Programare orientată pe obiecte (3)

Cuprins

4 Alte concepte ale orientării pe obiecte

4.1 Relaţii

4.1.1 Relaţia "un fel de"

În exemplul de mai jos, ca şi în următoarele utilizăm o notaţie care nu aparţine unui anumit limbaj de programare (este un pseudolimbaj). În această notaţie **class{...}** reprezintă definiţia unei clase. Între acolade se află două secţiuni, una fiind **attributes** şi alta **methods** care definesc repectiv implementarea structurilor de date şi ale operaţiilor.

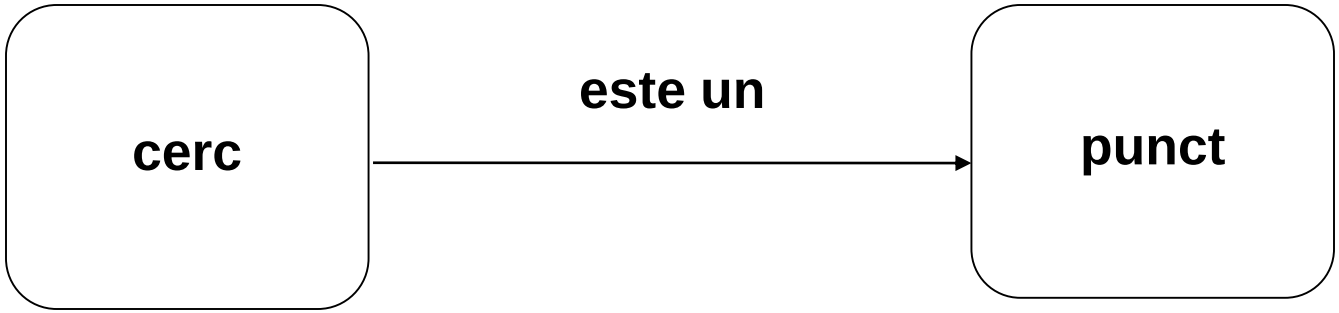
```
class Punct {
 attributes:
 int x, y
 methods:
 setX(int nouX)
 getX()
 setY(int nouY)
 getY()
 }
```

```
class Cerc {
 attributes:
 int x, y,
 raza
 methods:
 setX(int nouX)
 getX()
 setY(int nouY)
 getY()
 setRaza(int nouaRaza)
 getRaza()
 }
```

Clasa *Cerc* "adaugă" o nouă dată membră *raza* şi metode corespunzătoare de acces.

4.1.2 Relaţia "este un/o"

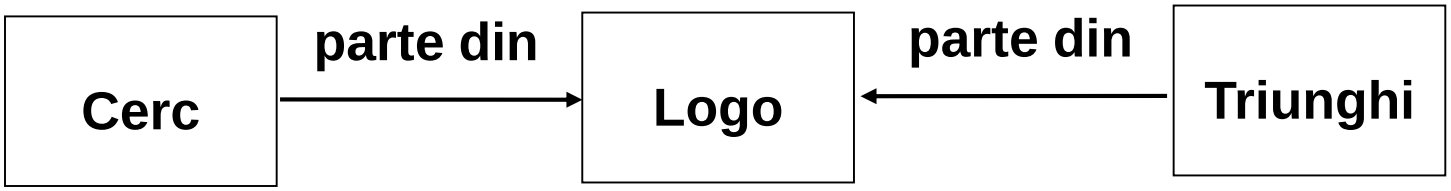
Relaţia precedentă este utilizată la nivelul clasei pentru a descrie relaţiile dintre două clase similare. Dacă sunt create obiecte a două astfel de clase, ne referim la relaţia lor ca la o relaţie "este un/o".



