Les concepts de la programmation objet

Walid-Khaled HIDOUCI
chargé de cours à l'Institut National d'Informatique - Alger
hidouci@ini.dz

Sommaire

1 - INTRODUCTION

De nos jours, beaucoup de langages de programmation, de systèmes de bases de données et d'autres logiciels ont le qualificatif d'orienté objet. A défaut d'une standardisation initiale, la terminologie utilisée pour décrire ces systèmes est devenu trop importante et anarchique. En conséquence, il est de plus en plus difficile d'aborder le domaine de l'orienté objet et bénéficier ainsi de ses nombreux avantages.

Les domaines d'application des concepts orientés objet sont très variés et touchent presque l'ensembles des activités en informatique. Historiquement issue des langages de programmations (tel que SIMULA, SMALLTALK, …), l'orientation objet a été appliquée aux domaines des bases de données, des systèmes d'exploitation, des logiciels de CAO, de Bureautique, de télécommunication, …

La notion de programmation objet peut être vue comme une extension de la programmation modulaire. L'idée principale est la réutilisation du code pour faciliter l'extensibilité et le développement incrémental d'applications complexes.

Dans le domaine des bases de données, l'apport des concepts objets se situe surtout au niveau modélisation. Le modèle objet est beaucoup plus riche sémantiquement que le modèle relationnel et permet une représentation aisée, et pratiquement, fidèle du monde réel.

Ce document présente les différentes notions relatives à l'orientation objet ainsi que leur mise en pratique à l'aide du langage de programmation C++. Il est destiné à être utilisé comme support de cours sur l'orientation objet, à des gens ayant déjà une première expérience en programmation.

Le choix du langage C++ pour présenter les concepts de l'orienté objet s'est facilement imposé pour les raisons suivantes :
- La mise en pratique des concepts objets est un point crucial pour la bonne compréhension du domaine.

- De nos jours, C++ est considéré comme un des langages les plus utilisés dans le développement d'applications toute architecture confondu.
- C++ est un langage standardisé (norme ANSI) et existe dans pratiquement toutes les plates-formes (Alpha, Sun, PC, HP, …) ce qui assure une portabilité maximale à vos applications.
- Bien que C++ ne soit pas purement un langage objet (comme smalltalk), il renferme malgré tout, les principale notions de l'orientation objet. De ce fait, deux types de programmation : procédurale et orientée objet, sont possibles avec C++. Ce qui permet un apprentissage graduel des nouvelles notions objets.

Les langages orientés objet sont répartis en trois grandes catégories :
- Les langages à classes d'usage général et très répandus.
- Les langages à frames orientés vers la représentation des connaissances en IA.
- Les langages à acteurs orientés vers la programmation parallèle et les systèmes ouverts.

Tous ont en communs certaines caractéristiques dont l'encapsulation et l'héritage.

2) NOTION D'OBJET

2-1) Encapsulation :

Un **objet** est une entité ayant une **identité** unique et regroupant à la fois des données et des procédures (et/ou des fonctions). Cette entité (logiciel) est sensée décrire un objet du monde réel. Le terme 'objet' est pris dans le sens large, cela peut être un objet physique (comme une voiture, un avion, …) ou un objet immatériel (comme un concept, une situation, une relation …). Ce regroupement des données et des opérations qui les manipulent, et connue sous le nom d'encapsulation.

Les données d'un objet sont appelés **'Attributs'** ou encore **'Champs'** et désignent sa partie statique, alors que les procédures ou fonctions sont appelées **'Méthodes'** et représentent sa partie dynamique. Les attributs et méthodes d'un objet, représentent ses **'propriétés'**. (voir Fig II-1)

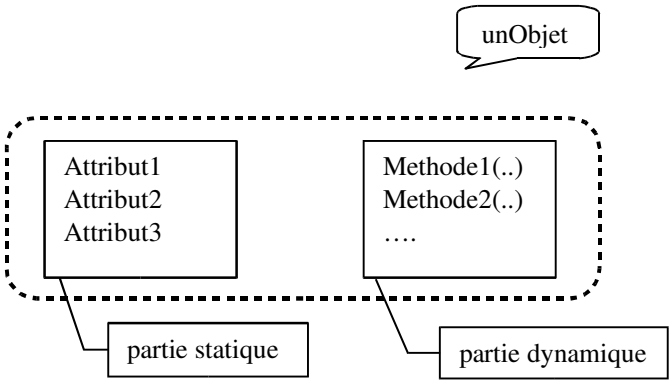


Fig II-1 : Objet = Identité + Attributs + Méthodes

La partie statique représente l'**état** de l'objet alors que la partie dynamique représente son **comportement**. Cette dualité de l'objet qui permet de regrouper sous une entité unique valeurs et comportements, facilite et rend plus naturel, le processus de modélisation du monde réel.

- Quelques exemples :

1) Une tasse de café est un objet (Fig II-2), dont les propriétés pourraient être :

la couleur, la quantité de boisson, sa température, … (représentant l'état de la tasse)

remplir( ) , boire(q), laver( ), … (représentant les opérations applicables)

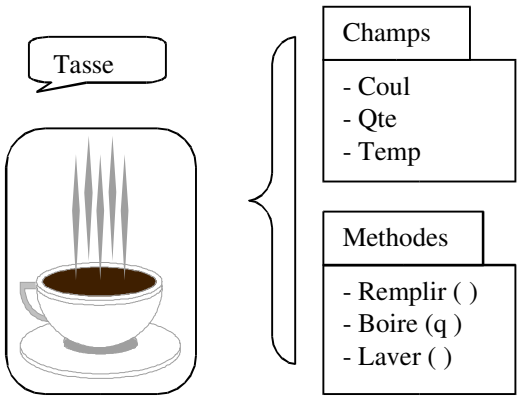


Fig II-2 : Objet représentant une tasse de café

2) Une file d'attente de personnes devant un guichet, peut aussi être représentée par un objet ayant comme attributs :

Tab : un tableau contenant les identifiants de chaque personne dans la file

Nb : le nombre de personne dans la file à un moment donné.

et comme méthodes :

Ajouter : pour enregistrer une nouvelle arrivée dans la file

Retirer : pour enregistrer un départ de la file.

La figure II-3 schématise ce type d'objet.

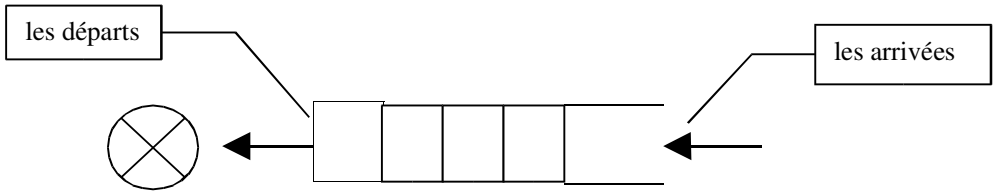


Fig II-3 : Objet decrivant une file FIFO

2-2) Abstraction des données :

En général, la partie statique d'un objet reste cachée au monde extérieur, c'est sa partie **privée**, alors que la partie dynamique (les méthodes) est visible et joue le rôle d'interface entre l'objet et son entourage, c'est sa partie **publique**.

Ce procédé de cacher les données du monde extérieur est ce qu'on appelle l'abstraction des données.

Ainsi seule les méthodes d'un objet pourront consulter et/ou modifier les valeurs des différents attributs. Ceci conduit à une programmation modulaire et favorise l'extensibilité des logiciels.

Dans le précédent exemple sur la file d'attente, les programmes utilisant cet objet ne pourront pas directement manipuler les champs Tab et Nb, seule les opérations Ajouter et Retirer sont connues. Ainsi, si par exemple on modifie l'objet 'file d'attente' pour qu'au lieu d'utiliser un tableau, on utilise une liste, seule la partie statique et l'implémentation des méthodes seront modifiées. L'interface (l'en-tête des opérations Ajouter(p), Retirer(p)) n'a pas à être modifiée, ce qui assure que les programmes utilisant l'objet 'file d'attente' restent corrects avec la nouvelle représentation.

2-3) Identité d'objet :

Chaque objet créé possède un identifiant unique indépendant de son état (ses valeurs de champs). De ce fait il est tout à fait possible que deux ou plusieurs objets ayant les mêmes propriétés soient égaux en valeur (ils possèdent le même état et comportement), mais chacun possède sa propre identité, c'est ce qui permet de les différencier au dernier recours.

Dans l'exemple 1, Il se pourrait que plusieurs tasses aient exactement les mêmes propriétés à un moment donné (couleur, quantité, température, …) mais néanmoins elle désignent des objets distincts ayant chacun sa propre identité (voir Fig II-4)

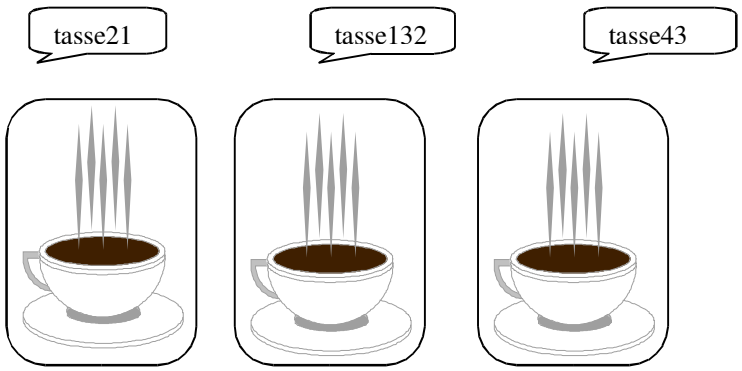


Fig II-4 : Trois objets égaux en valeur mais différents en identité

Dans les langages de programmation, l'identité d'objet est souvent matérialisée soit par le nom d'une variable, soit par un pointeur vers la zone mémoire contenant l'objet.

2-4) Communication entre objets :

L'abstraction des données implique que la communication avec un objet n'est possible qu'à travers ses méthodes (jouant le rôle d'interface avec le monde extérieur).

Dans un programme objet, le traitement n'est effectué que par un ensemble d'objets communicant entre eux, par invocation de méthodes publiques. La terminologie utilisée pour décrire l'appel d'une méthode d'un objet est l'**envoi de message**.

Quand un objet O1 invoque la méthode M (x,y,...) de l'objet O2, il y a envoi du message :

« O2 . M (x,y,…) »

Un message est donc composé de l'objet receveur (O2), du nom de la méthode invoquée (M) et des paramètres effectifs (x, y, …) de l'appel.

Dans les langages purement objet, chaque instruction est un envoie de message entre un objet émetteur et un objet receveur. Par exemple en SMALLTALK, l'expression suivante : 3 + 2 est une invocation de la méthode + qui fait partie de l'interface de l'objet 3, avec comme paramètre l'objet 2. Le résultat est un nouvel objet (5).

