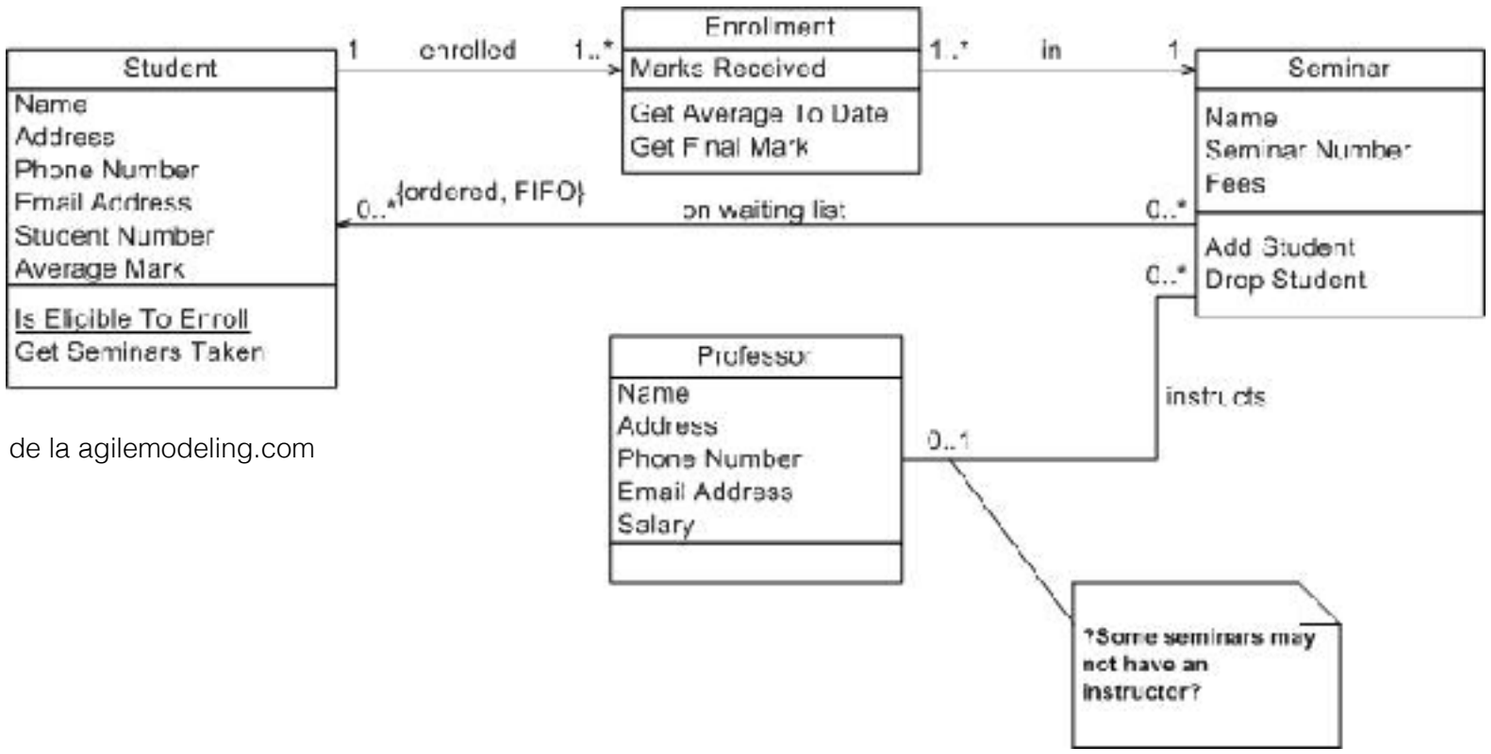
Universitatea din București

Metode de dezvoltare software

Diagrame UML de clase

06.03.2017

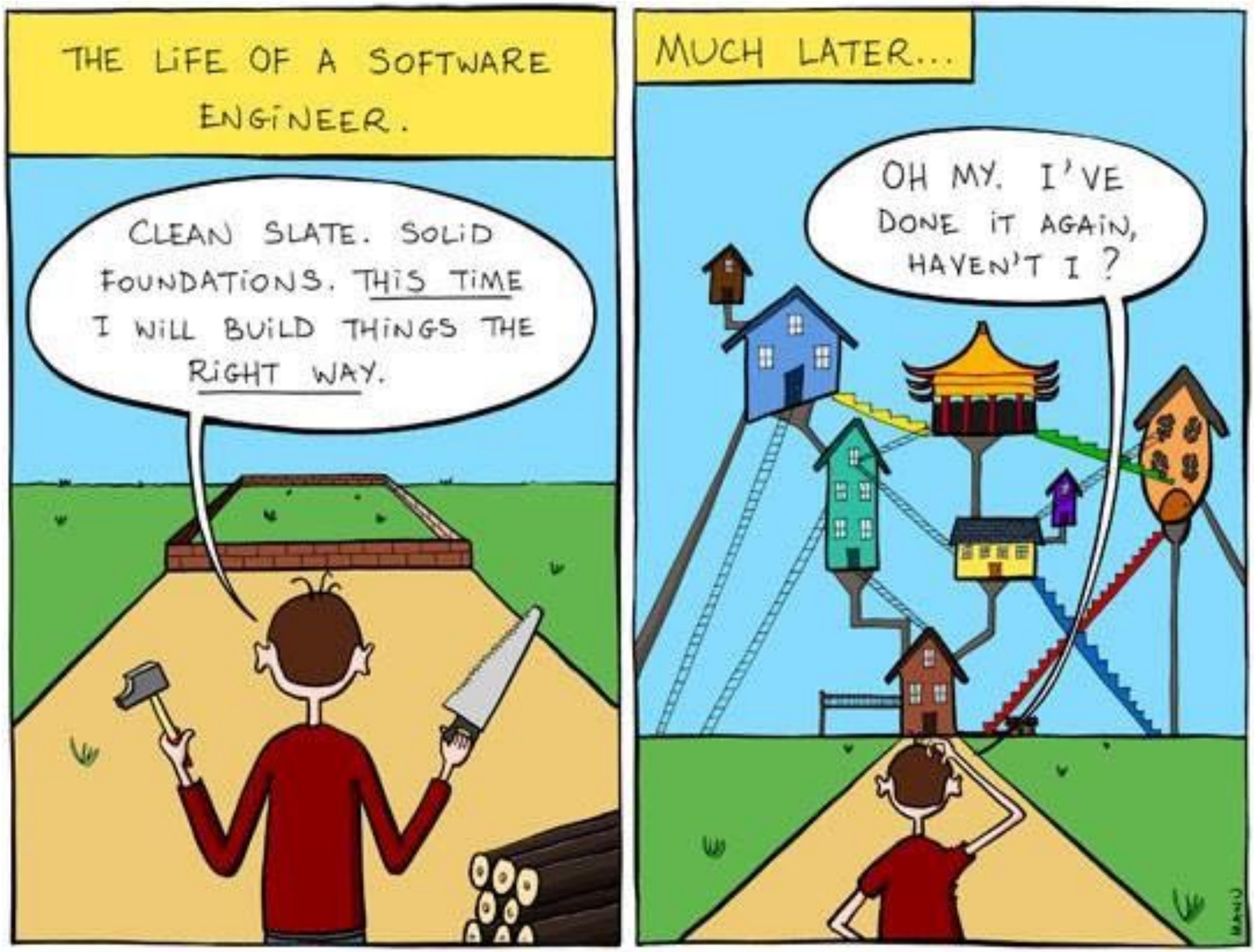
Alin Ștefănescu



de la agilemodeling.com

Diagrame de clase

Diagrame de clase… în practică



de la lolzland.com

Introducere la clase

- Modelarea unui sistem presupune identificarea lucrurilor care sunt importante pentru acesta şi care formează vocabularul lui. În UML, toate aceste lucruri sunt modelate folosind noțiunea de **clasă**.

- O clasă înseamnă descrierea unei mulţimi de **obiecte** care au în comun aceleaşi atribute, operaţii, relaţii şi semnificaţii.

- **Diagramele de clase** sunt folosite pentru a specifica structura statică a sistemului, adică:
  - ce clase există în sistem şi
  - care este legătura dintre ele.

Recapitulare paradigma “orientare pe obiect”

- **Obiect**: structură software care grupează stări și comportament, folosind conceptul de încapsulare
- **Clasă**: machetă folosită pentru crearea de obiecte.
- **Moștenire**: mecanism natural de organizare ierarhică a claselor, reutilizând elemente comune.
- **Interfață**: un contract între o clasă și lumea exterioară. Când o clasă implementează o interfață, promite să ofere comportamentul publicat de interfață.
- **Pachet**: organizare a claselor și interfețelor folosind un spațiu de nume (*namespace*). Folosirea pachetelor permite o structurare mai bună a proiectelor mari.

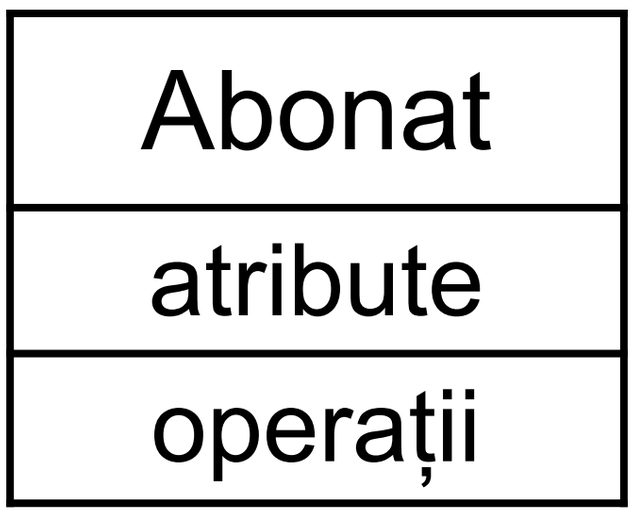
```
class Bicycle {
    int speed = 0;
    int gear = 1;
    void changeGear(int newValue) {
         gear = newValue;
    }
    void speedUp(int increment) {
         speed = speed + increment;
    }
    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }
}
```

```
class MountainBike extends Bicycle {
    // new fields and methods defining
    // a mountain bike go here
}

interface Bicycle {
    void changeGear(int newValue);
    void speedUp(int increment);
    void applyBrakes(int decrement);
}
class MyBicycle implements Bicycle {
    // remainder of this class
    // implemented as before
}

package ro.unibuc.fmi.bikes;

import ro.unibuc.fmi.bikes.*;
```

Reprezentarea grafică a unei clase

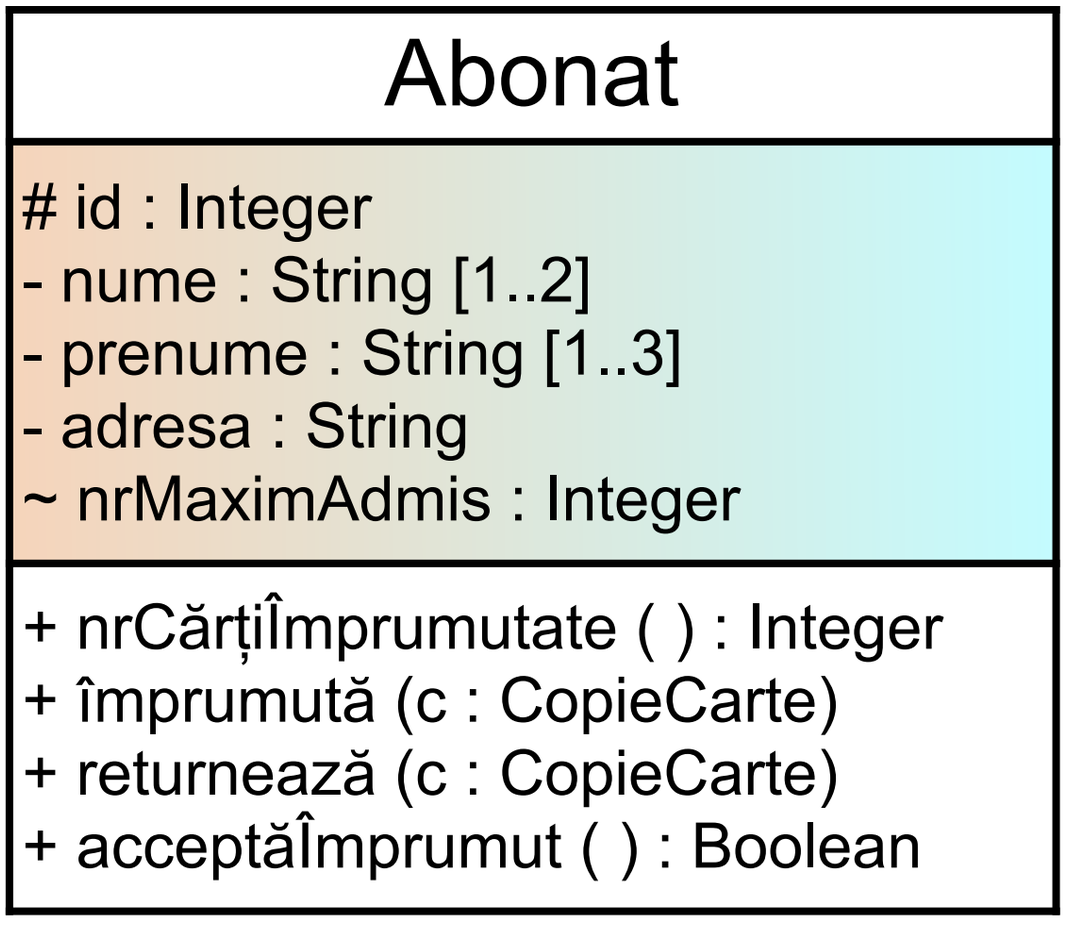
- În UML, o clasă este prezentată printr-un dreptunghi în interiorul căruia se scrie numele acesteia:

- Fiecare clasă este caracterizată printr-o mulţime de **atribute** şi **operaţii**:



Atribute

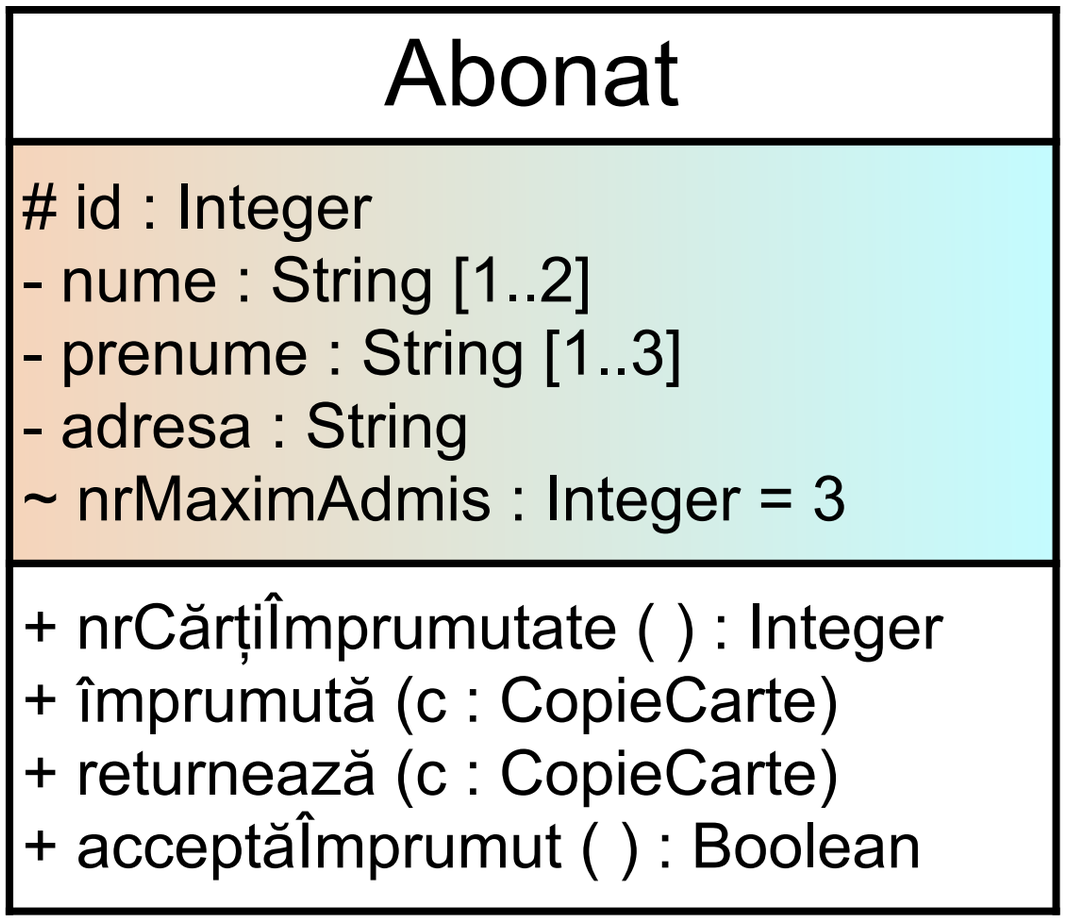
- Un **atribut** reprezintă o proprietate a unei clase.
- Atributele descriu datele conţinute de obiectele din clasa respectivă.
- Pentru fiecare atribut trebuie specificat tipul acestuia. **Tipurile** folosite pot fi tipuri de bază sau clase.
- Pentru fiecare atribut pot fi specificate vizibilitatea, multiplicitatea și valoarea inițială.



Vizibilitatea atributelor

Dpdv al **vizibilității**, atributele pot fi:

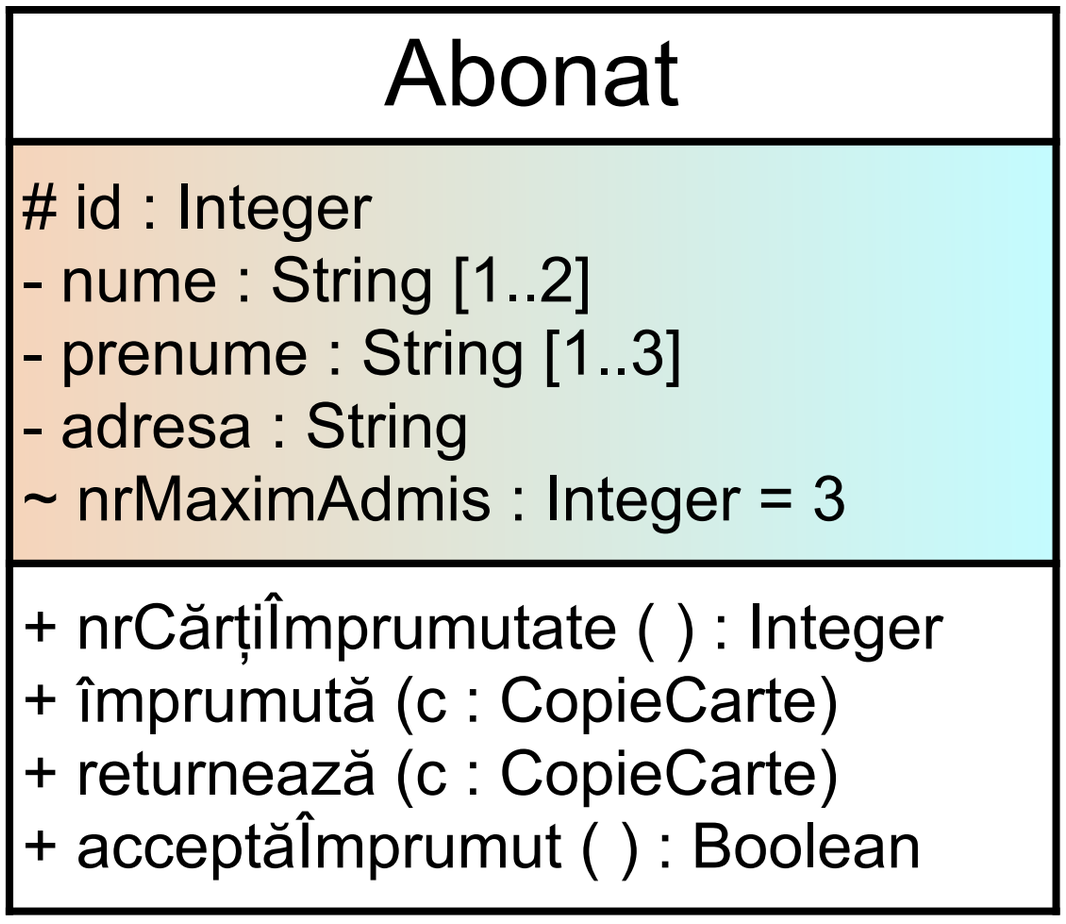
- **publice '+'**: pot fi accesate de orice altă clasă
- **private '-'**: nu pot fi accesate de alte clase
- **protejate '#'**: pot fi accesate doar de subclasele care descind din clasa respectivă
- **package '~'**: pot fi accesate doar de clasele din același "package"



Multiplicitatea atributelor

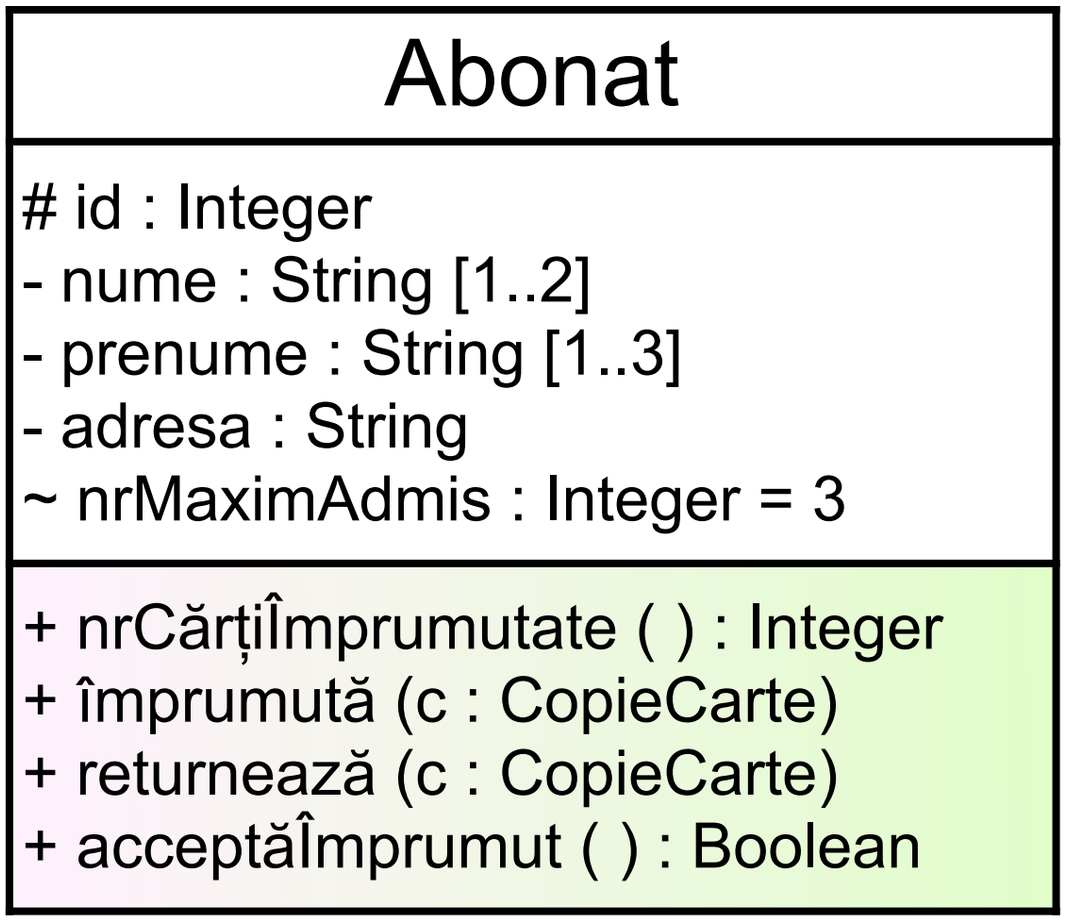
- Atributele pot avea diverse **multiplicități** (default e 1).
  De exemplu: 5, *, 1..3, 3..*, etc.

- Atributele pot avea o **valoare inițială**. Vezi '*nrMaximAdmis*' din clasa alăturată.



Operații

- **Operaţiile** clasei definesc modurile în care interacţionează obiectele.
- Când un obiect trimite un mesaj către un alt obiect, îi cere acestuia din urmă să execute o *operaţie*.
- Obiectul care primeşte mesajul va apela o *metodă* pentru a executa această operaţie.

