9 Derivarea claselor

9.1 Programarea orientată pe obiecte

Aceasta este caracterizată prin
- *moştenire*
- *legături dinamice*

Programarea orientată pe obiect extinde tipurile de date abstracte astfel încât ele să permită relații de forma *tip/subtip*. În loc să reimplementeze caracteristici comune altor clase, o clasă poate moşteni date membre şi funcții ale altor clase. În C++, moştenirea este suportată prin derivarea claselor.

Care au fost problemele care au dus la necesitatea derivării claselor?

Să luăm în considerare clasa `Screen`, definită în capitolele anterioare. Evoluția soft-ului din ultimii ani a promovat lucrul cu ferestre. Cum putem să reprezentăm o fereastră? De exemplu, ca un `Screen` cu facilități suplimentare (redimensionare, posibilități de mutare, pe un ecran să poată fi definite mai multe ferestre, etc.).

- O fereastră trebuie să-şi cunoască nu numai dimensiunea, ci şi poziția. De asemenea, ar avea nevoie de aceiaşi membri pe care i-a definit clasa `Screen`. Unii dintre aceştia pot fi reutilizați direct, alții trebuie implementați.
- O fereastră este un fel de `Screen` specializat. Este mai degrabă un subtip extins al lui `Screen` decât un tip de dată independent.
- La un anumit nivel de implementare, ele pot împărți date şi funcții membre.

Reprezentarea cea mai apropiată de această idee ar fi:

```
class Window
{
   public:  // ...
   private:
      Screen base;
// ...
};
```

Problemele care apar sunt generate de faptul că datele membre ale lui `Screen` sunt private; nu sunt direct accesibile în `Window`. Pentru ca membrii `Screen` să fie direct accesibile ar trebui făcută modificarea `public: Screen base`, care duce la încălcarea principiului de incapsulare a datelor.

```
Window w;
w.base.clear();
```

Să presupunem acum că `home()` a fost redefinită pentru `Window`. Cele două funcții

```
w.home();
w.base.home();
```

invocă două funcții diferite, situație care poate duce uşor la erori de programare. Pentru a păstra o sintaxă uniformă de tipul

```
w.clear();
```

trebuie definită o funcție de interfață de tipul

```
inline void Window::clear() { base.clear(); }
```

Daca clasa `Window` este extinsă, implementarea poate da bătăi de cap serioase. De exemplu un `Menu` este un tip de `Window`. Va avea propriile date membre şi funcții membre. Un meniu va împărți date cu `Window` sau chiar şi cu `Screen`.

Este nevoie de o soluție mai bună: *moşternirea*. Derivând `Window` din `Screen`, `Window` are acces la membrii nepublici `Screen` ca şi cum ar fi definiți în `Window`. Pentru a permite derivarea, sintaxei claselor i-au fost aduse două modificări:

1) introducerea *listelor de derivare* prin care se specifică
- clasele de bază (moştenite)

- tipul moştenirii (publică, privată)

2) introducerea secţiunii `protected`; membrii acestor secţiuni se comportă ca membri publici pentru clasele derivate public şi ca privaţi pentru restul programului.

O clasa derivată poate fi ea însăţi clasa de bază într-o nouă derivare.

Exemplu:
```
class Screen
   { ... };
class Window : public Screen
   { ... }; // mosteneste toti membri Screen
class Menu   : public Window
   { ... }; // mosteneste toti membri Window si Screen

Window w;
w.clear().home();
Menu m;
m.display();
```

Relaţiile speciale dintre tipurile de bază şi cele derivate permit efectuarea unor conversii standard predefinite. O clasă derivată poate fi atribuită oricărei clase publice de bază.

Exemplu:
```
Menu m;
Window &w = m;
Screen *ps = &w;
```

Moştenirea a promovat un stil generic de programare în care nu este necesară cunoaşterea tipului actual al obiectelor cu care se lucrează. Fie o funcţie externă:
```
void displayScreen( Screen *s );
```
Funcţia `displayScreen` va putea fi invocată cu un argument de tipul oricărei clase derivate public din `Screen`, indiferent de nivelul ierarhic pe care se găseste.

Exemplu:
```
Screen s;
Window w;
Menu m;
displayScreen(&s);
displayScreen(&w);
displayScreen(&m);
displayScreen(&IntArray);
   // Eroare. IntArray nu e in relatie de mostenire cu Screen
```

Dacă fiecare din clasele `Screen`, `Window` şi `Menu` au câte o funcţie membră numită `display()`, atunci `displayScreen()` poate fi definită astfel
```
void displayScreen(Screen *s)
{
   s->display();
}
```

Mecanismul prin care se alege funcţia membră prin intermediul unui obiect invocator pentru care în momentul compilării se cunoaşte doar tipul de bază se numeşte *legare dinamică*. În C++, funcţiile tipului de bază supuse legăturilor dinamice se declară `virtual`.