3) PRESENTATION DE C++

C++ est un langage inspiré du langage C avec des concepts orientés objet. Il a été Développé par Bjarne Stroustrup en 1983 (version 1.0 commercialisée en 1985, version 2 en 1989 et version 3 en 1992) aux laboratoires Bell AT&T. Depuis, C++ est devenu un standard de facto dans le monde de la programmation avancée.

En C++ (comme en C) on travail souvent avec des fichiers librairie (ou en-tête : headers). Ces fichiers (généralement avec l'extension '.h') définissent des constantes, des types et des fonctions d'usage générale et constituent une extension au langage. Ils sont écrit en source (C++).

L'utilisation d'un fichier en-tête dans un programme se fait par la directive #include<nom.h> qui permet de charger le fichier nom.h se trouvant dans les répertoires standards d'inclusion, alors que la forme #include "nom.h" permet d'inclure le fichier nom.h depuis le répertoire courant.

3-1) Les Types de Données :

Il existe trois (03) types fondamentaux :

Les caractères représentés par le type : *char* de taille un octet.
Les nombres entiers représentés par le type : *int* de taille 2 octets
Les nombres réels représentés par les types : *float* (4 octets) ou *double* (8 octets)

On peut déclarer une variable n'importe où dans un bloc d'instruction :

```
char c;
int i;
```

Ces deux instructions permettent de créer deux variables c et i pouvant contenir respectivement un caractère et un entier.

On peut aussi déclarer une variable et l'initialiser :

```
int x = 10;
```

Une **constante** est déclarée à l'aide du mot '*const*' placé avant le type :

const float PI = 3.141;

La définition d'un **tableau** se fait à l'aide de l'opérateur [] (type nom[taille]). L'exemple suivant déclare un tableau t de 10 caractères (t[0] t[1] t[2]… t[9]) :

char t[10];

On peut aussi définir et initialiser un tableau comme le montre la déclaration suivante :

```
int a[5] = {23 , 4 , 65 , 34 , 8};
int b[ ] = {1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 };
```

Dans la première déclaration, la taille du tableau (5) a été mentionnée alors que dans la deuxième elle a été déduite (10).

Un **pointeur** (variable contenant des adresses mémoire) est défini à l'aide de l'opérateur * (type *nom). Ainsi : *int *p;* déclare p comme étant un pointeur vers un entier, c'est à dire que *p* peut contenir l'adresse mémoire d'une variable de type *int*.

```
int x = 10;      // déclaration de la variable entière x et initialisation à 10
int *p;          // déclaration d'un pointeur vers un entier
p = &x;          // affectation de l'adresse de x à p. maintenant p pointe x
*p = *p + 1;     // incrémentation indirecte de x. maintenant x vaut 11
```

On peut créer une variable **dynamique** à l'aide de l'opérateur *new Type;* ainsi dans l'exemple suivant on crée une variable entière (*int*) référencée par le pointeur p :

```
int *p;          // déclaration d'un pointeur vers un entier
p = new int;     // création d'une variable dynamique pointée par p
*p = 10;         // affectation de 10 à la variable dynamique
```

Une variable de type tableau est en fait un pointeur constant vers le premier élément du tableau. *t[i]* devient équivalent à **(t+i)*

Les **chaînes** de caractères peuvent être représentées soit par des tableaux de caractères soit par des pointeurs vers un caractère. Une constante chaîne de caractères est une suite de caractères comprise entre guillemets: *"salut";* Sa taille est le nombre de caractères entre guillemets plus un, car chaque chaîne se termine par le caractère '\0' (le caractère de code 0), ainsi la chaîne *"salut"* occupe 6 octets et non pas 5.

On peut déclarer et initialiser une chaîne de caractères :

char ch[6] = "salut";

Par contre on ne peut pas affecter une chaîne en dehors d'une déclaration:

char ch[6];
ch = "salut"; // une erreur !

Pour ce faire, on doit affecter la chaîne caractère par caractère, ou bien utiliser les primitives de librairie 'string.h', comme '*strcpy(destination,source);*'

char ch[6];
strcpy(ch,"salut");

En C++, la manipulation des tableaux doit être faite avec précaution, car le compilateur ne vérifie pas les dépassement de bornes par les indices d'un tableau. Vous devez donc être sûre que la taille du tableau recevant la chaîne est assez grande pour la contenir.

On peut aussi utiliser les pointeurs pour manipuler les chaînes de caractères :

*char *p = "salut";   // p pointe vers une zone de 6 octets (5 caractères + '\0')*

3-2) Structures de contrôle et instructions standards

L'affectation a la forme *var = expression;* De plus elle retourne la valeur de '*expression'*. Par exemple il est tout à fait correcte d'écrire : *x = y = (3+20/3)*2;*

L'instruction composée est une séquence d'instructions terminée chacune par un ';' et est de la forme : *{ instruction1; instruction2; ... instruction n; }*

La conditionnelle a la forme suivante:

if (condition) instruction1; else instruction2; // la partie else étant facultative

Les valeurs de vérités en C++ (comme en C) sont de type entier. 0 indique FAUX et toute valeur différent de 0 est évaluée à VRAI.

Les opérateur relationnels sont :

==	*test d'égalité*	*if (x == 10) ...*
!=	*test de différence*	*if (x != 10) ...*
<	*inférieur*	*if (x < 0) ...*
>	*supérieur*	*if (x > 0) ...*

<=	*inférieur ou égal*	*if (x<= 10) ...*
>=	*supérieur ou égal*	*if (x >= 10) ...*

Les opérateur logiques sont:

&&	*le ET logique*	*if (x > 0 && x <= 10) ...*
||	*le OU logique*	*if (x == 0 || x == 1) ...*
!	*le NON logique*	*if (!x) ...*

On peut raccourcir certaines affectations comme :

x = x + 1;	*par*	*x++; ou bien ++x;*
x = x + exp;	*par*	*x += exp;*
*x = x * exp;*	*par*	*x *= exp;*

L'instruction du choix multiple (sélection de cas) a la forme :

```
switch (var)
{
        case val1: instruction_1; [break;]
        case val2: instruction_2; [break;]
        ...
        [default: instruction_n; ]
}
```

Suivant la valeur de *var* (*val1* ou *val2* ... ou autre *default*) on exécute l'instruction associée. La présence, optionnelle, de *break* à la fin d'une d'instruction, permet de sortir de l'instruction *switch* (branchement vers la fin). Son absence implique l'exécution des instructions suivantes.

Les boucles (itérations) peuvent être programmées à l'aide des instructions *for , while , do*:

```
for (initialisation-compteur; condition ; modification-compteur) instruction;
while (condition) instruction;
do instruction while (condition);
```

Par exemple pour calculer la somme des éléments d'un tableau *tab* d'entiers on peut écrire:

```
int i , somme = 0;
int tab[10] = { 2 , 4 , 8 , 1 , 13 , 32 , 4 , 2 , 0 , 5 };
for (i=0; i<10; i++) somme = somme + tab[i];
```

3-3) Fonctions

Un programme C++ est organisé en un ensemble de fonctions dont l'une est dite principale (la fonction *main*).

L'aspect général d'une définition de fonction en C++ est le suivant :

```
Type-retourné Nom-Fonction ( Type1 param1, Type2 param2 … )
{
        … Le corps de la fonction …
}
```

Si la fonction ne retourne pas de valeur (procédure) le type retourné sera alors 'void'. Par défaut, le type retourné d'une fonction est *int* (entier). L'exemple suivant montre la définition d'une fonction calculant le maximum de trois entiers :

```
int max3 (int a, int b, int c)
{
        int x;
        if (a > b) x = a; else x = b;
        if (x > c) return x; else return c;
}
```

L'instruction 'return expression;' termine l'exécution de la fonction et renvoie comme résultat la valeur de 'expression'.

Par défaut, le passage de paramètre lors d'un appel de fonction se fait **par valeur**, c'est à dire que chaque paramètre d'appel est une copie locale du paramètre effectif correspondant dans l'appel (c'est donc des paramètres d'entrée).

Le type **référence** (type&) permet entre autre le passage **par variable** en indiquant qu'un paramètre d'appel est un 'alias' du paramètre effectif correspondant. De ce fait une même variable aura deux noms distincts et c'est ce qui permet d'avoir des paramètres en sortie.

La procédure suivante permute ces deux arguments :

```
void permutation ( int& a, int& b )
{
        int temp = a;
        a = b;
        b = temp;
}
```

Notez que a et b sont des références vers des variables de type *int*. Lors d'un appel à cette procédure : *permutation(x, y);* les noms *a* et *x* désignent une même et unique variable entière et les noms *b* et *y* désignent aussi une même et unique variable. Ainsi toute affectation à la

variable *a* (*a = exp*) est une affectation à *x* (*x = exp*) et vice versa, et toute affectation à *b* est une affectation à *y*.

3-3) les entrées/sorties (E/S)

Le langage C++ ne défini pas en standard des primitives d'E/S. On doit donc utiliser l'une des librairies offertes ou alors en créer une.

'iostream.h' est l'une des librairies standards orientée vers la manipulation des flux d'E/S. Elle définie entre autre une primitive de sortie '*cout*' et une autre d'entrée '*cin*'. Grâce aux opérateurs de redirection << et >> on peut contrôler le flux d'E /S.

Pour afficher le contenu de la variable *x* on écrira : *cout << x ;* Pour lire une valeur dans *x* on écrira *cin >> x ;*

Voici un programme qui affiche la plus grande valeur entre 3 données lues :

```
#include<iostream.h>
int max3 (int a, int b, int c)
{
        int x;
        if (a > b) x = a; else x = b;
        if (x > c) return x; else return c;
}
void main( )
{
        int x,y,z ;
        cout << "Donner la 1ere valeur  ";
        cin >> x;
        cout << "Donner la 2e valeur ";
        cin >> y;
        cout << "Donner la 3e valeur ";
        cin >> z;
        cout << "Le maximum est " << max3(x,y,z);
}
```

La dernière ligne montre comment on peut utiliser l'opérateur << plusieurs fois dans une primitive '*cout*'.

3-4) Surcharge de fonctions et d'opérateurs

L'une des caractéristiques importante de C++ et de pouvoir redéfinir les fonctions et les opérateurs afin de les adapter et les rendre applicables à de nouveaux type d'arguments.

Lorsqu'on déclare plusieurs fois la même fonction (un même nom) avec des arguments différents (en type et/ou en nombre), on dit qu'il y a surcharge ou redéfinition de la fonction. Lorsque ce nom de fonction est utilisé dans un appel, la fonction choisie par le compilateur sera celle dont la signature (le type retourné, et les types des arguments) concorde avec l'instruction d'appel.

De même pour les opérateurs déjà définis dans le langage (comme = + - * / ++ -- [] ...), la plus part d'entre eux peuvent être surchargés afin de tenir compte des nouveaux types utilisateurs.