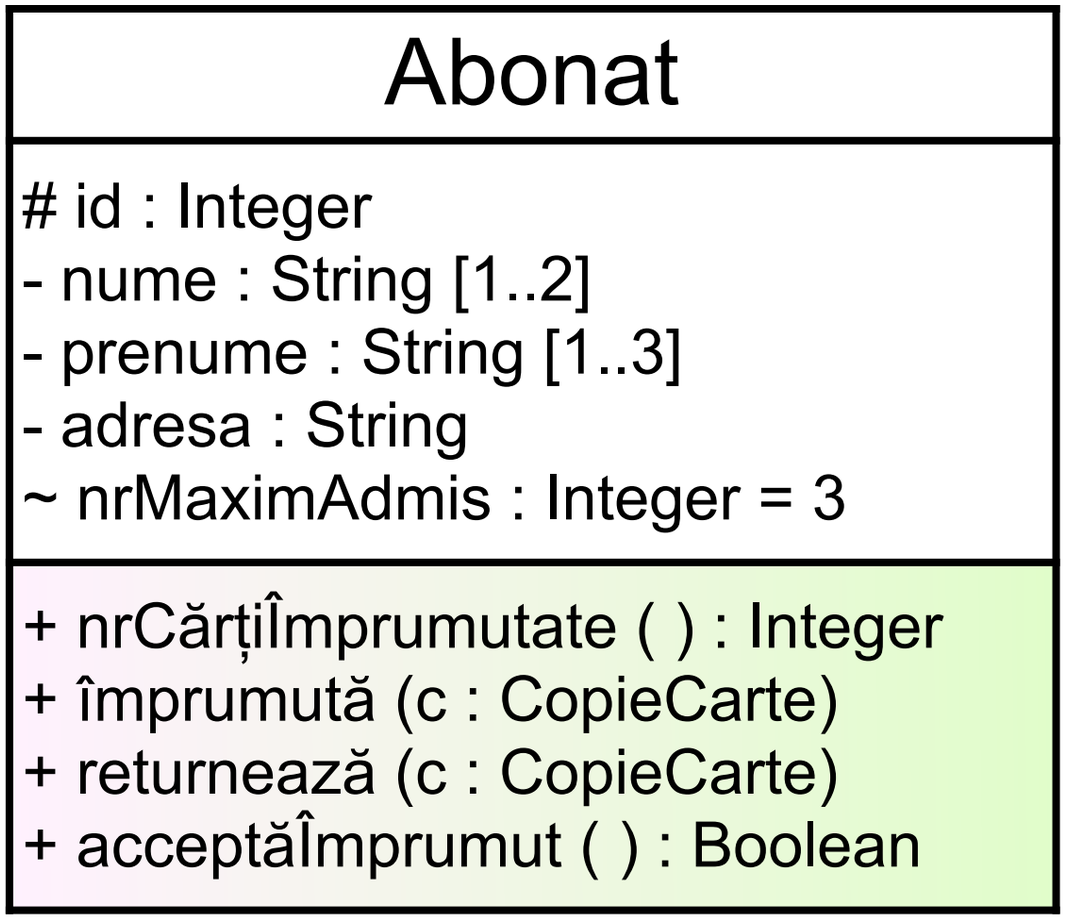


Operații

- **Semnătura** unei operaţii este formată din:
  - numele operației,
  - numele şi tipurile **parametrilor** (dacă e cazul) şi
  - tipul care trebuie returnat (dacă este cazul).

- La fel ca și atributele, operațiile pot fi publice, private, protejate, sau corespunzătoare unui pachet.

- Operații specifice: constructori, accesori (getX) și mutatori (setX).

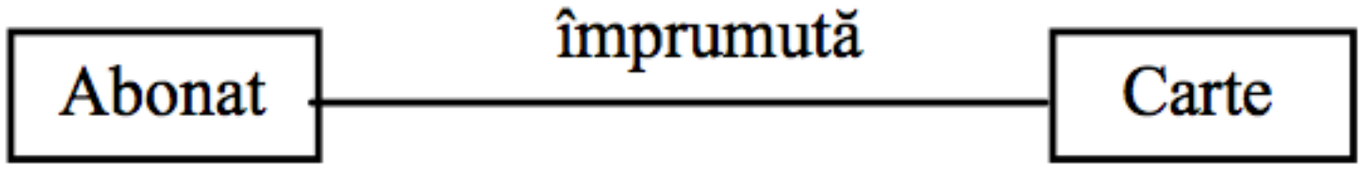


Relații între clase

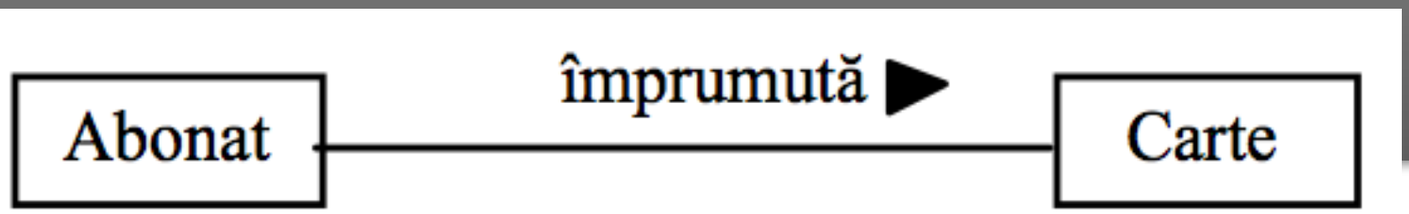
- Clasele formează relații
- Tipuri de relații între clase:
  - asociere
  - generalizare
  - dependență
  - realizare
- Pe acestea le vom studia în continuare

Asocieri

- **Asocierile** sunt legături structurale între clase.
- Între două clase există o asociere atunci când un obiect dintr-o clasă interacționează cu un obiect din cealaltă clasă.
- După cum clasele erau reprezentate prin substantive, asocierile sunt reprezentate prin verbe.
- Pentru a indica direcţia de citire a numelui asocierii (d. ex. de la Abonat la Carte) se poate folosi un triunghi negru.



Asocieri

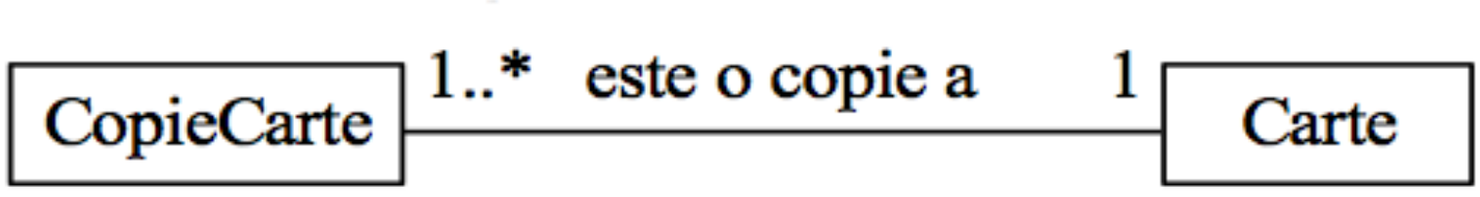


- În general, clasa A este asociată cu clasa B, dacă un obiect din clasa A trebuie să aibă cunoștinţă de un obiect din clasa B.
- În particular, putem identifica următoarele cazuri:
  - un obiect din clasa A are un atribut ale cărui valori sunt obiecte sau colecţii de obiecte din clasa B
  - un obiect din clasa A creează un obiect din clasa B
  - un obiect din clasa A trimite un mesaj către un obiect din clasa B

Multiplicități

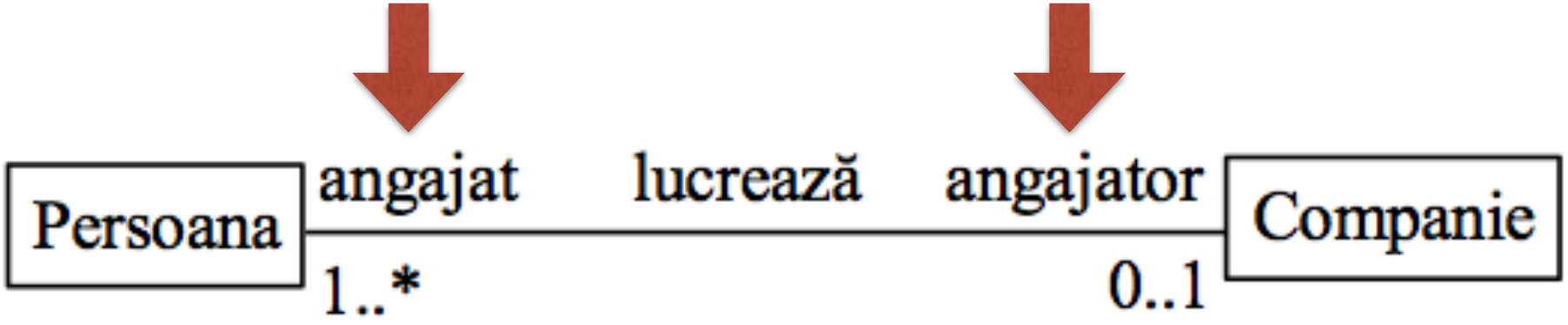
- Participarea unei clase la o asociere este caracterizată de o anumită ***multiplicitate***. De exemplu, o carte poate avea una sau mai multe copii, în timp ce o copie de carte aparţine doar unei singure cărți.

- În UML, multiplicităţile pot fi specificate în modul următor:
  - un număr, de exemplu 1
  - un interval de numere, de exemplu 2..5
  - un număr arbitrar, folosind simbolul *.



Roluri

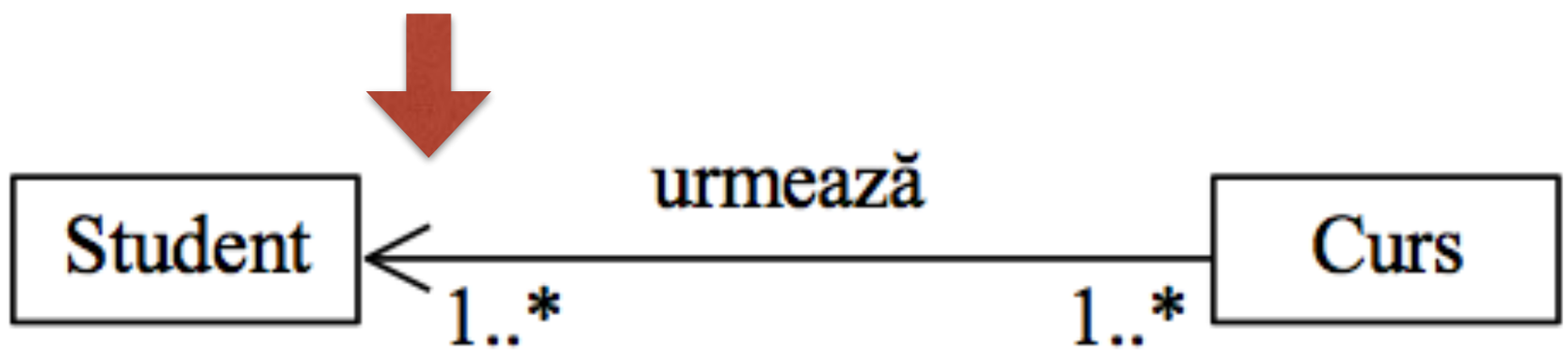
- Uneori asocierea este mai uşor de înţeles dacă rolurilor pe care obiectele le au în cadrul asocierii li se atribuie **nume** separate.



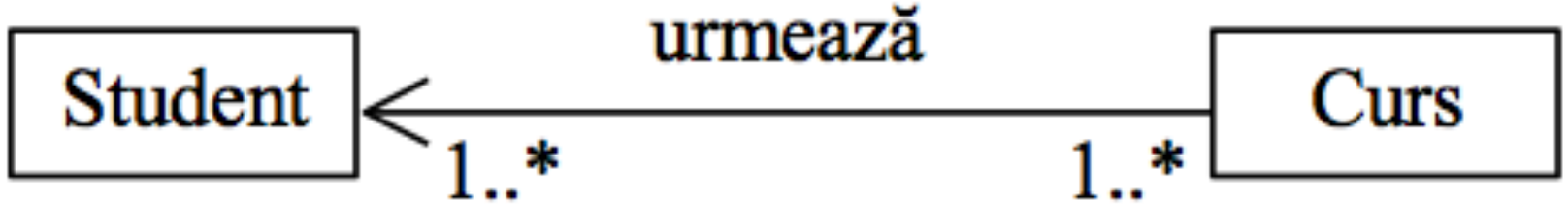
Navigabilitate

- În UML putem plasa o săgeată la unul dintre capetele liniei ce reprezintă asocierea pentru a arăta că este posibil să se trimită mesaje în direcţia săgeţii.