4.1.3 Relaţia "parte din"

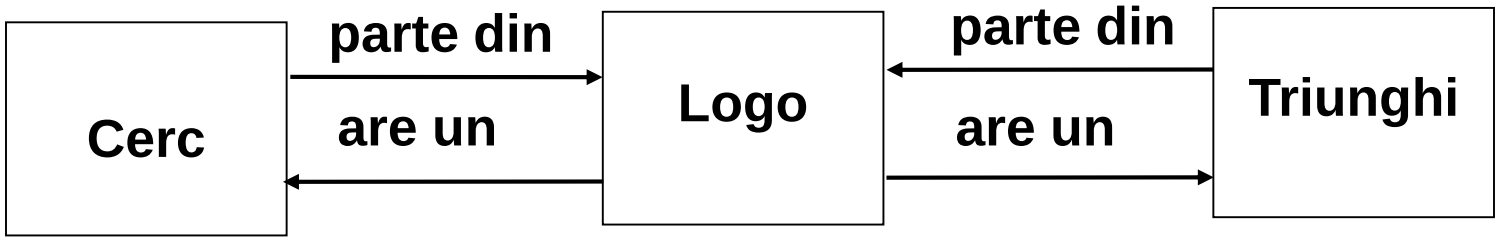
Dacă dorim să avem o figură specială, pe care o vom numi *logo*, formată dintr-un cerc şi un triunghi (presupunând că avem deja definită o clasă *Triunghi*) atunci această figură poate fi reprezentată de o nouă clasă *Logo* care constă din două părţi, *Cerc*, şi *Triunghi*. Spunem deci despre fiecare din aceste două clase că este "parte din" noua clasă.

```
class Logo {
 attributes:
 Cerc cerc
 Triunghi triunghi
 methods:
 set(Punct loc)
 }
```



4.1.4 Relaţia "are un/o"

Această relaţie este inversa relaţiei "parte a".



4.2 Moştenire

Moştenirea ne permite să utilizăm relaţiile "un fel de" şi "este un/o". Aşa cum se arată aici, clasele care sunt "un fel de" altă clasă au proprietăţile comune cu acesta. Exemplul nostru privitor la punct şi cerc poate fi rescris specificând că un cerc *moşteneşte de la* (în engleză *inherits from*) punct.

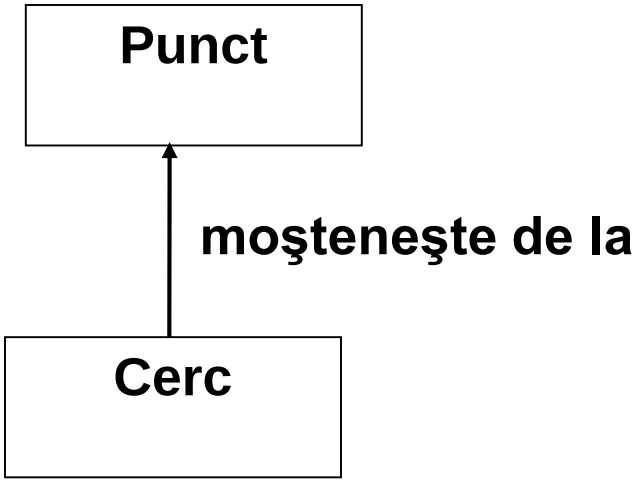
```
class Cerc inherits from Punct {
 attributes:
 int raza
 methods:
 setRaza(int nouaRaza)
 getRaza()
 }
```

Clasa *Cerc* moşteneşte toate datele membre şi metodele de la *Punct.* Nu este nevoie ca acestea să fie definite de două ori; se utilizează numai date şi metode deja definite şi bine cunoscute. La nivelul obiectelor putem să utilizăm un cerc aşa cum am utiliza un punct, deoarece un cerc "este un" punct.

4.2 Moştenire (continuare)

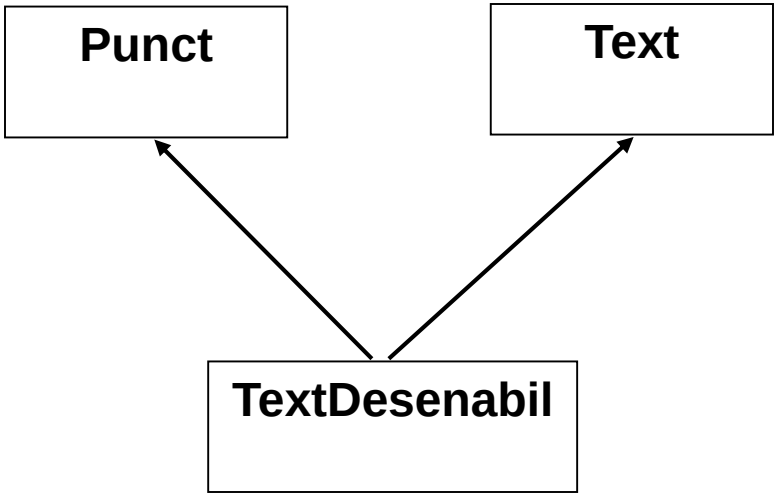
Definiţie (Moştenire). ***Moştenirea*** *este mecanismul care permite unei clase A să posede proprietăţile unei clase B. Spunem "A moşteneşte de la B". Obiectele clasei au astfel acces la atributele şi metodele clasei B fără a fi nevoie ca ele să fie redefinite.*