Voici quelques exemples de surcharge de fonctions :

```
#include <stdio.h>     // pour utiliser sprintf et sscanf
#include <string.h>    // pour utiliser strcat, strcpy, strlen

// concaténation de 2 entiers : concat(12,345) = 12345
long int concat(int x, int y)
{
        long int z;
        char ch[15];
        sprintf(ch,"%d%d",x,y);         // ecriture de x suivi de y dans une chaine de car
        sscanf(ch,"%ld",&z);            // lecture d'un entier long à partir de la chaine
        return z;
}

// concaténation de 2 chaine de car : concat ("hello ","world") = "hello world"
char *concat(char *str1, char *str2)
{
        char *result = new char[strlen(str1)+strlen(str2)+1];
        strcpy(result,str1);
        strcat(result,str2);
        return result;
}

// fonction donnant le maximum de 2 entiers
int maximum(int x, int y) { return (x>y? x : y); }      // si x > y alors x sinon y

// fonction donnant le maximum de 3 entiers
int maximum(int x, int y, int z)
{
        int temp;
        if (x > y) temp = x; else temp = y;
        if (temp > z) return temp; else return z;
}

// fonction donnant le maximum d'un tableau de 10 entiers
```

```
int maximum(int tab[10])
{
        int m = tab[0];
        for (int i=1; i<10; i++)
                if (tab[i] > m) m = tab[i];
        return m;
}
```

Des exemples de surcharge d'opérateurs seront donnés plus loin.

4) LIENS SEMANTIQUES

On regroupe sous cette notion les différents liens pouvant exister entre les objets au cours d'une application: l'instanciation, l'héritage, la référence et la composition.

4-1) L'instanciation (Lien : 'instance _de')

Dans le monde réel, les objets sont classés en catégories suivant leurs caractéristiques (propriétés). Par exemple toutes les 'tasses de café' peuvent être caractérisée par une couleur, une quantité de boisson et une température, de même que toutes les tasses peuvent être remplies, vidées et lavées.

Une **classe** d'objets est une catégorie représentant un **type d'objets** et qui sert de moule pour la création (ou l'**instanciation**) de nouveaux objets. Dans la figure IV-1, les objets Tasse1, Tasse2 et Tasse3 sont des **instances de** la classe 'Tasse de Café'

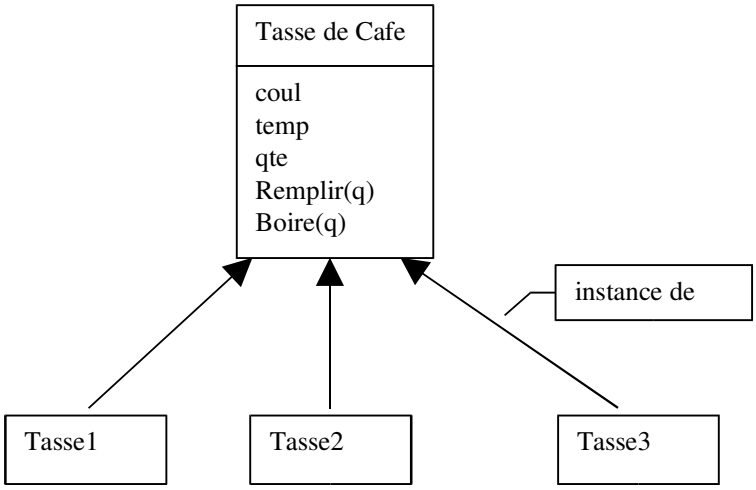


Fig IV-1 : Exemple de liens d'instanciation

Tous les objets ayant une même structure et même comportement appartiennent à la même classe (ils sont de même type).

Ainsi avant de commencer à créer des objets, il est nécessaire de définir d'abord leurs classes d'appartenances. La définition d'une classe consiste à lui donner un nom unique et à définir les attributs et méthodes formant ses propriétés.

En C++ on pourra définir une classe à l'aide de la déclaration :

class nom_classe { declaration d'attribut et de methodes };

Par exemple, on pourra définir la classe *tasse_cafe* de la manière suivante :

```
// fichier librairie "exemple1.h"
//
class tasse_cafe
{
private:
        char couleur[10];       // chaine de 10 caractères pour la propriété couleur
        int qte;                        // quantité,
        float temp;             // et température de la boisson
public:
        void remplir(int q);    // méthode remplir(...)
        void boire(int q);      // méthode boire(...)
};
```

Les mots *private* et *public* désignent respectivement la partie privée (en général les attributs) et la partie visible (en général les méthodes) des objets de type tasse_cafe. En suivant cette approche de cacher les attributs et laisser visible les méthodes, on peut considérer une classe comme étant un **type abstrait**. C'est à dire un type qui n'est défini, à l'extérieur, que par ses opérations.

Le corps des méthode peut être défini à l'intérieur ou à l'extérieur de la classe. Dans ce dernier cas on utilisera l'opérateur de résolution de portée '*::*'

```
// suite du fichier librairie "exemple1.h"
//
void tasse_cafe::remplir(int q)         // le corps de méthodes est défini à l'extérieur de la
{                                       // classe. remarquer l'utilisation de l'opérateur ::
        qte = qte + q;
}
void tasse_cafe::boire(int q)
{
        qte = qte - q;
}
```

L'instanciation en C++ peut être réalisée de deux façons : à l'aide de déclarations de variables statiques ou bien à l'aide de l'opérateur d'allocation de variable dynamique *new* (objets dynamiques).

Par exemple en utilisant la déclaration de la classe précédente tasse_cafe dans le fichier "exemple1.h", on peut instancier des objets et les manipuler. On utilise le nom de la classe comme un nom de type. Ainsi une déclaration du genre : *tasse_cafe tasse1;* est en fait une instanciation d'un nouvel objet de cette classe, *tasse1* représente alors l'identifiant de l'objet créé. Par contre une déclaration de la forme : *tasse_cafe *p;* ne représente pas une instanciation car *p* n'est qu'un pointeur pouvant contenir l'adresse d'un objet dynamique de cette classe. Pour créer un objet dynamique on utilise l'opérateur *new* :

```
#include "exemple1.h"          // inclusion du fichier où est définie la classe tasse_cafe
void main()
{
        tasse_cafe tasse1;     // tasse1 est un objet qui vient d'être créé
        tasse_cafe *p;         // p est un pointeur
        p = new tasse_cafe;    // p pointe un nouvel objet dynamique
        tasse1.remplir(10);    // envoie du message remplir(10) à l'objet tasse1
        p->remplir(20);        // envoie du message remplir(20) à l'objet pointé par p
        tasse1.boire(5);       // envoie du message boire(5) à l'objet tasse1
        delete p;              // destruction de l'objet pointé par p
}
```

Dans C++ et par défaut, quand on déclare une variable dans un bloc *{...}* sa destruction sera prise en charge par le système dés que l'on sort du bloc. Par contre une variable dynamique doit être explicitement créée (par exemple à l'aide de l'opérateur *new*) par le programmeur et explicitement détruite aussi (par exemple à l'aide de l'opérateur *delete*) et ce indépendamment de l'endroit où a été créée la variable dynamique.

- Notion de constructeur et destructeur

Dans l'exemple précédent, la déclaration : *tasse_cafe tasse1;* permet de créer un nouvel objet (*tasse1*). L'espace mémoire qui lui est nécessaire a été alloué par une méthode spéciale qu'on appèle **constructeur**. Cette méthode porte toujours le même nom que sa classe (*tasse_cafe*) et est définie automatiquement par le compilateur (c'est le constructeur par défaut).

Le programmeur peut s'il le désir étendre le constructeur par défaut, en déclarant un ou plusieurs constructeurs pour sa classe, afin de réaliser certaines tâches lors de la création de nouveaux objets (par exemple initialisation des attributs par des valeurs données). Les différents constructeurs déclarés par le programmeur portent tous le même nom (le nom de la classe) mais diffèrent par la liste des arguments. Un constructeur ne retourne jamais une valeur. Aucun type ne doit être spécifié comme résultat.

Dans l'exemple précédent, on aurait aimé initialiser la variable *qte* de la classe *tasse_cafe* à la valeur 0 à chaque création d'objet. Ca sera le rôle de notre nouveau constructeur :

```
class  tasse_cafe
{
private:
        char couleur[10];       // chaine de 10 caractères pour la propriété couleur
        int qte;                        // quantité,
        float temp;             // et température de la boisson
public:
        tasse_cafe();           // notre constructeur, remarquer l'absence de type
        void remplir(int q);    // méthode remplir(...)
        void boire(int q);      // méthode boire(...)
};
tasse_cafe::tasse_cafe() { qte = 0; }  // notre constructeur initialise qte à 0
void tasse_cafe::remplir(int q) { qte = qte + q; }
void tasse_cafe::boire(int q) { qte = qte - q; }
```

Maintenant, après l'exécution du code suivant :

```
tasse_cafe  tasse1;     // création de l'objet tasse1 et initialisation de qte à 0
tasse1.remplir(10);     // qte = 10  (qte = 0 + 10)
```

on est sûre que la quantité de café dans l'objet *tasse1* est exactement de 10 unités.

Voici un autre exemple pour voir l'utilisation des constructeurs :

```
class date              // classe représentant le type date
{
private:
        int jour, mois, annee;
public:
        // ... un certain nombre de méthodes de manipulation de dates
        date(int j, int m, int a);          // un constructeur à 3 arguments
};
date::date(int j, int m, int a) { jour = j; mois = m; annee = a; }
```

La classe *date* ci-dessus, a été déclarée avec un constructeur à trois arguments, donc à chaque instanciation d'un nouvel objet, il est nécessaire de l'initialiser avec exactement trois arguments:

```
date  d1 = date(11,12,1998);            // forme standard
date independence_day(5,7,1962);    // forme abrégée mais correcte
date d2;                                // Erreur, le constructeur attend 3 arguments
```

On aurait pu déclarer plusieurs constructeurs pour accepter plusieurs types d'initialisations comme le montre l'exemple suivant, où on déclare trois type de constructeurs représentant trois type d'initialisations :

```
#include <stdio.h>     // pour utiliser sscanf : lecture formater à partir d'une chaine
#include <string.h>    // pour utiliser strcmp : comparaison de chaine
class date
{
private:
        int jour, mois, annee;
public:
        // ... un certain nombre de méthodes de manipulation de date
        date(int j, int m, int a);          // un constructeur à 3 arguments
        date(char *ch);                     // un constructeur à 1 arguments de type chaîne
        date();                             // un constructeur sans arguments
};

date::date(int j, int m, int a)             // premier constructeur
{ jour = j; mois = m; annee = a; }

date::date(char *ch)                        // deuxième constructeur
{
        int j,a,i,continu;
        char m[4];
        char months[12][4] = { "jan","feb","mar","apr","may","jun","jul","aug",
                               "sep","oct","nov","dec" };
        sscanf(ch," %d %s %d",&j,m,&a);  // decouper la chaine ch en 3 champs
        i = 0; continu = 1;
        while (i < 12 && continu) {
                continu = strcmp(m,month[i]);         // retourne 0 si m = month[i]
                i++;
        }
        jour = j; mois = i; annee = a;
}

date::date()                                // troisième constructeur
{ jour = 1; mois = 1; annee = 2000; }
```

Maintenant les trois instanciations suivantes sont correctes :

```
date  uneDate(24,10,1969);
date  uneFete("1 may 1999");
date  grand_Bug;
```

On peut aussi instancier un objet par copie d'un autre objet, on utilise pour cela soit le constructeur de copie par défaut qui recopie les valeurs de tous les attributs, soit notre propre constructeur de copie afin de l'adapter aux caractéristiques de la classe (surtout dans le cas où

certain attributs sont de type pointeur). Le constructeur de copie d'une classe C peut être défini par : *X::X(const X&)*.