- În exemplul de mai jos, spunem că obiectul *Curs* va avea cunoştinţă de obiectul *Student*, dar nu şi invers.

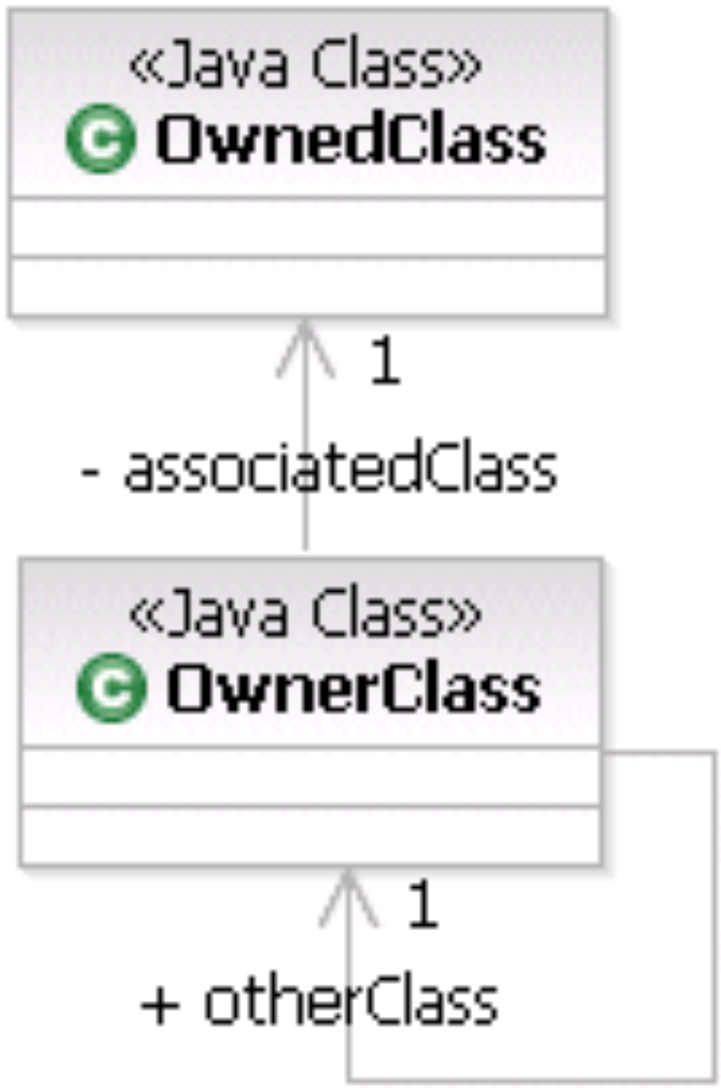


Navigabilitate



- Specificarea unei direcții de navigație nu înseamnă neapărat ca nu este posibilă o navigație în sens invers, ci ca acest lucru este mai greu de realizat.
- Atunci când se specifică sensul de navigare de la clasa A la clasa B, există o modalitate precisă și directă de acces de la un obiect din clasa A la un obiect din clasa B (A stochează o referință la B).
- În exemplul (*Student* ⟵ *Curs*), asocierea poate fi implementată printr-un atribut care să reprezinte mulţimea de obiecte din clasa *Student* care participă la *Curs*.
- **Dacă nu există nici o săgeată** (navigabilitatea nu este specificată explicit), atunci se consideră ca asocierea este **bidirecțională**.
- În general, navigabilitatea este specificată explicit doar când este cu adevărat importantă pentru aplicație.

Navigabilitate - exemplu în Java



OwnedClass.Java:

```
public class OwnedClass {

        // <<class body>>
}
```

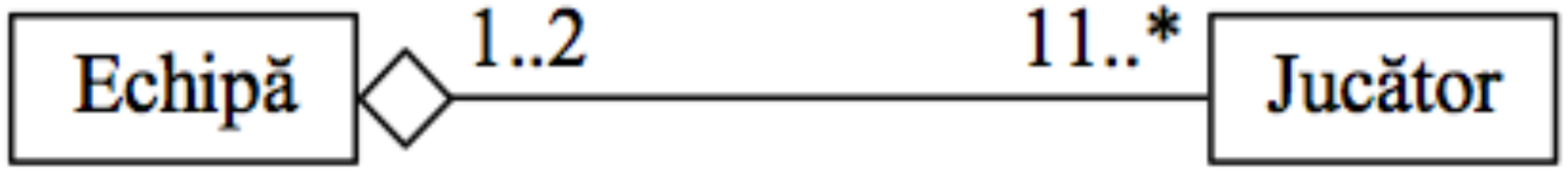
OwnerClass.Java:

```
public class OwnerClass {
      private OwnedClass associatedClass;
      public OwnerClass otherClass;

        // <<class body>>
}
```

Observație: o asociație poate să refere o singură clasă

Agregare și compunere

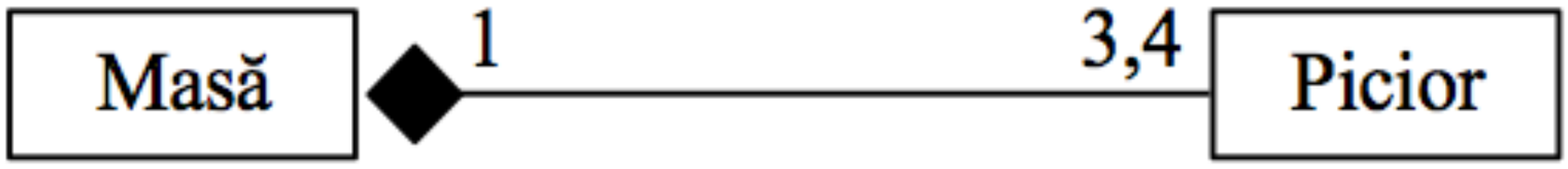
- **Agregarea** şi **compunerea** reprezintă tipuri de asociere în care un obiect dintr-o clasă **face parte** dintr-un obiect din altă clasă.
- **Agregarea** este modul cel mai general de a indica în UML o relaţie de tip parte-întreg.



- Diferența dintre o simplă asociere și agregare este pur **conceptuală**: folosirea agregării indică faptul că o clasă reprezintă un lucru "mai mare" (întregul), care conține mai multe lucruri "mai mici" (părțile).

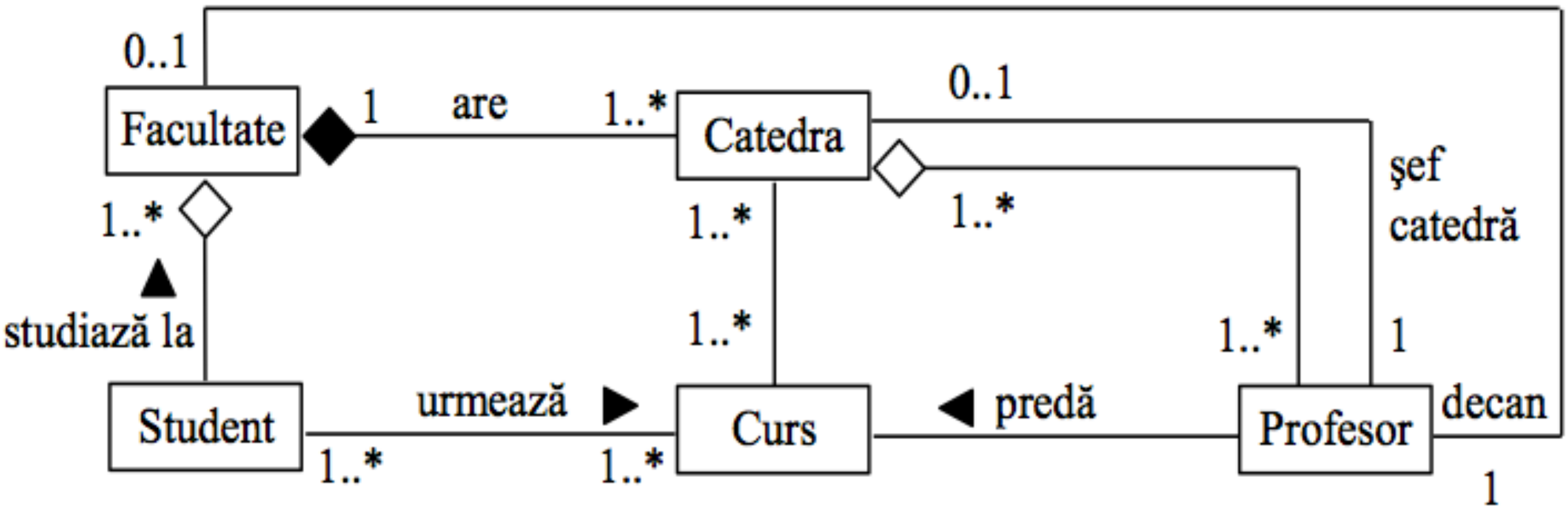
Compunerea

- Există însă un **caz special de agregare**, **compunerea**, în care relația dintre întreg şi părţile sale este mai puternică.



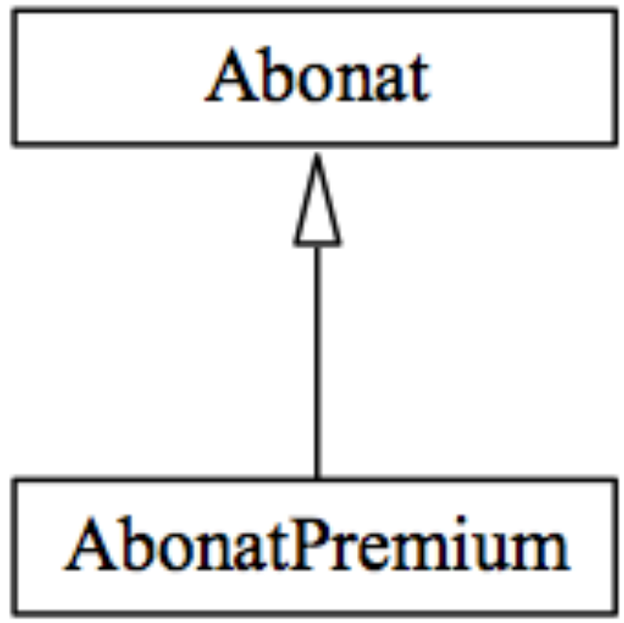
- Compunerea implică o apartenență puternică a părții la întreg și o coincidență între durata de viață a părții și a întregului: dacă întregul este creat, mutat sau distrus, atunci şi părţile componente sunt create, mutate sau distruse.
- În cazul compunerii, o parte nu poate să fie conţinută în mai mult de un singur întreg, astfel încât multiplicitatea asocierii la extremitatea întregului trebuie să fie 1 sau 0..1.