Exemplu:
```
class Screen
{
   public:
      virtual void display();
// ...
};
```

O funcţie se va declara virtuală în următoarele situaţii:

1) clasa va fi probabil moştenită (va face obiectul unei derivari);
2) implementarea funcţiei este dependentă de tip.

Prin funcţiile virtuale stilul de programare nu este numai generic ci şi extensibil: funcţia `displayScreen()` nu suferă nici o modificare în cazul în care se introduce un nou tip derivat din `Screen`, `Window` sau `Menu`.

Relaţia dintre tipul de bază şi cel derivat se numeşte *ierarhie de derivare* sau *ierarhie de moştenire*. O derivare din mai multe tipuri de bază se va numi *graf de derivare* sau *graf de moştenire*.

Observaţie:

- *Simula* = primul limbaj care a utilizat mecanismul de clasă pentru tipuri abstracte extinse
- *SmallTalk* = limbajul care introdus termenul de orientare pe obiect pentru a denumi încapsularea tipurilor definite de utilizator.

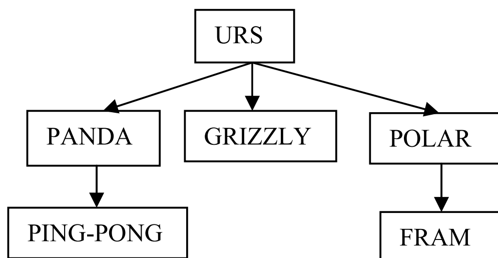
9.2 Reprezentarea unei derivări

În cele ce urmează, vom analiza un graf de moştenire cu ajutorul unei grădini zoologice. Pe mulţimea animalelor din zoo pot fi definite mai multe niveluri de abstractizare:

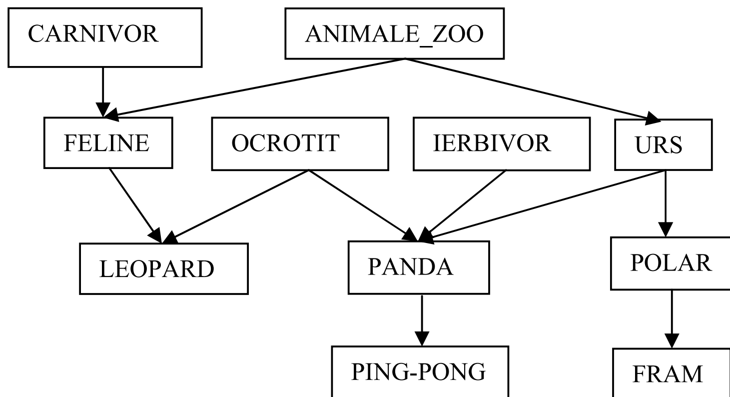
- la nivel de *individ* : PingPong
- la nivel de *specie*: panda
- la nivel de *familie*: urs
- la nivel de *regn*: animale din zoo

Alte tipuri de abstractizări pot fi : ierbivore, carnivore, ocrotite, etc.

Să ilustrăm o ierarhie de derivare pentru familia urşilor:



Această ierarhie de derivare este *simplă*, adică fiecare clasă derivată are o singură clasă de bază la nivelul imediat superior. Printr-o ierarhie de moştenire simplă nu pot fi ilustrate toate caracteristicile unei clase, de ex. clasa `Panda`, însă se poate printr-un graf de *moştenire multiplă*.



Clasa AnimaleZoo cuprinde setul de membri (date şi funcţii) comune tuturor animalelor din zoo. Derivările din această clasă vor defini doar particularităţile claselor respective. O clasă care este bază pentru o ierarhie de derivare se numeşte *superclasă*.

Exemplu:
```
// clase de baza
class AnimaleZoo { ... };
class Ocrotit    { ... };
class Carnivor   { ... };
class Ierbivor   { ... };

// derivari simple
class Urs : public AnimaleZoo { ... };

// derivari multiple
class Feline : public AnimaleZoo, private Carnivor { ... };
class Panda  : private Ocrotit, public Urs, private Ierbivor { ... };

// obiect de tip Panda
Panda PingPong;
```

Sintaxa claselor de bază diferă de cea a claselor obişnuite prin două aspecte:

1) membrii la care clasele derivate trebuie să aibă acces, dar nu sunt `public`, devin `protected`;

2) funcţiile a căror implementare depinde de derivările ulterioare sunt declarate `virtual`.