A l'aide du constructeur de copie on peut écrire :

Type nouvel_obj = ancien_obj;

Ainsi lors de la création de '*nouvel_obj'* les valeurs d'attributs de '*ancien_obj'* seront recopiées dans le nouvel objet.

Quand un objet cesse d'exister (sortie du bloc où il a été déclaré ou destruction explicite par *delete*), une méthode particulière appelée **destructeur** est automatiquement exécutée. Le destructeur par défaut permet de libérer l'espace occupé par l'objet. Le programmeur pourra aussi déclarer son propre destructeur pour réaliser, en plus, certaine tâches particulières. Le nom d'un destructeur est toujours le nom de sa classe précédé par le caractère : '~' (par exemple *~tasse_cafe()* pour la classe *tasse_cafe* ou *~date()* pour la classe *date*). Un destructeur comme les constructeur, ne retourne jamais de type et ne peut pas avoir des paramètres.

Afin de mieux illustrer les rôles des constructeurs et destructeurs, on donne ci-dessous un exemple de classe représentant le type abstrait 'chaîne de caractères' :

- Exemple2 (la classe *chaine*)

Une chaîne de caractère sera définie par la classe *chaine* qui encapsule (regroupe) les attributs et méthodes suivantes:

- un pointeur (*char *p*) vers la zone mémoire contenant la chaîne de caractères,
- un entier (*int taille*) indiquant la taille de la zone mémoire réservée à la chaîne,
- une méthode (*ecrire_ch*) pour affecter une chaîne à un objet,
- une méthode (*lire_ch*) pour récupérer la chaîne stockée dans un objet,
- une méthode (*lire_taille*) pour récupérer la taille de la zone mémoire réservée,
- un constructeurs (*chaine()*) permettant d'allouer une zone pour 256 caractères,
- un autre constructeur (*chaine(char *c)*) permettant d'initialiser l'attribut *p* à la chaîne *c* et de lui allouer la zone mémoire nécessaire,
- un constructeur de copie (*chaine (const chaine& obj_copie)*) permettant d'initialiser un nouvel objet par recopie d'un ancien objet,

- un destructeur (*~chaine()*) se chargeant de libérer la zone allouée à la chaîne avant de détruire l'objet, et
- l'opérateur = surchargé, afin de pouvoir affecter un objet de type *chaine* à un autre, hors initialisation.

Les attributs seront cachés (partie privée) et les méthodes représentent l'interface avec le monde extérieur (la partie publique).

```
// "exemple2.h"
//
#include<string.h>              // pour l'utilisation de strncpy
classe chaine  {
private: char *p;               // la chaine de caractères
         int taille;            // la taille de la zone réservée
public:
        void ecrire_ch(char *c);                // affecter un nouvelle chaine à l'objet
        char *lire_ch() { return p; }           // consulter la chaine stockée
        int lire_taille() { return taille; }    // consulter la taille de zone réservée
        chaine();                               // constructeur sans paramètres
        chaine(char *c);                        // constructeur à un paramètre
        chaine(const chaine& obj_copie);        // constructeur de copie
        ~chaine() { delete p; }                         // destructeur
        chaine& operator=(const chaine& obj_affecte);   // surcharge de =
};

void chaine::ecrire_ch(char *c)
{ strncpy(p,c,taille-1); p[taille-1] = '\0'; }  // copie de taille -1 car et on termine par \0

chaine::chaine() {                              // constructeur sans paramètres
        taille = 256;                           // taille par défaut = 256 car
        p = new char[taille];                   // allocation d'une zone mémoire de 256 car
}

chaine::chaine(char *c) {                       // constructeur à 1 paramètre
        taille = strlen(c) + 1;                 // strlen retourne le nb de car de c sans le \0
        p = new char[taille];                   // allocation mémoire pour la chaine de car
        strcpy(p,c);                            // copie de la chaine c dans p
}
chaine::chaine(const chaine& obj_copie)         // constructeur de copie
{
        taille = obj_copie.taille;              // copie de la taille
        p = new char[taille];                   // allocation mémoire pour la chaine de car
        strcpy(p,obj_copie.p);                  // copie de la chaine depuis obj_copie
}
```

```
chaine& chaine::operator=(const chaine& obj_affecte)  // l'opérateur =
{
        if (this != &obj_affecte) {             // si ce n'est pas :  c = c;
                delete p;                       // libérer l'ancienne zone mémoire
                taille = obj_affecte.taille;    // affecter la nouvelle taille
                p = new char[taille];           // allouer un nouvelle zone
                strcpy(p, obj_affecte.p);       // copier la nouvelle chaine
        }
        return *this;                           // retourner l'objet receveur
}
```

Remarques concernant l'exemple précédent:

- Dans le corps de la méthode *operator=(...)* la variable *this* représente un pointeur vers l'objet recevant le message. C'est l'équivalent du *Self* de Smalltalk. Ainsi, si *c1* et *c2* sont des objets de la classe *chaine* alors l'instruction : *c1 = c2;* est considérée comme étant un envoie du message *operator=(c2)* à l'objet *c1* (ou d'une autre façon comme étant un appel de méthode *c1.operator=(c2);*).

- Certains corps de méthodes ont été définis à l'intérieur de la déclaration de la classe (*lire_ch*, *lire_taille* et *~chaine*). C'est juste pour des raisons de performance, quand le corps d'une méthode est très petit, l'exécution d'un appel de méthode sera plus rapide.

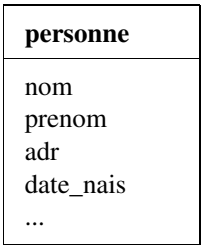
- La différence entre le constructeur à un paramètre *chaine(char *p)* et le constructeur de copie *chaine(const chaine& obj_copie)* est que le premier sera utilisé pour instancier un nouvel objet *chaine* à l'aide d'une chaine de caractères alors que le deuxième le fera à l'aide d'un autre objet *chaine* en procédant par copie des ses attributs:

chaine ch1("Salut"); ou bien *chaine ch1=chaine("Salut");* *// constructeur à 1 paramètre*
chaine ch2 = ch1; *// constructeur de copie*

4-2) L'héritage (Lien : 'est-un')

Lorsqu'on déclare une nouvelle classe C, on a la possibilité de spécifier que cette nouvelle classe doit hériter des propriétés d'une ou de plusieurs classes mères. Ainsi les nouvelles propriétés définies, seront rajoutées à celles héritées pour former un nouveau concept plus spécifique que ceux déjà définis dans les classes mères. La classe C est alors une **spécialisation** de ses classes mères. Ces dernières sont appelées les **super-classes** de C.

Soit la classe *personne* décrivant les "objets" de type personne:



La figure IV-2 montre un exemple ou la classe *employe* hérite de la classe *personne* car tout employé est d'abord une personne, et donc a comme propriétés un nom, un prénom, une adresse, etc... De plus, un employé est aussi caractérisé par d'autres propriétés propres aux employés (comme le NSS, un salaire, une fonction, etc ...). Les employés forment alors une sous-classe de l'ensemble des personnes. De même la classe *etudiant* hérite aussi de la classe *personne* pour les mêmes raisons. La classe *stagiaire* est une sous-classe de *employe* et de *etudiant* (héritage multiple). Donc un objet de type *stagiaire* et en même temps un *employe* et un *etudiant*.

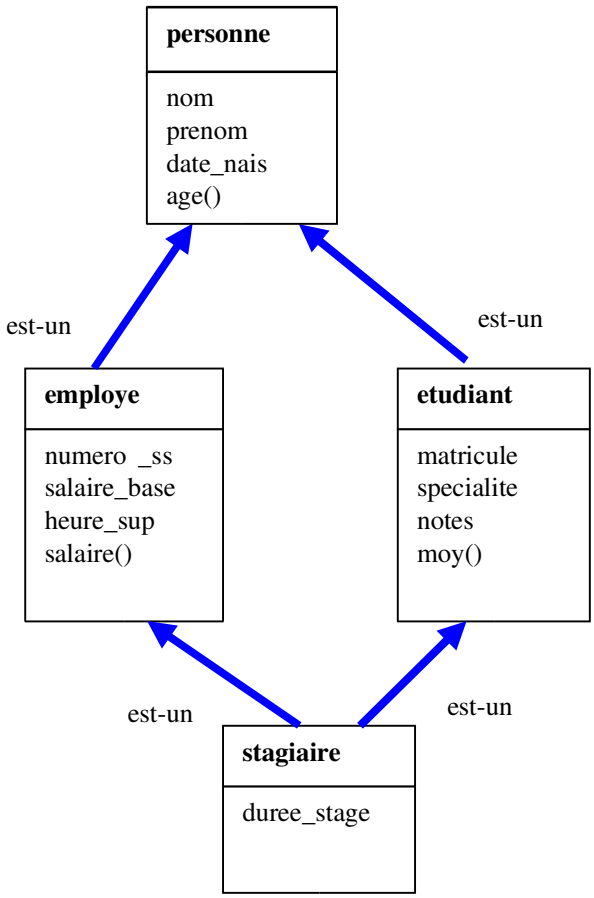


Fig IV-2 : un exemple de graphe d'héritage

Chaque objet de type *employe* aura comme propriétés ceux de sa classe (*employe*) plus ceux de ses super-classes (*personne, ...*)

En C++ on peut déclarer la classe *employe* de la manière suivante :

```
class employe : public personne  { // personne est une super-classe de employe
private:
        int numero_ss;                                          // les attributs locaux
        float salaire_base;
        int heure_sup;
public:
        int lire_nss() { return numero_ss; }                    // les méthodes locales
        void ecrire_nss(int n) { numero_ss = n; }
        ...
        float salaire() { return salaire_base + (500 * heure_sup); }
};
```

Lors de la conception d'une application, il est très important de bien structurer son graphe d'héritage. Il est judicieux de définir d'abord les classes générales, même si elle ne représentent pas forcément un concept utilisé dans l'application. Ensuite à partir de ses classes de bases, on dérive à l'aide du lien d'héritage des classes de plus en plus spécifiques (spécialisées) et adaptées aux besoins de l'application. Cette mise en facteur des propriétés communes dans les classes générales, facilitera beaucoup l'extension de votre application vers de nouvelles exigences.