Exemplu


0..1
Facultate
1
are
1..*
Catedra
0..1
şef
catedră
1..*
1..*
1..*
studiază la
1..*
1
Student
urmează
Curs
predă
Profesor
decan
1..*
1..*
1

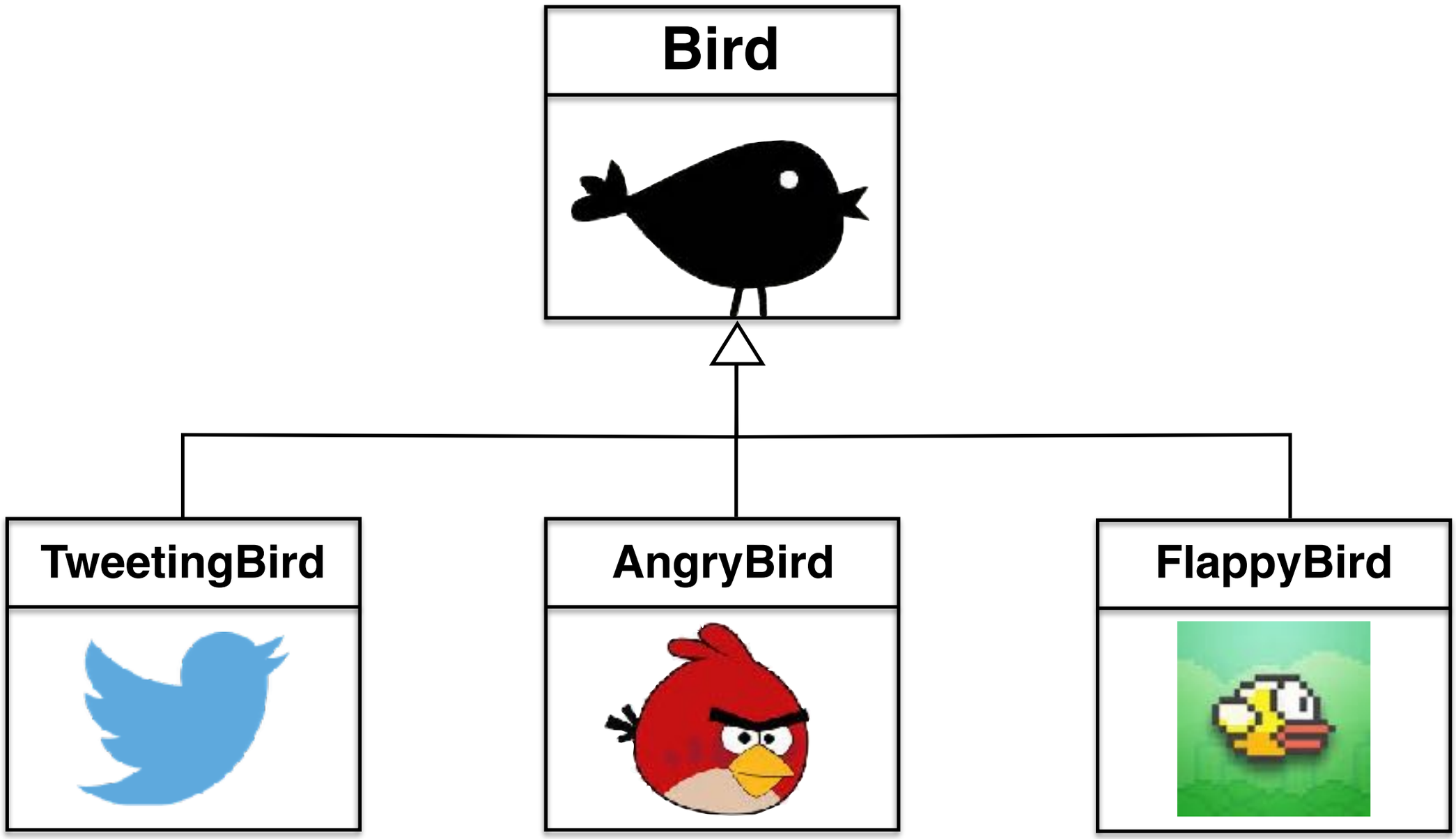
Generalizare (generalization) și moștenire (inheritance)

- În paradigma programării orientate pe obiect, o subclasă poate **moșteni** structura și comportamentul unei superclase (adică toate atributele, operațiile și relațiile care există în superclasă vor exista și în subclasă)
- În UML se vorbește despre relația inversă, de **generalizare**



- **Generalizare**: relație între un lucru general (numit superclasă sau părinte, ex. *Abonat*) și un lucru specializat (numit subclasă sau copil, ex. *AbonatPremium*)
- Un obiect al unei clase mai generale poate fi substituit cu un obiect al unei clase mai specializate în orice context, dar nu şi invers.

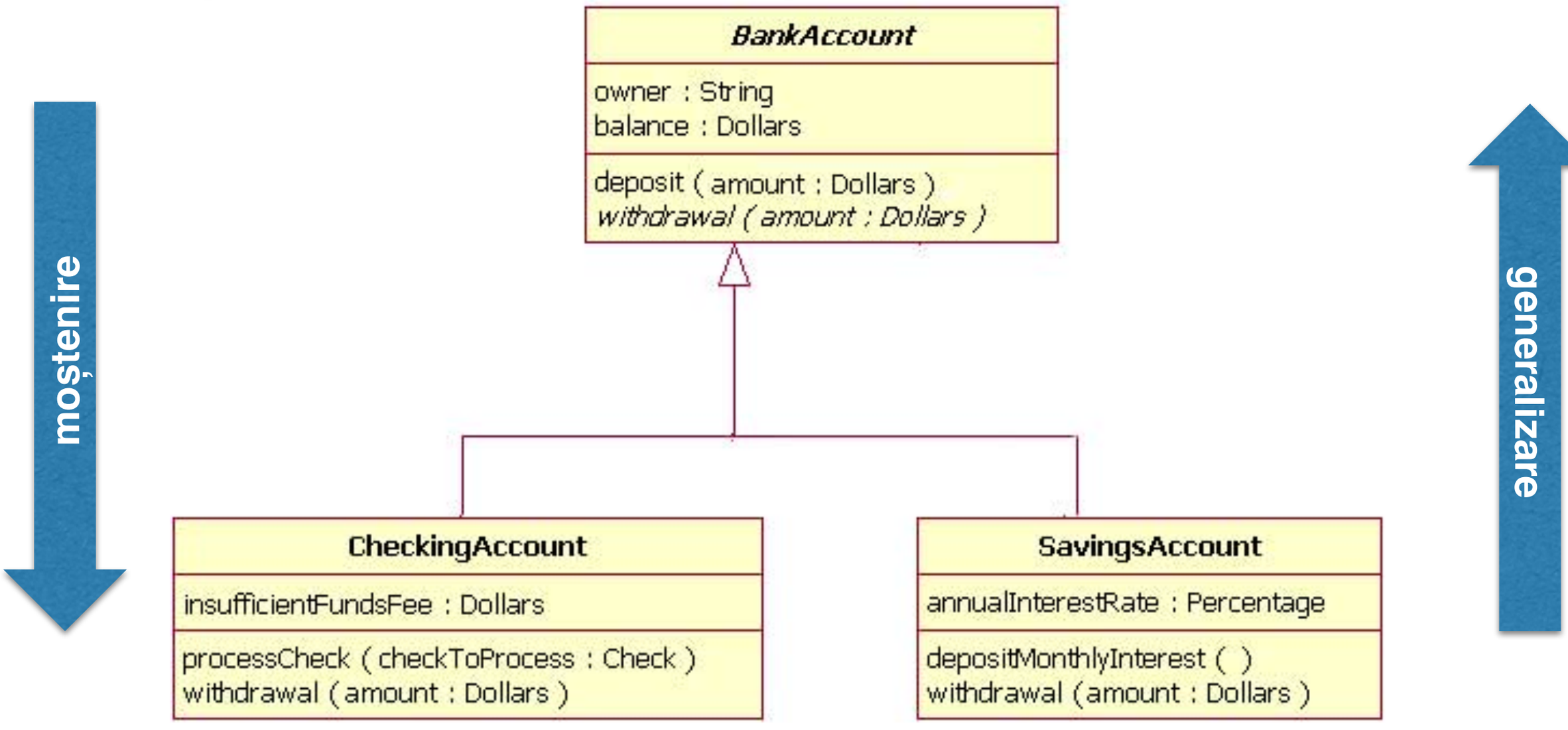
Generalizarea… în practică



Alin Ștefănescu

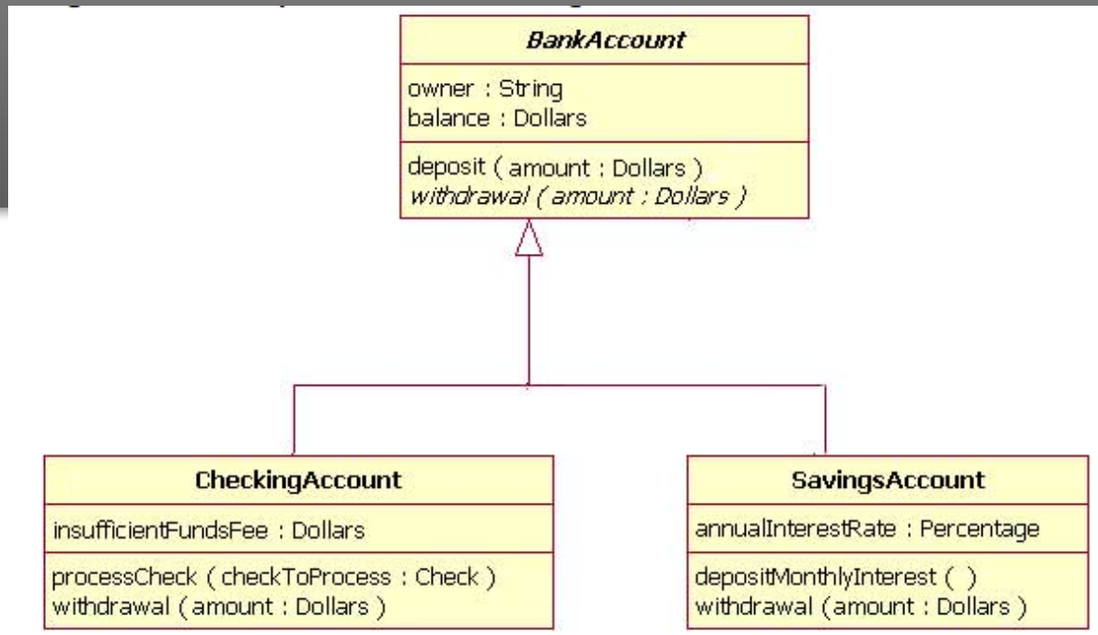
Moștenirea (inheritance)

Q: Cum poti deveni bogat în mod orientat pe obiect?
A: Printr-o “moștenire”



de la http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/index.html

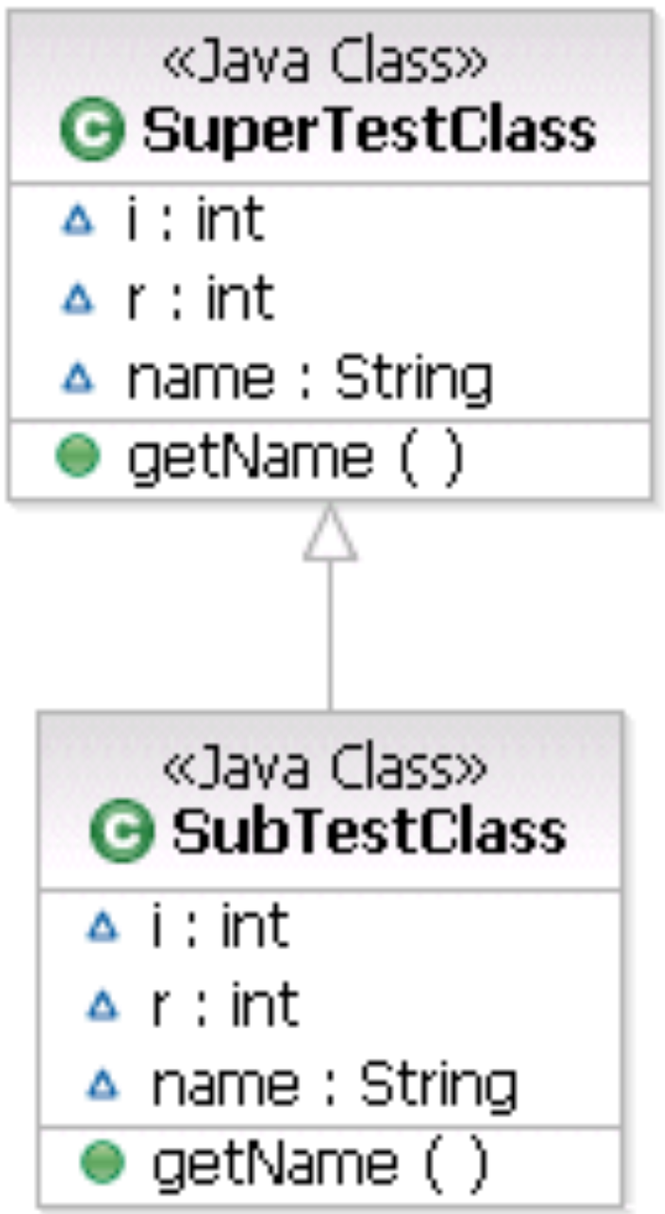
Moștenirea



- Pe de altă parte, copilul poate **adăuga** structură și comportament nou, adică poate avea atribute, operații și relații noi față de cele ale superclasei.