Definiţie (Supraclasă/Subclasă). *Dacă o clasă A moşteneşte de la o clasă B, atunci B este numită* ***supraclasă*** *a clasei A. A se numeşte* ***subclasă*** *a clasei B.*



4.3 Moştenire multiplă

Prin *moştenire multiplă* se înţelege faptul că o subclasă are *mai mult decât o singură* supraclasă. Aceasta permite subclasei să moştenească proprietăţile a mai mult decât o supraclasă şi să "combine" proprietăţile supraclaselor sale. Presupunem că avem o clasă *Text* care permite manipularea convenabilă a textelor. De exemplu, putem avea o metodă de a *adăuga* text. Ne propunem să utilizăm această clasă pentru a adăuga text la obiectele care reprezintă desene. Dorim să utlizăm şi rutine deja existente, cum este *move( )*, pentru a deplasa textul adăugat. Are deci sens ca textul desenabil să aibă un punct care defineşte poziţia sa în interiorul zonei desenate. Prin urmare, se deriveaza o nouă clasă *TextDesenabil* care moşteneşte proprietăţi de la *Punct* şi *Text*,




```
class TextDesenabil
inherits from Punct,
Text {
 attributes:
 /* Toate mostenite de
la supraclase */
 methods:
 /* Toate mostenite de
la supraclase */
 }
```

4.3 Moştenire multiplă (continuare)

Putem utiliza obiectele clasei *TextDesenabil* atât ca puncte, cât şi ca texte.

Deoarece un *TextDesenabil* "este un" *Punct*, putem să îl deplasăm

```
TextDesenabil textd
...
 textd.move(10)
...
```

Deoarece "este un" *Text*, putem să-i adăugăm alt text:

```
textd.append("Vulpea cea
roscata...")
```

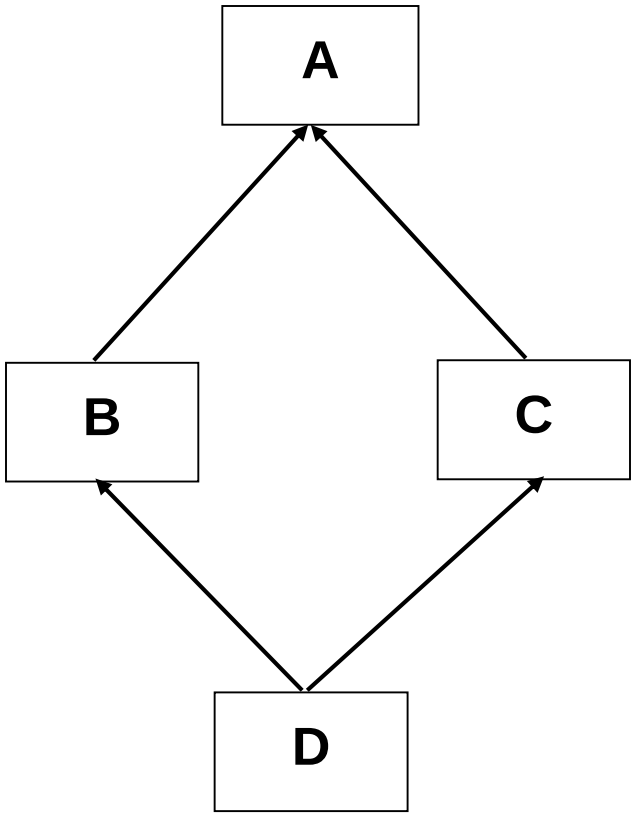
4.3 Moştenire multiplă (continuare)