Voici quelques exemples de graphes d'héritages:

1) La classification des animaux en catégories (les mammifères, les oiseaux, les reptiles,...) à leur tour les mammifères se divisent en carnivore, herbivore, omnivore, etc ...

2) Les moyens de locomotion peuvent être classés en véhicules terrestres, marins et aériens. La classe des véhicules amphibies héritera des classes 'véhicule terrestres' et 'véhicules marins' en même temps.

3) En médecine, on peut classer les maladies suivant leur type : infectieuses, héréditaires, cardio-vasculaire,...

4-3) Les objets complexes (Lien : 'référence-à' et 'composé-de')

Quand on veut représenter un objet du monde réel, on est souvent confronté au problème de la complexité du concept réel, car beaucoup d'objets courants, sont inter-dépendants et/ou composés les uns par rapport aux autres. Par exemple un objet *voiture* est composé d'un grand nombre d'objets d'autres types comme un *moteur*, des *roues*, une *carrosserie*, un *système de*

freinage, une *boite à vitesse*, etc ... Chacun de ces objets (par exemple le *moteur*) est lui même composé d'une multitude d'autres sous-objets (des *pistons*, des *soupapes*, des *joints*, des *boulons*, ...). Ce type d'objet aussi complexe soit-il est manipulé de la même manière qu'un objet simple (comme en entier ou une chaîne de caractères).

Pour pouvoir modéliser des objets complexes, on a la possibilité, lors de la déclaration d'une classe, de définir des attributs ayant comme type une classe déjà définie. Cette manière de procéder permet d'établir des associations (liens de référence ou de composition) entre plusieurs classes.

Un lien de composition peut exister par exemple entre la classe *voiture* et la classe *moteur*. Alors qu'entre cette même classe *voiture* et la classe *personne* il y a un lien de référence, pour indique qu'une voiture est conduite par une personne (voir figure IV-3).

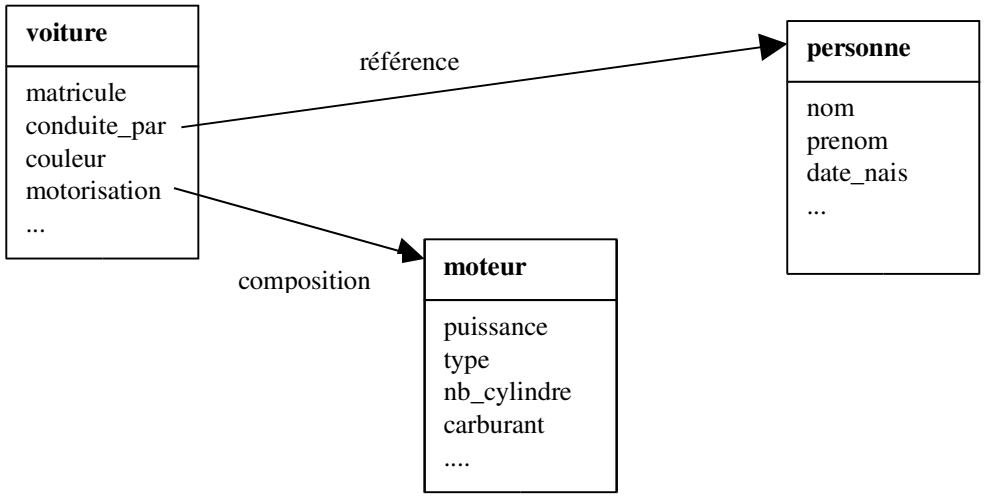


Fig IV-3 : Exemple d'objet complexe

Un lien de composition est un lien de référence sémantiquement plus riche. Par exemple si *v* est un objet de la classe *voiture*, composé du *moteur m* et conduite par la *personne p*. L'existence de l'objet *m* dépend de l'objet *v*, car c'est un sous-objet ou encore un objet composant. Par contre l'objet *p* ne fait pas partie de l'objet *v*, il ne fait pas partie de ses objets composants. Donc si on décide de détruire l'objet *v* (par exemple la voiture en question ne fait plus partie du parc automobile de l'entreprise) tous ses composants doivent aussi être détruits (son moteur *m*) de la base de données, par contre son conducteur *p* reste non affecté par cette opération (il conduira un autre véhicule par exemple).

En C++ il n'y a pas de différence lors de la déclaration d'une classe entre un attribut de type référence et un attribut de composition. C'est au programmeur de coder la sémantique de la composition, généralement au niveau des constructeurs et destructeurs de la classe.

Voici un autre exemple montrant l'utilisation du lien de référence:

```
class personne
{
private:
        char nom[20];
        char prenom[20];
        char situation_fam;             // 'm','d' ou'c' pour mariée, divorcée ou célibataire
        personne *conjoint;             // lien de référence utilisant un pointeur
public:
        char *lire_nom() { return nom; }
        char *lire_prenom() { return prenom; }
        char lire_situation() { return situatio_fam; }
        personne *lire_conjoint() { return conjoint; }
        void ecrire_nom(char *ch) { strncpy(nom,ch,19); nom[19] = \0'; }
        void ecrire_prenom(char *ch) { strncpy(prenom,ch,19); prenom[19] = \0'; }
        void ecrire_situation(char s) { situation_fam = s; }
        void ecrire_conjoint(personne *p) { conjoint = p; }
};

void main()
{
        personne p1,p2;
        ...
        // si on veut connaitre le nom du conjoint de p1 on écrira:
        if (p1.lire_situation() == 'm')         // si la personne est mariée
                cout << "Nom du conjoint = " << p1.lire_conjoint()->nom << endl;
        ...
}
```

5) POLYMORPHISME ET METHODES VIRTUELLES

On a vu dans le chapitre précédent que lorsqu'une classe *C1* hérite d'une autre classe *C2*, cette dernière lui transmet toutes ses propriétés (attributs et méthodes). *C1* pourra alors rajouter de nouvelles propriétés pour spécialiser la définition de sa classe mère. En réalité, *C1* peut rajouter de nouveaux attributs de nouvelles méthodes et/ou redéfinir une ou plusieurs méthodes héritées afin de tenir compte des nouveaux attributs de *C1*. C'est la notion de surcharge de fonction utilisée dans C++ adaptée aux méthodes d'une classe.

Les méthodes redéfinie doivent avoir les mêmes signatures (même déclaration d'entête), les seuls changements concernent les corps des méthodes. C'est la notion de méthodes **polymorphique**, c'est à dire un même message pouvant être envoyé à plusieurs types d'objets différents, chacun le traitant à sa manière.

Le polymorphisme existe dans beaucoup de langages de programmation sous une forme ou une autre. Par exemple l'opération '+' permet d'additionner des entiers comme des réels ou alors réaliser la concaténation de deux chaînes de caractères, tout dépend de la nature de ses arguments. C'est donc une opération polymorphique mais restreinte à certains types prédéfinis. Les langages orientés objets permettent de généraliser le concept d'opération polymorphique en l'étendant aux types utilisateurs (nouvelles classes).

Considérons l'exemple d'une hiérarchie de classes représentant les formes géométriques sur le plan:

```
#include <iostream.h>                 // pour les E/S cin et cout
#include <math.h>                     // pour la fonction sqrt(...)
#include <stdio.h>                    // pour la fonction getchar()

const double PI = 3.14159265358979;

class point             // classe représentant un point du plan
{
        double x,y;
public:
        point()                                 // constructeur sans paramètres
                { x = 0.0; y = 0.0 }
        point(double a, double b)// constructeur à 2 param
        { x = a; y = b; }
        double abs()                    // abscisse du point
                { return x; }
```

```
        double ord()                    // son ordonnée
                { return y; }
        void aff_abs(double a)          // modifier l'abscisse
                { x = a;}
        void aff_ord(double b)          // modifier l'ordonnée
                { y = b;}
        void allumer()                  // afficher un point
                { cout << "(" << x << "," << y<< ")"; }
        double distance(point p)        // distance avec le point p
        {return sqrt((x-p.abs())*(x-p.abs()) +
                (y-p.ord())*(y-p.ord())); }
};
```

Dans la classe *point* la méthode *allumer()* affiche les coordonnées à l'écran. La méthode *distance(point p)* calcule la distance entre le point courant (recevant le message) et le point p (*sqrt()* est la fonction racine carrée).

```
class forme             // classe abstraite représentant une
{                               // forme géométrique sur le plan
        point ref;      // le point de référence ou l'origine
public:
        forme(point p)          // constructeur à un param
        { ref.aff_abs(p.abs());ref.aff_ord(p.ord()); }
        point origine()
                { return ref; }
        double virtual surface()= 0;   // méthode virtuelle
        double virtual perimetre()= 0;// méthode virtuelle
        void virtual dessiner() = 0;    // méthode virtuelle
        void translater(point p)                // translation de la forme
        { ref.aff_abs(ref.abs() + p.abs());
       ref.aff_ord(ref.ord() + p.ord()); }
};
```

Dans la classe *forme* l'attribut *ref* représente le point de référence de la forme géométrique, il joue le rôle d'origine pour les éventuels autres point de la figure. Les méthodes *surface()*, *perimetre()* et *dessiner()* sont abstraites (*=0*), elles ne seront définies que dans les sous-classe de *forme*. Elles sont déclarées *virtual* pour permettre la liaison dynamique et donc le polymorphisme. La méthode *translater(..)* permet de déplacer la forme vers un autre emplacement dans le plan (*ref*+*p*).

```
class rectangle : public forme          // rectangle hérite de
{                                                       // forme
        point sd;               // coin sup droit relatif à l'origine
```

```
public:
        rectangle(point inf_g, point sup_d): forme(inf_g)
        { sd.aff_abs(sup_d.abs()-inf_g.abs());
                sd.aff_ord(sup_d.ord()-inf_g.ord()); }
        point coin_inf_g()      // coin inf gauche absolu
                { return origine(); }
        point coin_sup_d()      // coin sup droit absolu
                { return point(origine().abs()+sd.abs(),
                                        origine().ord()+sd.ord()); }
        double surface();       // méthode virtuelle définie
        double perimetre();     // méthode virtuelle définie
        void dessiner()                 // méthode virtuelle définie
                { cout << "rect : "; coin_inf_g().allumer();
                cout << " : "; coin_sup_d().allumer();
                cout << endl; }
};
```

La classe *rectangle* hérite de la classe *forme* l'attribut *ref* et ajoute un autre attribut *sd* de type *point*. *ref* désigne le coin inférieur gauche du rectangle et *sd* le déplacement relatif par rapport à *ref* pour atteindre le coins supérieur droit du *rectangle*.