- În plus, copilul poate chiar **schimba** comportamentul părintelui. Acest lucru se întâmplă atunci când o operație a subclasei care are aceeași semnătură ca și o operație a superclasei **suprascrie** acea operație. Acest lucru poartă numele de **polimorfism**.

Generalizare - exemplu în Java



SuperTestClass.java:

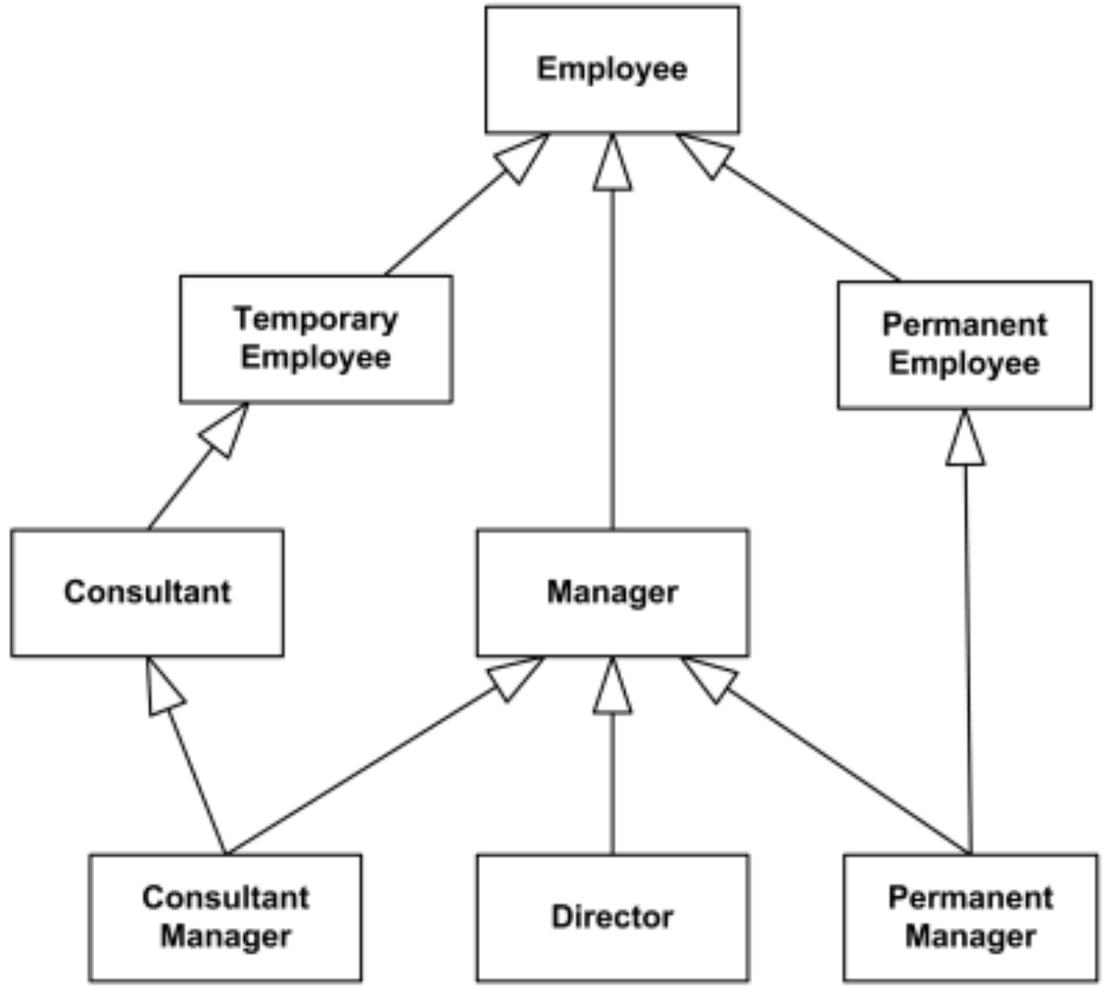
```
public class SuperTestClass {
	int i = 3;
	int r = 5;
	String name ="myName";

	public void getName(){
	};

}
```

SubTestClass.java:

```
public class SubTestClasss
	 extends SuperTestClass {
	int i = 2;
	int r = 3;
	String name ="myName";

	public void getName(){
	};

}
```

Moștenire multiplă

- Atunci când o clasă are un singur părinte, spunem că folosește **moștenire simplă**; în caz contrar, moștenirea se numește **multiplă**.
- În majoritatea cazurilor, moștenirea simplă este suficientă, dar există însă și cazuri când moștenirea multiplă este mai eficientă.

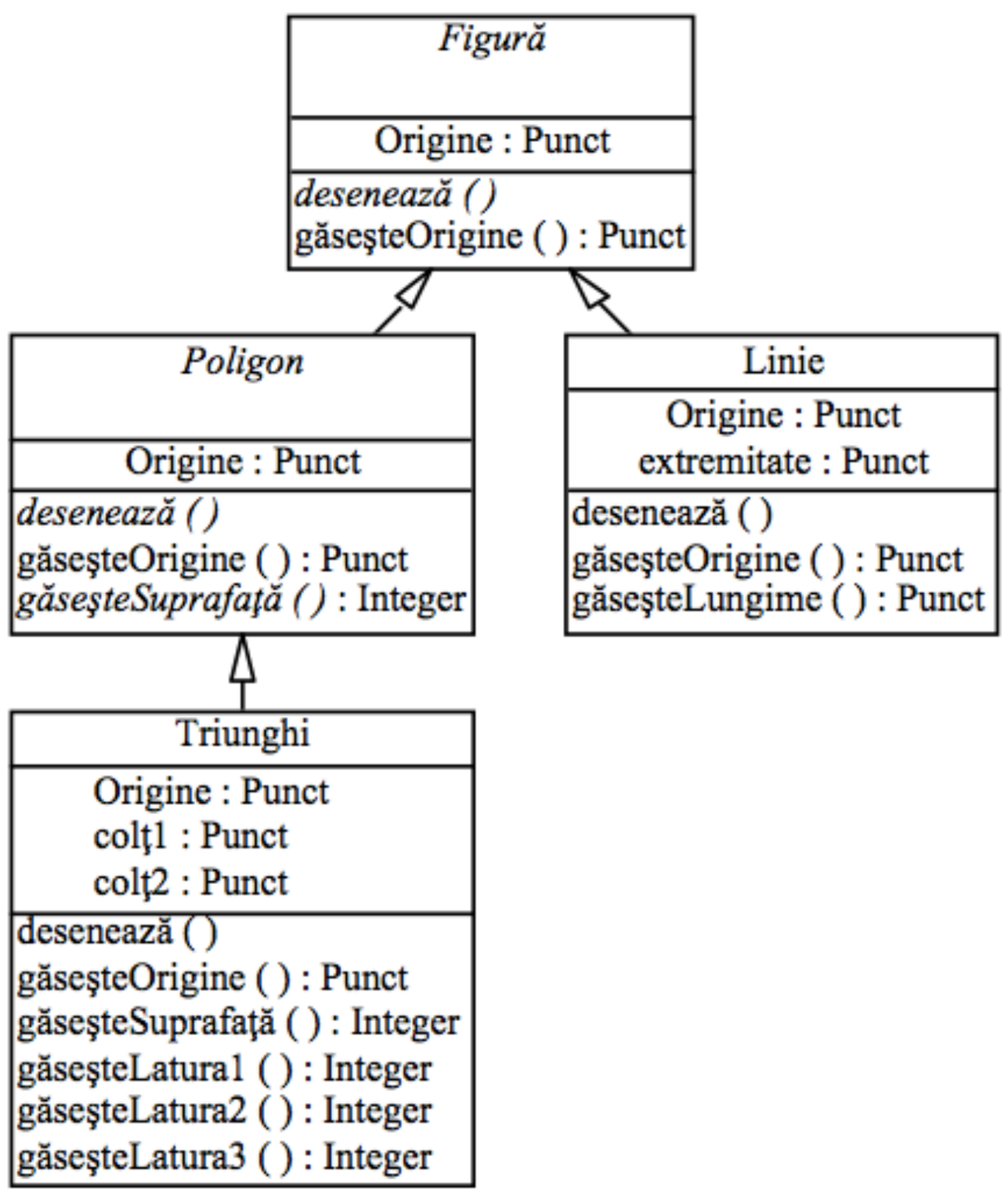


de la http://www.uml-diagrams.org/generalization.html

- Moștenirea multiplă poate fi problematică dacă un copil are mai mulți părinți al căror comportament se suprapune. (*În Java moștenirea multiplă nu este permisă.*)

Clase abstracte

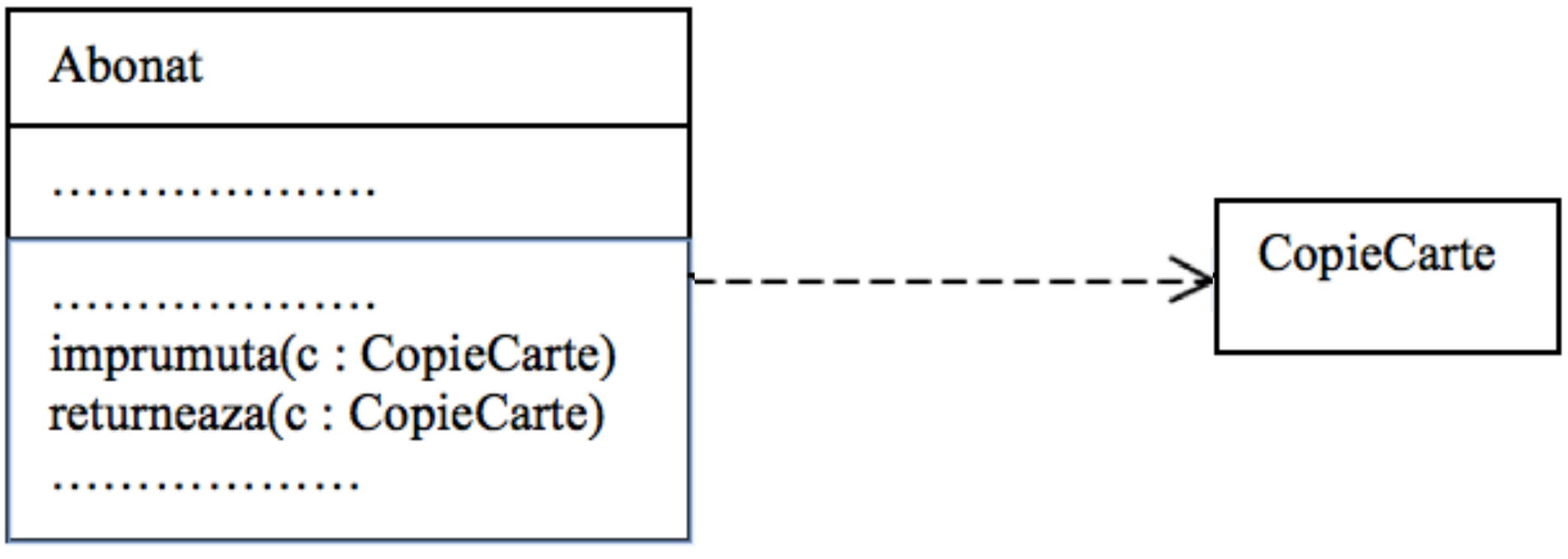
- O **clasă abstractă** este o clasă pentru care nu pot exista instanţe directe.
- O clasă abstractă nu furnizează implementarea pentru cel puţin una dintre operaţiile sale.
- O clasă *concretă* este una care poate avea instanţe directe.
- În UML numele unei clase abstracte se scrie în *italic*.



- Clasele abstracte pot fi folosite ca superclase într-o ierarhie de clase între care există o asociere de tip generalizare. O astfel de ierarhie va avea ca nod rădăcină o clasă abstractă, iar ca noduri frunze clase concrete.