Exemplu:
```
class AnimalZoo
{
   public:
      AnimalZoo ( char *nm, char *file, short loc );
      virtual ~AnimalZoo();
      virtual void draw();
      void locate();
      void inform();
   protected:
      char *name;
      char *infoFile;
      short location;
      short count;
};

class Urs : public AnimalZoo
{
   public:
      Urs(char *nm, char *fil, short loc, char dn, char sp);
      ~Urs();
      void locate(int);
   protected:
      char isDanger;
      char specia;
};
```

Limbajul nu impune limite asupra numărului de clase de bază. Însă fiecare clasă de bază poate apărea o singură dată! Tipul derivării se specifică înaintea fiecărei clase de bază. Implicit derivarea este de tip `private`. Nu este nici o restricţie în specificarea unei clase de bază ca publică sau privată. Membrii unei clase derivate sunt accesaţi conform nivelului lor de încapsulare şi pe baza numelui lor. Dacă clasa derivată defineşte un membru cu acelaşi nume ca al unuia din clasa de bază, atunci membrul din clasa derivată il va domina (*ascunde*) pe cel din clasa de bază.

Exemplu:
```
void ex( Urs& ursus )
{
   if( ursus.isDanger )
      ursus.locate(HIGH); // HIGH - constanta intreaga
   ursus.AnimalZoo::locate(); // ursus.locate(); =>eroare, lipseste
                              // primul argument
   ursus.inform();
}
```

Unor membri ai unor clase de bază trebuie să li se specifice domeniul (clasa) atunci când:

1) clasa derivată utilizează un membru cu acelaşi nume;

2) sunt mai multe clase de bază care conțin un membru cu acelaşi nume.

Inițializarea membrilor claselor de bază se face prin lista de inițializare a membrilor din definitia constructorului.

Exemplu:
```
Urs::Urs( char *nm, char *fil, short loc, char dn, char cp )
: AnimalZoo( nm, fil, loc ), specie(sp), isDanger(dn)
{
}
```

- Un tip de bază care nu are constructor sau care are un constructor implicit nu trebuie specificat în lista de inițializare a membrilor.

- Un tip de bază trebuie să fie inițializat cu argumentele cerute de unul din constructorii săi, cu un obiect de acelaşi tip sau dintr-un tip derivat public.

Exemplu:
```
Urs::Urs( AnimalZoo& z ) : AnimalZoo( z ) { ... };
Urs::Urs( const Urs& u ) : AnimalZoo( u ) { ... };
```

Constructorii claselor de bază sunt apelați înaintea constructorului clasei derivate. Astfel, prin lista de inițializare a membrilor NU pot fi explicit inițializati membrii claselor de bază, pentru că aceştia au fost deja inițializați.

Exemplu:
```
Urs::Urs( char *nm ) : name (nm) { ... }
   // eroare
```

9.3 Ascunderea informației la derivare

O clasă derivată moşteneşte toți membrii clasei de bază, dar nu are nici un fel de acces la membrii privați ai acesteia.

Declarațiile `friend` dintr-o clasă derivată conferă un nivel de acces la membrii clasei de bază identic cu cel al fiecărui membru al clasei derivate.

Membrii neprivați ai unor clase de bază publice îşi păstrează aceleaşi nivele de acces şi în cadrul claselor derivate.

Membrii neprivați ai unor clase derivate privat devin membri privați ai clasei derivate.

Tabelul următor prezintă modul în care se modifică nivelul de încapsulare al membrilor din clasa de bază în prezența derivării.

		TIPUL DERIVĂRII		
		PUBLIC	**PROTECTED**	**PRIVATE**
NIVELUL DE ÎNCAPSULARE DIN CLASA DE BAZĂ	**PUBLIC**	PUBLIC	PROTECTED	PRIVATE
	PROTECTED	PROTECTED	PROTECTED	PRIVATE
	PRIVATE	NO ACCESS	NO ACCESS	NO ACCESS

Membrii clasei de bază pot fi exceptați de la regulile de derivare specificându-le numele, şi numai numele, în secțiunile `public` sau `protected` ale tipului derivat. Noul nivel de acces trebuie să coincidă cu cel din clasa de bază.

Exemplu:

```
class AnimalZoo
{
   public:
      void inform();
// ...
};

class Rozatoare : private AnimalZoo
{
   public:
      AnimalZoo :: inform;   // ok
// ...
};

class Rozatoare : private AnimalZoo
{
   protected:
      AnimalZoo::inform;  // eroare, nivelul de incapsulare nu
                          // coincide cu cel din clasa de baza
// ...
};
```