Les méthodes rajoutées sont celles donnant le coin inférieur gauche, qui n'est autre que le point *ref* et le coin supérieur droit (*ref* + *sd*. Les méthodes (virtuelles) *surface(), perimetre()* et *dessiner()* sont redéfinies. La méthode *translater* est héritée sans changement.

Remarquer que le constructeur *rectangle(inf_g,sup_d)* appele en premier le constructeur *frome(inf_g)* pour initialiser le point de référence *ref* à *inf_g*, ensuite il initialise coin supérieur droit relatif (*sd*) à *inf_d - inf_g*. (Fig V-1)

```
double rectangle::surface()
{
        point p = point(coin_inf_g().abs(),coin_sup_d().ord());
        return (        coin_inf_g().distance(p) *
                        p.distance(coin_sup_d()) );
}
double rectangle::perimetre()
{
        point p(coin_inf_g().abs(),coin_sup_d().ord());
        return (        2*coin_inf_g().distance(p) +
                        2*p.distance(coin_sup_d()) );
}
```

La classe cercle est définie par un centre, le point de référence (*ref*) et un rayon (*ray).* (Fig V-1)

```
class cercle : public forme  {  // cercle hérite aussi de forme
        double ray;     // rayon du cercle
public: cercle(point centre, double rayon) : forme(centre)  { ray = rayon; }
        point centre() { return origine(); }
        double rayon() { return ray; }
        double surface()                // méthode virtuelle définie
                { return PI * ray * ray; }
        double perimetre()              // méthode virtuelle définie
                { return 2 * PI * ray; }
        void dessiner()  {              // méthode virtuelle définie
                cout << "cercle : "; centre().allumer();
                cout << " : " << rayon() << endl;
                }
};

void main()  {
        forme *f;       // pointeur vers forme, rectangle ou cercle
        f = new rectangle(point(0.0,0.0),point(10.0,5.0));
//
// cette ecriture montre comment utiliser un constructeur
// pour créer (temporairement) les objets : point(0.0,0.0)
// et point(10.0 , 5.0).
//
        f->dessiner();
        cout << "sa surface : " << f->surface() << endl;
        cout << "son perimetre : " << f->perimetre() << endl;
//
// tous ces appels précédent f->dessiner(), f->surface() et
// f->perimetre() se font par liaison dynamique.
//
        f->translater(point(10.0,25.0));
        cout << "translation du rect de (10.0,25.0)" << endl;
        f->dessiner();
        cout << "sa surface : " << f->surface() << endl;
        cout << "son perimetre : " << f->perimetre() << endl;
        delete f;       // destruction de l'objet pointé par f
        f = new cercle(point(0.0,0.0),5.0);
//
// création d'un nouvel objet de type cercle pointé par f.
//
        f->dessiner();
        cout << "sa surface : " << f->surface() << endl;
        cout << "son perimetre : " << f->perimetre() << endl;
        f->translater(point(-25.0,20.0));
        cout << "translation du rect de (-25.0,+20.0)" << endl;
        f->dessiner();
        cout << "sa surface : " << f->surface() << endl;
        cout << "son perimetre : " << f->perimetre() << endl;
        while (getchar() != '\n');
}
```

Voici le résultat de son exécution :

```
rect : (0,0) : (10,5)
sa surface : 50
son perimetre : 30
translation du rect de (10.0 , 25.0)
rect : (10,25) : (20,30)
sa surface : 50
son perimetre : 30
cercle : (0,0) : 5
sa surface : 78.5398
son perimetre : 31.4159
translation du rect de (-25.0 , +20.0)
cercle : (-25,20) : 5
sa surface : 78.5398
son perimetre : 31.4159
```

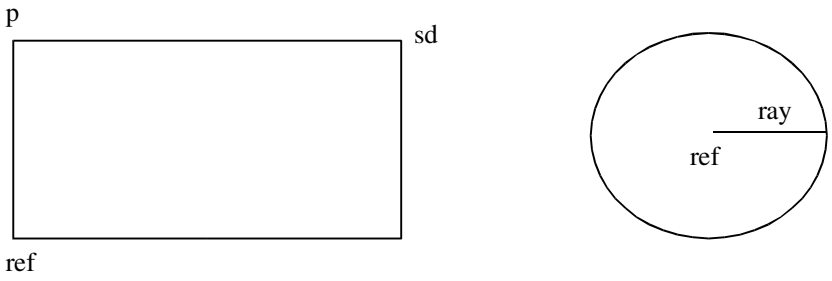


Fig V-1 : un objet rectangle et un objet cercle

6) COLLECTION D'OBJETS ET CLASSES GENERIQUES

Lorsqu'un attribut d'un objet regroupe plusieurs valeurs en même temps, on dit qu'il est multivalué. Par exemple un objet de type personne peut avoir comme attribut multivalué, le champs enfant qui désigne un ensemble d'objets de type personnes représentant les enfants du premier objet.

Un autre exemple serait un objet de type livre formé d'un ensemble d'objet de type chapitre et écrit par un ensemble d'auteurs (objet de type personne), ...

Les collections d'objet sont des classes particulières permettant de représenter et de manipuler des ensembles d'objets. Par exemple la classe ensemble, la classe liste, la classe tableau, etc...

Il existe trois type de collections:

a- Les collections où les objets contenus sont tous d'un même type connu à la compilation. Par exemple un véhicule est formé par un ensemble de roues. Chaque objet de l'ensemble est une roue. Dans ce cas, on pourra envoyer tous les messages définis dans la classe roue, à chaque objet contenu dans la collection.

b- Les collections où les objets sont de type hétérogène. Par exemple un parking contient un ensemble de véhicules pouvant être des voitures, des camions, des motos, etc ... Dans ce cas on ne pourra appeler que les méthodes virtuelle de la classe véhicule pour les objets de la collection.

c- Les collection où le type des objets ne sera connu qu'à l'exécution. Par exemple une file d'attente d'objet de type T où T est un paramètre. Dans ce cas on ne pourra utiliser que les méthodes propre à la collection en question.

Les traditionnelle structures de données sont utilisées pour décrire les collections.

Listes Linéaires Chaînées (LLC)

Une LLC est liste d'éléments appelés maillons, chacun renferme une valeur et pointe le prochain maillon. (Fig 6-1)

Ce type de structures de données est bien adapté au traitement séquentiel et lorsque le nombre d'éléments de la collection ne peut être prédit à l'avance.

Les opérations classiques sur les LLC sont: l'allocation et la destruction de maillons, l'insertion à une position donnée dans la liste, la recherche d'une valeur donnée.

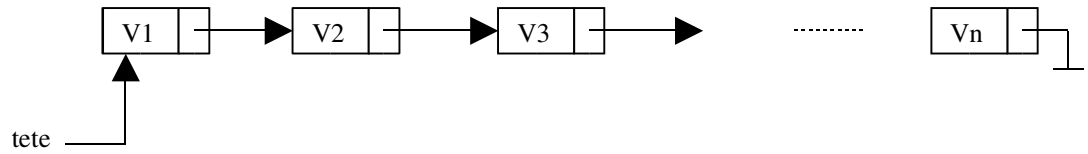


Fig VI-1 : Une Liste Linéaire Chaînée

Une liste est identifiée par sa tête (le pointeur vers le premier maillon). Le dernier maillon pointe 0 (ou NULL le pointeur vide).

Exemple d'utilisation d'une liste linéaire chaînée:

Soit la classe *Ecrivain* où un attribut *publications* donne la liste des *Ouvrages* ecrits par un auteur donné :

```
class Ecrivain : public personne              // Ecrivain hérite de personne
{
        // ... des attributs propores à un ecrivain ...
        Maillon_ouvrage *publications;        // pointeur vers la tête de liste des ouvrages
public:
        // ... des méthode propres à un ecrivain ...
        void Lister_titre();                  // permet de lister les titres des ouvrages
};

void Ecrivain::Lister_titre()
{
        Maillon_ouvrage *p = publications; // initialisation de p vers la tete de liste
        cout << "Liste des ouvrages :" << endl;
        while (p != NULL) {                   // tantque non fin de liste faire
                p->Valeur()->Afficher_titre();
                p = p->Suivant();
        }
        cout << " Fin de liste." << endl;
}

class Ouvrage  {                              // la classe Ouvrage
        // ... des attributs propres à un ouvrage ...
        char *titre;                          // chaine de car pour le titre
public:
        // ... des méthodes propres à un ouvrage ...
        void Afficher_titre();                // pour afficher le titre de l'ouvrage
};
```

```
void Ouvrage::Afficher_titre()
{
        cout << titre << endl;
}

class Maillon_ouvrage                   // maillon contenant un ouvrage
{
        Ouvrage *val;                           // val est pointeur vers un ouvrage donné
        Maillon_ouvrage *adr;           // adr pointe le maillon suivant
public:
        // ... les méthodes propres à cette classe
        Ouvrage *Valeur() { return val; }
        Maillon_ouvrage *Suivant() { return adr; }
};
```

Liste linéaire chaînée bidirectionnelle

C'est une liste linéaire chaînée où on rajoute un pointeur vers le maillon précédents, afin de pouvoir la parcourir dans les deux sens (Fig VI-2).

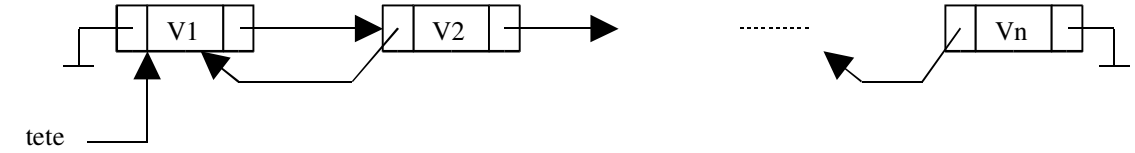


Fig VI-2 : Une LLC Bidirectionnelle

Chaque maillon est une structure à trois champs: une valeur et deux pointeurs. Dans l'exemple précédent on aurait déclaré la classe *Maillon_ouvrage* comme ceci :

```
class Maillon_ouvrage                   // maillon contenant un ouvrage
{
        Ouvrage *val;                           // val est pointeur vers un ouvrage donné
        Maillon_ouvrage *adrd;          // adrd pointe le maillon suivant
        Maillon_ouvrage *adrg;          // adrg pointe le maillon précédent
public:
        // ... les méthodes propres à cette classe
        Ouvrage *Valeur() { return val; }
        Maillon_ouvrage *Suivant() { return adrd; }
        Maillon_ouvrage *Precedent() { return adrg; }
};
```

Une liste bidirectionnelle est identifiée par un pointeur vers le premier maillon (tête de liste), mais on peut aussi rajouter un pointeur vers la queue de la liste pour faciliter les traitements en fin de liste.

Les Tableaux

Un simple tableau est idéal pour représenter une collection car il permet d'accéder directement ses éléments par contre sa taille doit être spécifiée au début des traitements (soit en statique soit en dynamique).