Dependențe

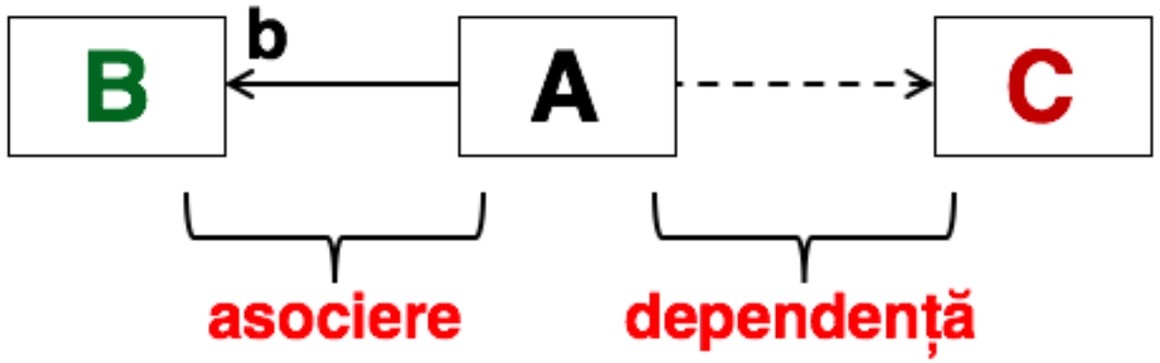
- o clasă A **depinde** de o clasă B dacă o modificare în specificația lui B poate produce modificarea lui A, dar nu neapărat și invers
- cel mai frecvent caz de dependență este relația dintre o clasă care folosește altă clasă ca parametru într-o operație
- notația este o săgeată cu linie punctată dinspre clasa care este dependentă spre cealaltă clasă



Dependențe sau asociere?

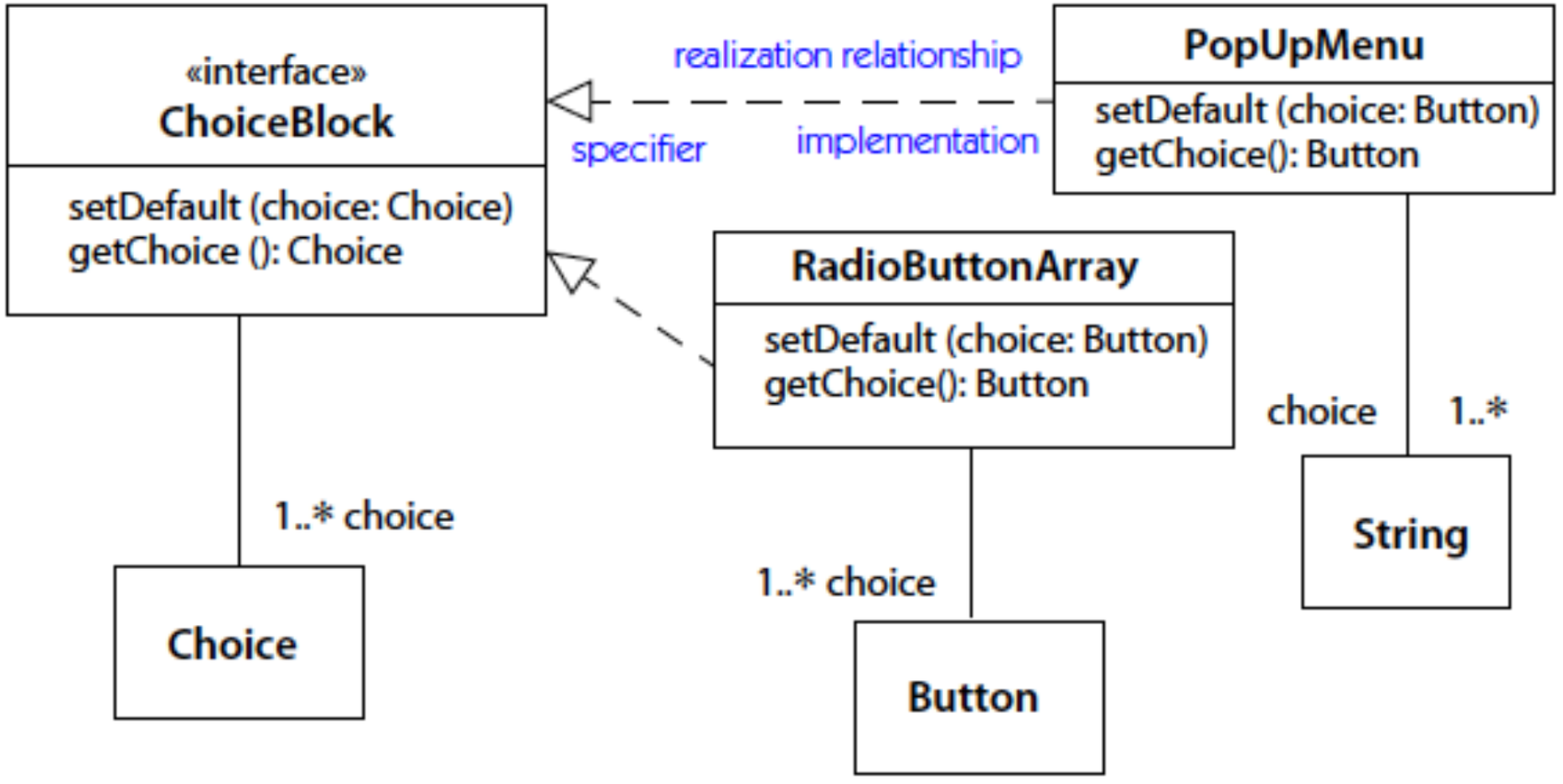
- O clasă ***depinde*** de clasele care apar ca parametrii în funcții și e ***asociată*** cu clasele atributelor

```
public class A
{
   private B b;
   public void myMethod(C c) {
      c.someMethod();
   }
}
```



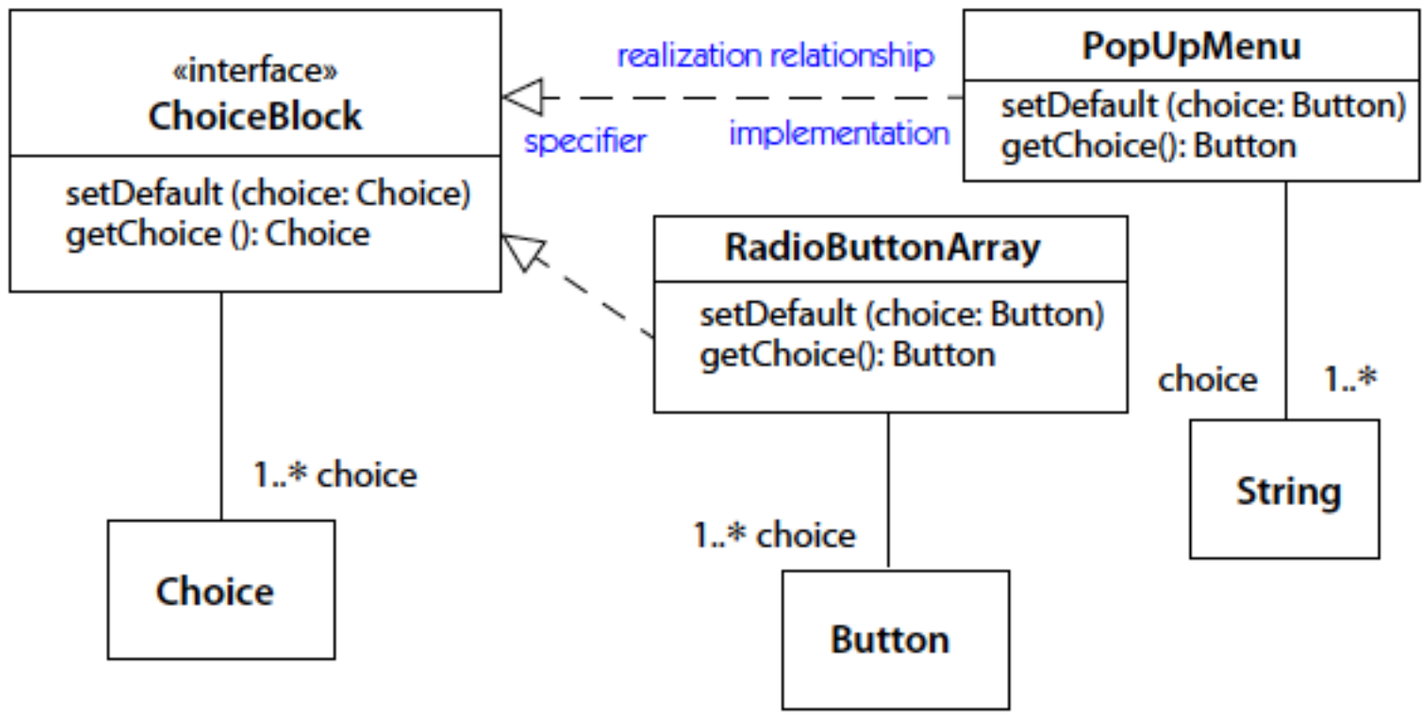
Interfețe

- În UML, o **interfaţă** specifică o colecţie de operaţii și/sau atribute, pe care trebuie să le furnizeze o clasă sau o componentă.
- O interfață este evidențiată prin eticheta **« interface »** deasupra numelui.

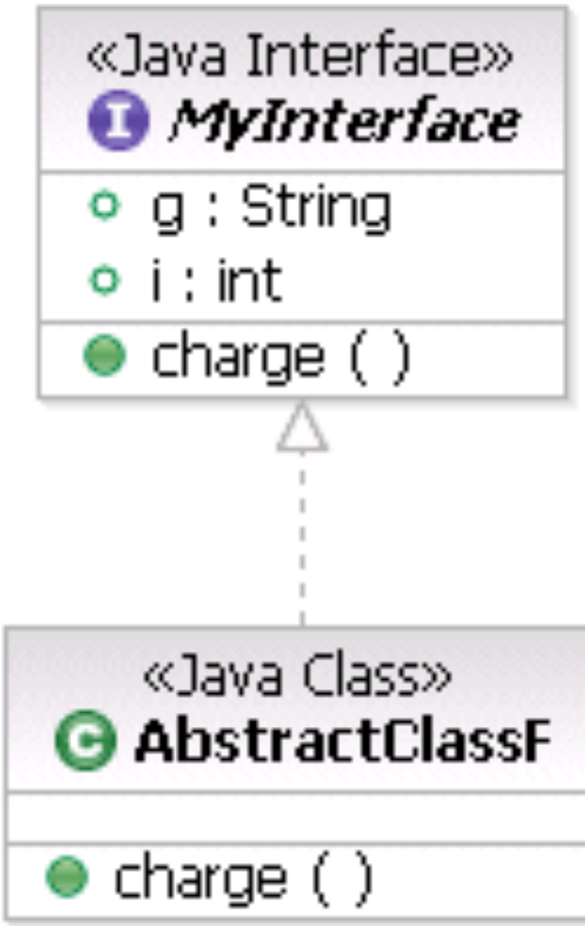


Interfețe și realizări

- D.ex. interfaţa specifică operaţiile unei clase care sunt vizibile în afara acesteia. Ea nu trebuie să specifice toate operaţiile pe care le poate efectua acea clasă, astfel încât aceeaşi clasă poate corespunde mai multor interfeţe, iar o interfaţă poate corespunde mai multor elemente.
- Faptul că o clasă **realizează** (sau corespunde) unei interfațe este reprezentat grafic printr-o linie întreruptă cu o săgeată triunghiulară (alternativ, există și o notație cu cerc, dar nu o discutăm aici).



Realizarea unei interfețe - exemplu în Java



MyInterface.java:

```
public interface MyInterface {
      String g= "";
      int i= 0;

      public void charge (int x);
}
```

AbstractClassF.java:

```
public class AbstractClassF
            implements MyInterface {

      public void charge(int x){
      };

}
```

Interfață sau generalizare?

