex(double modul, double faza) {
      this.modul = modul;
      this.faza = modul;
  }
  public double getReal() {
        return this.modul*Math.cos(this.faza);
  }
  public double getImag() {
        return this.modul*Math.sin(this.faza);
  }
  public double getModul() {       return this.modul;     }
  public double getFaza() {        return this.faza;     }
}
```