L'exemple suivant montre comment utiliser un tableau pour représenter un Sac d'objets Postaux pouvant être soit des lettres, des lettres recommandées, des colis, etc ...:

```
#include<stdio.h>
#include<stdlib.h>

class ObjPostal {                    // classe générale regroupant les objets postaux
 friend class SacPostal;             // la classe SacPostal peut acceder aux proprietés
        int poids;                   // privées (poids)
 public :
        int tarif;
        virtual void Affranchir() { tarif = 0; }
        int poidsObj() { return poids; }
        ObjPostal(int p) { poids = p; tarif = 0; }      // constructeur à un param (le poids)
        ObjPostal() { poids = 20; tarif = 0; }                  // constructeur sans param
};
```

L'attribut tarif a été déclaré public pour simplifier le code. La méthode Affranchir() est virtuelle car le tarif dépendra de la nature de l'objet postal (lettre normale, recommandé, par avion, ...).

```
class Colis : public ObjPostal {              // Colis est un ObjPostal ayant comme
        int volume;                           // propriété supplémentaire le volume
 public :
        void Affranchir() { tarif = 4 + 2*volume + 3*poidsObj(); }
        Colis(int p, int v) : ObjPostal(p)
                { volume = v; }
};
```

La méthode Affranchir() est définie dans ce cas comme étant la formule :

4 DA + 2**volume* +3**poidsObj().*

Remarquer que, contrairement à *volume*, l'attribut *poids* ne peut pas être directement manipuler par les méthodes de la class *Colis*. Pour qu'un attribut d'une classe C puisse être vu par les classe filles, il doit être déclaré, au niveau de C, soit *public* soit *protected.* En le déclarant *private*, il n'est visible qu'à l'intérieur de sa classe ou des classes et fonctions amies (*friend*).

```
class Lettre : public ObjPostal {               // Lettre hérite de ObjPostal
        int urgent;                             // urgent = 0 ou 1
 public :
         void Affranchir() { tarif = 2 + 2*poidsObj() + (urgent ? 3:0); }
         Lettre(int p, int u) : ObjPostal(p) { urgent = u; }
};

class ParAvion : public Lettre {                // ParAvion hérite de la classe Lettre
 public :
        void Affranchir()                       // appele Affranchir de sa classe mère
                { Lettre::Affranchir(); tarif += 7; }   // et rajoute 7 DA en plus
        ParAvion(int p, int u) : Lettre(p,u) { } // le constructeur appele Lettre(p,u)
};

class CourrierAdmin : public Lettre {           // une autre sous-classe de Lettre
 public :
        void Affranchir() { tarif = 0; }        // sans tarif
        CourrierAdmin (int p, int u) : Lettre(p,u) { }
};

class SacPostal {                       // SacPostal représente une collection d'objets
        int poids_vide;                         // de type ObetPostal (Lettre, Colis, Par
avion,...)
        int nbelts;             // Nb d'elements dans le sac
        int capacite;           // capacité maximale du sac
        ObjPostal** sac;        // sac est un pointeur vers pointeurs vers ObjPostal
public:
        void Affranchir_Sac();
        int Ajouter(ObjPostal* x)       // Inserer un obj dans la collection
        {
          if (nbelts == capacite) return 0;     // retourner 0 si depassement de capacité
          sac[nbelts++] = x;
          return nbelts;                        // sinon retourner le nb d'elements
        }
        int Enlever(ObjPostal *p)       // Recupéré dans p le dernier obj de la collection
        {
          if (nbelts == 0) return 0;            // retourner 0 si le sac est vide
          p = sac[--nbelts];
          return 1;                             // sinon retourner 1.
        }
        int poids();                    // poids total du sac
        int Valeur();                   // valeur totale du sac
        SacPostal(int, int);            // constructeur
        ~SacPostal() {  delete sac; }   // destructeur
        friend SacPostal operator+(SacPostal&, SacPostal&);     // concaténation de sac
};                                      // en redéfinissant l'opérateur +

void SacPostal::Affranchir_Sac()
{
   int i;
```

```
   for (i=0; i<nbelts; i++)              // cela revient à affranchir chaque objet du sac
         sac[i]->Affranchir();
}

int SacPostal::poids()
{
  int s = poids_vide;   // poids à vide + la somme des poids de tous les obj du sac
  for (int i=0; i<nbelts; i++) s += sac[i]->poidsObj();
  return s;
}

int SacPostal::Valeur()
{
  int s = 0;
  int i;
  for (i=0; i<nbelts; i++)      // somme des tarifs de tous les obj du sac
         s += sac[i]->tarif;
  return s;
}

SacPostal::SacPostal(int cap, int p)
{
  poids_vide = p;
  capacite = cap;
  nbelts = 0;
  sac = new ObjPostal* [cap];         // Allocation d'un tableau de cap éléments de type
}                                     // pointeur vers ObjPostal

SacPostal operator+(SacPostal& s1, SacPostal& s2)
{
                                      // création d'un nouveau Sac
  SacPostal sac(s1.capacite+s2.capacite, s1.poids_vide+s2.poids_vide);
  sac.nbelts = s1.nbelts+s2.nbelts;
  for (int i=0; i<s1.nbelts; i++)
         sac.sac[i] = s1.sac[i];
  for (int j=0; j<s2.nbelts; i++,j++)
         sac.sac[i] = s2.sac[j];
  return sac;
}


void main()
{
  char c;
  SacPostal Sac1(10,5), Sac2(8,3);
  ObjPostal* x;
  do {
         printf("------ M E N U -------\n\n");
         printf("1) Inserer un Colis dans Sac1\n");
         printf("2) Inserer une Lettre dans Sac1\n");
```

```
printf("3) Inserer un CourrierAdmin dans Sac1\n");
printf("4) Inserer un Colis dans Sac2\n");
printf("5) Inserer une Lettre dans Sac2\n");
printf("6) Inserer un CourrierAdmin dans Sac2\n");
printf("7) Voir Poids et Valeur de Sac1\n");
printf("8) Voir Poids et Valeur de Sac2\n");
printf("9) Faire Somme de Sac1 et Sac2\n");
printf("0) Fin.\n\n");
printf("\tVotre Choix : ");
c = getchar();
printf("\n\n");
switch (c) {
  case '1' :
        printf("Donner poids et volume : ");
        scanf(" %d %d",&p,&v);
        x = new Colis(p,v);
        Sac1.Ajouter(x);        // Ajouter au Sac1 un obj de type Colis
        break;
  case '2' :
        printf("Donner poids et urgent : ");
        scanf(" %d %d",&p,&v);
        x = new Lettre(p,v);
        Sac1.Ajouter(x);        // Ajouter au Sac1 un obj de type Lettre
        break;
  case '3' :
        printf("Donner poids et urgent : ");
        scanf(" %d %d",&p,&v);
        x = new CourrierAdmin(p,v);
        Sac1.Ajouter(x);        // Ajouter au Sac1 un obj de type CourrierAdmin
        break;
  case '4' :
        printf("Donner poids et volume : ");
        scanf(" %d %d",&p,&v);
        x = new Colis(p,v);
        Sac2.Ajouter(x);        // Ajouter au Sac2 un obj de type Colis
        break;
  case '5' :
        printf("Donner poids et urgent : ");
        scanf(" %d %d",&p,&v);
        x = new Lettre(p,v);
        Sac2.Ajouter(x);        // Ajouter au Sac2 un obj de type Lettre
        break;
  case '6' :
        printf("Donner poids et urgent : ");
        scanf(" %d %d",&p,&v);
        x = new CourrierAdmin(p,v);
        Sac2.Ajouter(x);        // Ajouter au Sac2 un obj de type CourrierAdmin
        break;
  case '7' :
        printf("Poids du Sac1  = %d\n",Sac1.poids());
```

```
            printf("Valeur avant affranch. du Sac1 = %d\n",Sac1.Valeur());
            Sac1.Affranchir_Sac();          // Affranchissement de tous les obj contenus
                                            // dans Sac1, (grâce au polymorphisme)
            printf("Valeur après affranch. du Sac1 = %d\n",Sac1.Valeur());
            getchar();                      // pause
            break;
        case '8' :
            printf("Poids du Sac2  = %d\n",Sac2.poids());
            printf("Valeur avant affranch. du Sac2 = %d\n",Sac2.Valeur());
            Sac2.Affranchir_Sac();          // Affranchissement de tous les obj contenus
                                            // dans Sac2, (grâce au polymorphisme)
            printf("Valeur après affranch. du Sac2 = %d\n",Sac2.Valeur());
            getchar();                      // pause
            break;
        case '9' :
            SacPostal grandSac(18,8);
            grandSac = Sac1 + Sac2;         // Concaténation de 2 Sacs
            printf("Poids du grandSac  = %d\n",grandSac.poids());
            printf("Valeur du grandSac = %d\n",grandSac.Valeur());
            getchar();                      // pause

        }
  } while (c != '0');
}
```

Dans la fonction principale *main()* on rempli les tableaux *Sac1* et *Sac2* par des objets de différents types (*Lettre, CourrierAdmin, Colis, ...*) et à la demande on peut invoquer la méthode virtuelle *Affranchir()* qui permet d'évaluer le tarif de chaque objet postal stocké dans les deux tableaux.

Les Piles (Last In First Out : LIFO)

Ce type de structure est très utilisé en programmation. C'est une collection d'objet où le dernier élément à avoir été stocké sera le premier à en être sortie.

Une pile est un type abstrait de données défini par ses opérations :

- Empiler(e,p) : permet de stocker l'élément e dans la pile p.
- Depiler(e,p) : permet de supprimer le dernier élément stocké dans la pile p et le retourner dans e.
- Pilevide(p) : Teste si la pile p est vide. Retourne Vrai ou Faux.
- Pilepleine(p) : Test si la pile p est pleine. Retourne Vrai ou Faux.

On peut déclarer une classe C1 pour représenter une pile d'entier, et une classe C2 pour représenter une pile de réels, et une classe C3 pour une pile de chaînes de caractères, etc...

En fait, quelque soit le type des éléments stockés dans une pile, les opérations ci-dessus restent inchangées. Donc il serait plus intéressant de procéder comme dans l'exemple précédent sur la collection d'objets postaux, en définissant une classe abstraite représentant tous les types d'éléments et de dériver une classe spécialisée pour chacun des types considérés. Ou encore mieux on pourrait déclarer une pile comme étant une classe paramétrée (qu'on appelle aussi classe générique) où le type des éléments joue le rôle de paramètre de la classe.