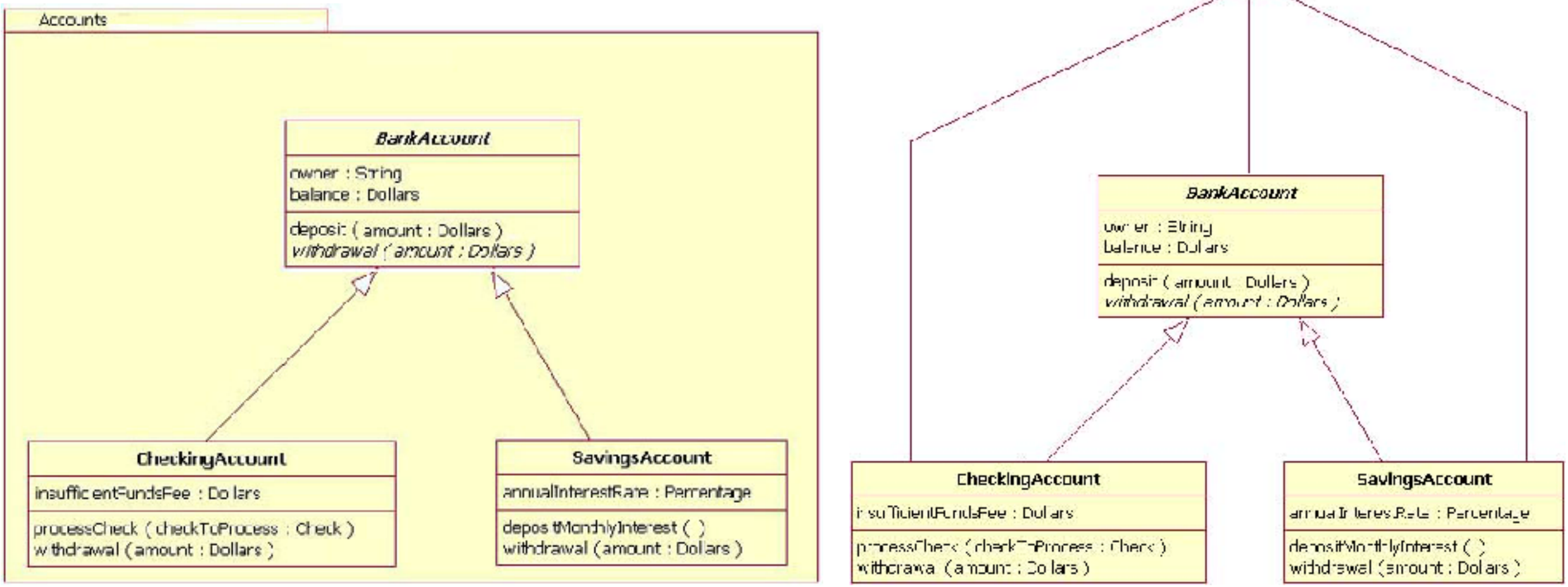
Interfețele și generalizările sunt asemănătoare (realizarea unei interfețe poate fi privită ca un fel de moștenire). Însă există diferențe:

- *Conceptual*: Interfața nu presupune o relație strânsă între clase precum generalizarea.
  - atunci când se intenționează crearea unor clase înrudite, care au comportament comun, atunci trebuie folosită generalizarea.
  - dacă se vrea doar o mulțime de obiecte care sunt capabile să efectueze niște operații comune (d.ex. afișare), atunci interfața este de preferat.
- *Implementare*: Anumite limbaje (Java) oferă doar moștenire simplă, astfel încât interfața este în acest caz singura soluție pentru implementarea moștenirii multiple. O clasă poate moșteni de la o singură superclasă, dar poate implementa mai multe interfețe.

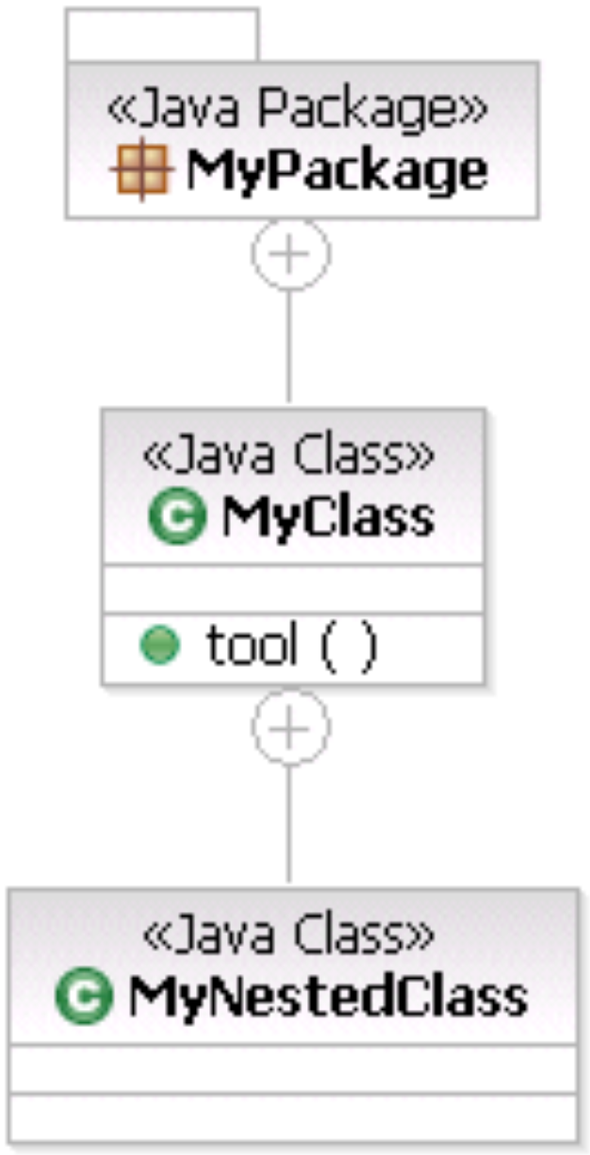
Pachete

Un mod de a organiza clasele într-o diagrama este folosirea **pachetelor**.

- Grafic, un pachet este un dreptunghi cu numele în colțul din stânga-sus, iar clasele care-i aparțin sunt reprezentate în dreptunghi. Alternativ, se pot lega clasele de pachet cu notația ⊕.



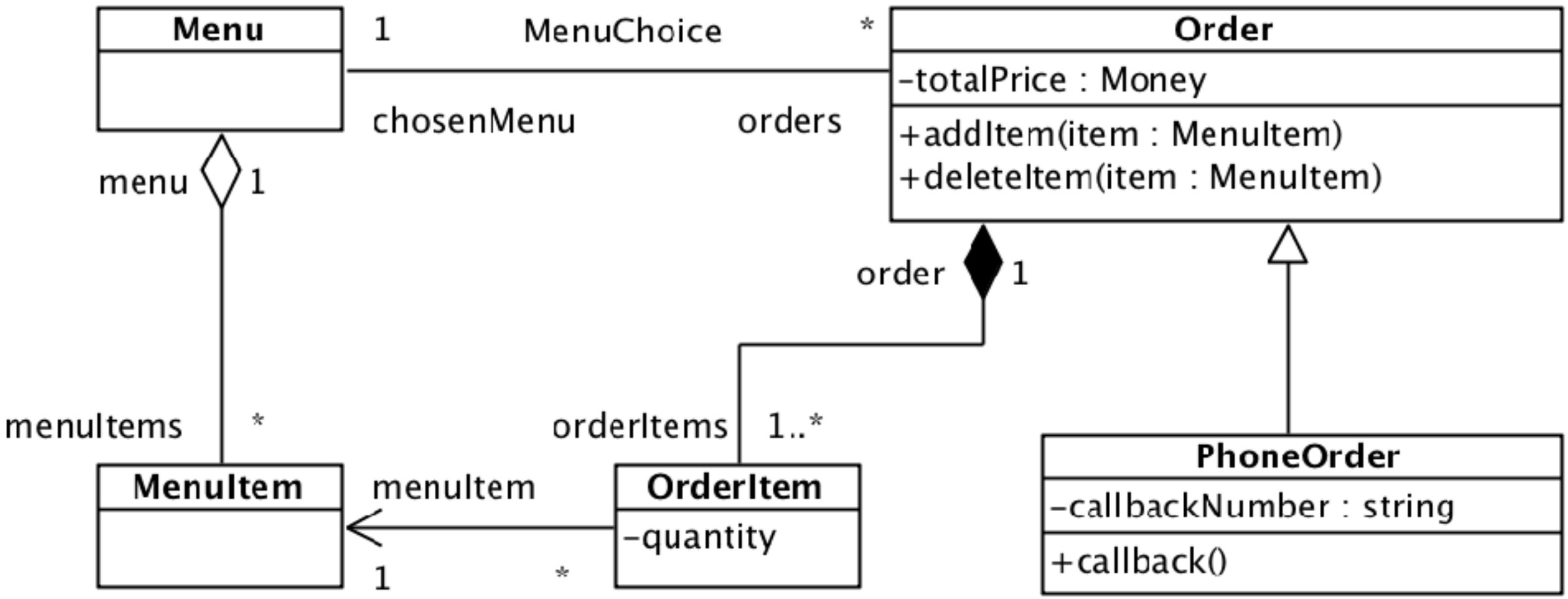
Pachete - implementare în Java




```
/* Package that owns MyClass and MyNestedClass */

package MyPackage;

public class MyClass {
      public void tool (Util util) {
      }

      /* Nested class defined inside MyClass */

      public class MyNestedClass {
      }

}
```

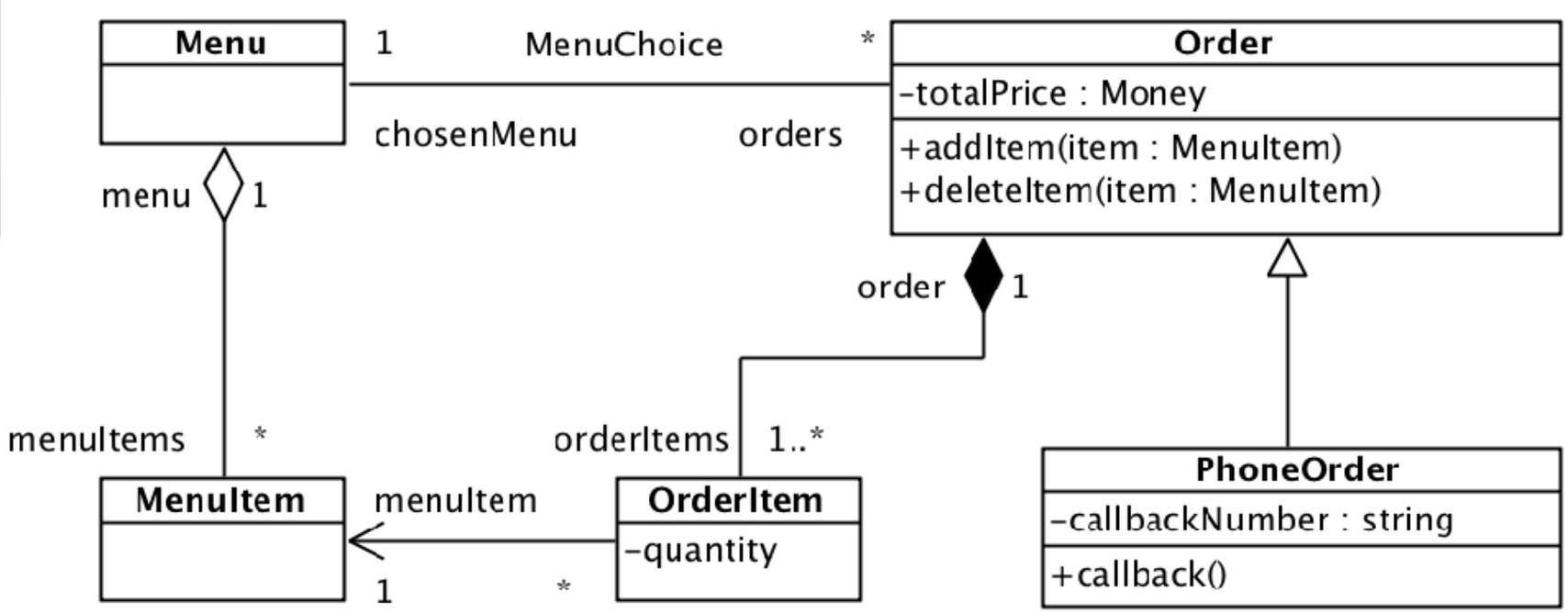
Observație: notația ⊕ poate fi folosită și între clase, nu numai între clase și pachete.

Încă un exemplu



de la `http://msdn.microsoft.com/en-us/library/dd409437.aspx`

Cod generat




```
public class Menu {
  Collection<Order> orders;
  Collection<MenuItem> menuItems;
}
```

```
public class MenuItem {
  Menu menu;
}
```

```
public class OrderItem {
  Order order;
  MenuItem menuItem;
  private int quantity;
}
```

```
public class Order {
  Menu chosenMenu;
  Collection<OrderItem> orderItems;
  private Money totalPrice;
  public void addItem(MenuItem item) {}
  public void deleteItem(MenuItem item) {}
}
```

```
public class PhoneOrder extends Order {
  private string callbackNumber;
  public void callback() {}
}
```