Definiţie (Moştenire multiplă). *Faptul că o clasă A moşteneşte de la mai mult de o clasă, adică A moşteneşte de la clasele B1, B2,..., Bn, cu n>1 se numeşte* ***moştenire multiplă****. Acest fapt poate introduce* ***conflicte de nume*** *în A dacă cel puţin două din supraclasele sale definesc propietăţi cu acelaşi nume.*

Conflictele de nume au loc dacă mai mult o supraclasă a unei subclase utilizează acelaşi nume pentru atribute sau metode. De exemplu, să presupunem că avem o clasă *Text* care defineşte o metodă *setX()* care atribuie ca valoare textului o secvenţă de caractere "X". Se pune întrebarea: ce va moşteni *TextDesenabil* ? Versiunea din *Punct*, cea din *Text*, sau niciuna dintre ele?

4.3 Moştenire multiplă (continuare)

Un tip special de conflict de nume este introdus dacă o clasă *D* moşteneşte multiplu de la supraclasele *B* şi *C* care sunt şi ele derivate dintr-o supraclasă *A*.



Ce proprietăţi moşteneşte efectiv clasa *D* de la *A* prin supraclasele sale *B* şi *C* ? O soluţie este ca *D* să moştenească din ambele ramuri de moştenire. În această soluţie, *D* posedă **două** copii ale proprietăţilor din *A* : una este moştenită de *B* şi cealaltă de *C*.

Limbajele de programare dau diferite soluţii pentru conflictele de nume.

C++ le tratează cu operatorul de rezoluţie **::** care permite specificarea explicită a supraclaselor cărora le aparţin proprietăţile moştenite.

4.4 Clase abstracte

Definiţie (Clasă abstractă). *O clasă A se numeşte clasă abstractă dacă ea este utilizată numai ca supraclasă pentru alte clase. Clasa A doar specifică proprietăţi. Ea nu este utilizată pentru a se crea obiecte. Clasele derivate trebuie să definească proprietăţile clasei A.*

Clasele abstracte ne permit să structurăm graful de moştenire. Nu dorim să creăm obiecte (instanţe) din ele; dorim numai să exprimăm caracteristicile comune ale unei mulţimi de clase.

5 Din nou concepte ale orientării pe obiecte

5.1 Tipuri generice

Atunci când scriem definiţia unei clase trebuie să putem preciza dacă această clasă defineşte un tip generic. Deoarece nu ştim cu care tipuri va fi utilizată clasa, trebuie să putem defini clasa cu ajutorul unui "locţiitor" la care ne referim ca şi cum ar fi tipul asupra căruia operează clasa. Astfel, definiţia clasei ne este dată ca un *model* sau *şablon* (traducerea termenului englez *template*) al unei clase efective. Definiţia clasei este creată de fapt atunci când declarăm un obiect particular.

Exemplu: clasă de liste de orice tip

```
template class Lista for T {
 attributes:
... /* Structura de date necesara pentru */
 /* implementarea listei */
 methods:
 append(T element)
 T getFirst()
 T getNext()
 bool more()
 }
```

Modelul de clasă *Lista* de mai sus arată ca orice altă definiţie de clasă. Totuşi, prima linie declară *Lista* ca fiind un model pentru diferite tipuri.

Identificatorul *T* este utilizat ca locţiitor pentru un tip real la declararea argumentelor şi a valorilor funcţiilor.

5.1 Tipuri generice (continuare)

Tipul locţiitor trebuie precizat pentru a se crea instanţe ale clasei. Pentru fiecare precizare a tipului compilatorul crează o definiţie de clasă obişnuită.

De exemplu, utilizând modelul de clasă definit mai sus putem să declarăm un obiect listă de persoane dacă există o definiţie a tipului *Persoana*:

```
Lista for Persoana persoanaLista
Persoana oPersoana, altaPersoana
persoanaLista.append(altaPersoana)
persoanaLista.append(oPersoana)
```

Prima linie declară *persoanaLista* ca fiind o listă de persoane. În acest moment, compilatorul utilizează definiţia modelului, înlocuieşte fiecare apariţie a lui *T* cu *Persoana* şi crează o definiţie efectivă de clasă pentru ea.

5.1 Tipuri generice (continuare)

Definiţie (Model de clasă). *Dacă o clasă A este parametrizată cu un tip de clase B, A se numeşte* ***model de clasă****. Atunci când un obiect din A este creat, B este înlocuit cu un* ***tip efectiv****. Aceasta permite definiţia unei* ***clase efective*** *bazată pe modelul specificat pentru A şi tipul efectiv.*

Putem defini modele de clase cu mai mult decât un parametru. De exemplu *cataloagele* sunt colecţii de obiecte în care fiecare obiect este referit printr-o *cheie*. Bineînţeles, un catalog ar trebui să fie capabil să conţină orice tip de obiect. De asemenea, sunt diferite posibilităţi pentru chei. De exemplu, ele pot fi şiruri de numere. Prin urmare vom defini un model de clasă *Catalog* care este bazat pe doi parametri-tipuri, unul pentru chei şi unul pentru obiectele conţinute.

5.2 Legare statică şi dinamică

Definiţie (Legare statică). *Dacă tipul T al unei variabile este asociat în mod explicit cu numele său N prin declaraţie, spunem că N este* ***legată static*** *de T. Procesul de asociere se numeste* ***legare statică****.*

Unele limbaje de programare (Pascal, C, C++, etc.) permit numai legarea statică a variabilelor.

Definiţie (Legare dinamică). *Dacă tipul T al unei variabile cu numele N este implicit asociat prin conţinutul său, spunem că N este* ***legată dinamic*** *de T. Procesul de asociere se numeşte* ***legare dinamică****.*

Unele limbaje de programare (de exemplu JavaScript) permit numai legarea dinamică a variabilelor.

5.3 Polimorfism

Polimorfismul (termen de origine greacă poli=mai multe morphe=formă) permite unei entităţi (de exemplu: variabilă, funcţie sau obiect) să aibă diferite reprezentări (adică sub acelaşi nume se prezintă diferite conţinuturi). Vom pune în evidenţă mai multe tipuri de polimorfism.

Definiţie (Polimorfism (1)). *Conceptul de legare dinamică permite unei variabile să primească diferite tipuri în funcţie de conţinutul său într-un anume moment. Această particularitate a unei variabile este numită* ***polimorfism****.*

5.3 Polimorfism (continuare)

Definiţie (Polimorfism (2)). *Dacă o funcţie (sau metodă) este definită prin combinarea dintre*

- *numele său şi*
- *lista tipurilor parametrilor săi*

spunem că avem ***polimorfism****.*

Acest tip de polimorfism ne permite să reutilizăm acelaşi nume pentru funcţii (sau metode) dacă listele de argumente sunt diferite. Uneori acest tip de polimorfism este numit *supraîncărcare, supraacoperire* sau *supradefinire.*

5.3 Polimorfism (continuare)

Definiţie (Polimorfism (3)). *Obiectele supraclaselor pot să fie încărcate cu obiecte ale subclaselor lor. Operatorii şi metodele subclaselor pot să fie definiţi pentru a fi evaluaţi în două contexte:*

*1. Pe baza tipului **declarat** al obiectului (numit şi **tip static**), ceea ce conduce la o evaluare în interiorul domeniului supraclasei.*

*2. Pe baza tipului **conţinutului** obiectului (numit şi **tip dinamic**), ceea ce conduce la o evaluare în interiorul domeniului subclasei care le conţine.*

*Al doilea tip este numit **polimorfism**.*

5.3 Polimorfism (continuare)

Exemplu de polimorfism de tipul 3 : Definim în pseudolimbaj

1) O clasă

```
class Baza
{
 attributes:

 methods:
 virtual foo()
 bar()
 }
```

2) O clasă care moşteneşte pe prima

```
class Derivata inherits from Baza
{
 attributes:

 methods:
 virtual foo()
 bar()
 }
```

3) O funcţie

```
demo(Baza o)
{
 o.foo()
 o.bar()
 }
```

care se folosesc astfel:

```
Baza obaza
Derivata oderivata
demo(obaza)
demo(oderivata)
```

Presupunem că metodele *foo( )* şi *bar( )* sunt definite astfel încât singurul lucru pe care îl fac este să-şi scrie numele şi clasa în care sunt definite. Atunci ieşirea este:

```
foo() din Baza apelata.
bar() din Baza apelata.
foo() din Derivata apelata.
bar() din Baza apelata.
```

Cuvantul cheie *virtual* arată că metodele *foo( )* sunt apelate conform cu tipul dinamic al obiectului invocant.