En C++ on utilise les *template* (ou patron) pour représenter une classe générique:

```
template <class T> class Pile
{
        T  *tab[100];           // la taille maximale de la pile est de 100 pointeurs
        int nb_elt;             // nombre d'éléments dans la pile
public:
        Pile() { nb_elt = 0; }  // constructeur : initialise le nombre d'éléments
        int Empiler(T *e);      // retourne 0 si débordement de pile sinon 1
        int Depiler(T *e);      // retourne 0 si la pile est vide sinon 1
        int Pilevide() { return (nb_elt == 0); }
        int Pilepleine() { return (nb_elt == 100); }
};

template <class T> int Pile<T>::Empiler(T *e)
{
        if (nb_elt < 100) {
                tab[nb_elt++] = e;
                return 1;
        } else return 0;
}
template <class T> int Pile<T>::Depiler(T *e)
{
        if (nb_elt > 0) {
                e = tab[--nb_elt];
                return 1;
        } else return 0;
}
```

Maintenant utilisant cette classe générique (*pile*) pour représenter une collection de livres (empilés les uns sur les autres):

```
#include <string.h>

// la classe des éléments manipulés
class Livre  {
        char *Titre;
        // ... des attributs représentant les livres ...
public:
        Livre(char *ch);                // le constructeur
        ~Livre() { delete Titre; }      // et le destructeur
        // ... des méthodes pour manipuler les livres ...
};

Livre::Livre(char *ch)  {
        Titre = new char[strlen(ch) + 1];
        strcpy(Titre,ch);
}

void main()  {
        Pile<Livre> pile1;              // instantiation d'une pile de livre
        char ch[40];                    // pour lire les titres de livres
        cout << "Donner un titre de livre : ";
        cin >> ch;                      // lire un titre
        Livre *p = new Livre(ch);       // création d'un obj Livre avec le titre lu
        pile1.Empiler(p);               // et empilement du livre
        cout << "Donner un titre de livre : ";
        cin >> ch;                      // lire un autre titre
        Livre *p = new Livre(ch);       // création d'un autre obj Livre avec le titre lu
        pile1.Empiler(p);               // et empilement du livre
        // ... etc ...
}
```

Les Files (First In First Out : FIFO)

Ce type de collection est similaire au type Pile sauf que le premier entré sera le premier à sortir. C'est une file d'attente standard.

On peut l'implémenter par un tableau circulaire ou par une liste linéaire chaînée.

Comme les piles, les Files sont souvent représentées par des classes génériques. Les opérations permises sont pour le type abstrait de données File sont :

- Enfiler(e,F) : insérer un élément en queue de file F.
- Defiler(e,F) : supprimer le premier élément (la tête de file) de F et le retourner dans e.
- Filevide(F) : teste si la file F est vide.
- Filepleine(F) : teste si la file F est pleine.

7) PERSISTANCES DES OBJETS ET BASE DE DONNEES OBJET

Notion de persistance d'objet:

Dans un programme orienté objet, les instances sont créées dans la mémoire centrale de l'ordinateur. A la fin de l'exécution, tous les objet seront détruits implicitement ou explicitement. Dans certaines applications, et pour éviter cette éventuelle perte d'information, il est nécessaire de sauvegarder les objets sur une mémoire non volatile (le disque par exemple) sous forme de fichiers ou de bases de données orientées objets. C'est la notion de persistance.

Un objet est dit persistant, si son existence (sa durée de vie) dépasse celle du programme qu'il l'a créé. Ce type d'objet est très courant dans les applications de type bases de données.

Dans les langages de programmations (comme C++), le programmeur doit se charger de faire persister les objets qu'il juge nécessaires en écrivant des procédure de sauvegarde et de chargement de certaine parties de l'objet (les attributs). Les méthodes, par contre, sont déjà stockées sous forme de fichiers dans le programme source où la classe à été déclarée.

Le stockage sur disque de la partie statique de l'objet (les attributs) est presque similaire au stockage des données sur des fichiers. Pour cela on utilise les opérations de lecture/écriture standard du langage C (fread, fwrite, fprintf, fscanf, ...) ou alors les classes standard de manipulation de fichiers en C++ (fstream, ifstream et ofstream).

Voici un exemple en C++, illustrant les mécanismes de sauvegarde et de chargement d'objets en mémoire secondaire. Considérons une mini gestion d'un agenda téléphonique :

```
#include <fstream.h>          // librairie standard de manipulation de fichiers

class personne                        // une entrée de l'agenda
{
        char nom[20];
        char adresse[25];
        char numtel[15];
public:
        void Lire_Donnees();
        void Afficher_Donnees();
        void Lire_Fichier(ifstream& fich);
        void Ecrire_Fichier(ofstream& fich);
};
```

```
void personne::Lire_Donnees()  {
        char temp;      // terminateur de saisie d'une chaine de caractères
        // Lecture du Nom de la personne:
        cout << "Entrer le Nom : ";
        cin.get(nom,20);        // méthode prédéfinie dans iostream pour la lecture d'une
                                // chaine de 20 car (max) pouvant contenir des espaces.
        cin.get(temp);          // lire le car terminateur de saisie '\n' par défaut.
        // Lecture de l'Adresses de la personne :
        cout << "Entrer l'Adresse : ";
        cin.get(adresse,25);
        cin.get(temp);
        // Lecture du Numéro de Téléphone de la personne :
        cout << "Entrer son Numéro de Téléphone : ";
        cin.get(numtel,15);
        cin.get(temp);
}

void personne::Afficher_Donnees()  {
        cout << "Nom : " << nom;
        cout << "\tAdresse : " << adresse;   // '\t' designe une tabulation
        cout << "\tTelephone : " << numtel << endl;
}

void personne::Lire_Fichier(ifstream& fich)  {
        char temp;
        fich.get(nom,20);       // Lire un max de 20 car dans l'attribut nom
        fich.get(temp);                 // Lire le séparateur '\n'
        fich.get(adresse,25);   // Lire un max de 25 car dans l'attribut adresse
        fich.get(temp);                 // Lire le séparateur '\n'
        fich.get(numtel,15);    // Lire un max de 15 car dans l'attribut numtel
        fich.get(temp);                 // Lire le séparateur '\n'
}

void perssone::Ecrire_Fichier(ofstream& fich)  {
  fich << nom << endl;          // Ecrire les car formant le nom et terminer par '\n'
  fich << adresse << endl;      // Ecrire l'adresse et terminer par '\n'
  fich << numtel << endl;       // Ecrire le numéro de tel et terminer par '\n'
}

void main()  {
        personne agenda[100];           // Notre agenda peut contenir 100 personnes
        int compteur = 0;               // compteur d'objet de type personne
        int c = -1;
        char tmp;
        while (c != 0) {
                cout << "------- M E N U --------" << endl << endl << endl;
                cout << "1)  Inserer une nouvelle personne dans l'Agenda" << endl;
                cout << "2)  Supprimer une personne de l'Agenda" << endl;
                cout << "3)  Recherche par nom" << endl;
                cout << "4)  Recherche par Numero de Tel" << endl;
```

```
                cout << "5)  Chargement de l'Agenda depuis le fichier" << endl;
                cout << "6)  Sauvegarde de l'Agenda dans le fichier" << endl;
                cout << "0)  Fin de traitement" << endl << endl << endl;
                cout  << "\tVotre Choix : ";
                cin >> c; cin.get(tmp);
                switch (c) {
                        case 1 :
                                // ... Traitement de l'insertion dans une table ...
                                break;
                        case 2 :
                                // ... Traitement de la suppression dans une table ...
                                break;
                        case 3 :
                                // ... Traitement de la recherche par nom ...
                                break;
                        case 4 :
                                // ... Traitement de la recherche par numtel ...
                                break;
                        case 5 :
                                // Chargement à partir du fichier "agenda.dat"
                                // assignation de l'objet f_in au fichier "agenda.dat"
                                // et ouverture de celui-ci en mode lecture :
                                ifstream f_in("agenda.dat");
                                // commencons par lire le nb d'objets stockés:
                                f_in >> compteur;
                                // ensuite on récupère les objets en séquence:
                                for (int i=0; i<compteur; i++)
                                        agenda[i].Lire_Fichier(f_in);
                                // Fermer le fichier :
                                f_in.close();
                                break;
                        case 6 :
                                // Sauvegarde des objets dans le fichier "agenda.dat"
                                // assignation de l'objet f_out au fichier "agenda.dat"
                                // et ouverture de celui-ci en mode ecriture :
                                ofstream f_out("agenda.dat");
                                // commencons par stocker le nombre d'objets:
                                f_out << compteur;
                                // ensuite les objets en séquence:
                                for (int i=0; i < compteur; i++)
                                        agenda[i].Ecrire_Fichier(f_out);
                                // Fermer le fichier :
                                f_out.close();
                                break;
                }
        }
}
```

Dans l'exemple ci-dessus, on a vu comment faire persister un objet en sauvegardant son état (les valeurs de ses attributs) dans des structures de fichiers. Cependant il existe un problème lié à la référence d'objet:

Supposons qu'un objet A référence un objet B, c'est à dire qu'un des attributs de A contient un pointeur vers B. Lorsqu'on sauvegarde les objets A et B sur disque, comment maintenir le lien entre A et B sachant que les pointeurs représentent des adresses en mémoire centrale où les objets ont été créés.

Si on sauvegarde sur disque les pointeurs, lors du chargement on ne peut pas recréer les objets chargés aux mêmes emplacements en mémoire centrale, c'est le principe de l'allocation dynamique. (voir Fig VII-1)

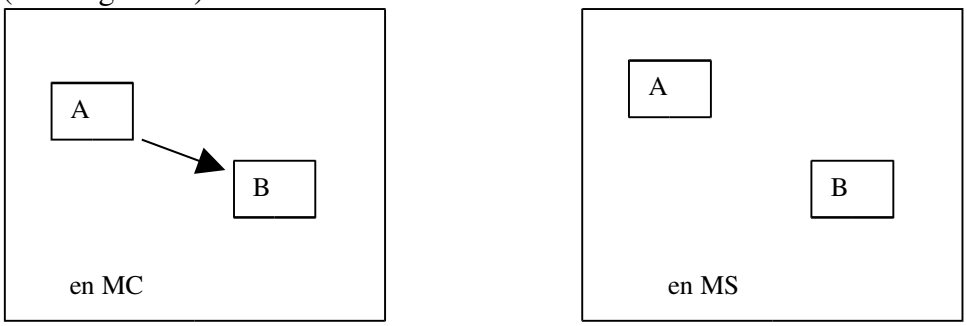


Fig VII-1 : Problème de la référence d'objet

La solution serait de rajouter un attribut à chaque objet, contenant un code unique permettant d'identifier l'objet, indépendamment de son emplacement en mémoire centrale ou en mémoire secondaire. C'est la notion d'IDentifiant d'Objet (OID) très utilisée dans les bases de données objets.

Ainsi si l'objet A référence l'objet B, au lieu de mettre le pointeur de B dans A, on placera l'OID de B dans A. Ceci assure que quelque soit l'emplacement de A et de B, (en MC ou MS) la référence existera toujours.

Bases de données objet:

La tendance actuelle dans le domaine des bases de données et d'essayer de remplacer les systèmes relationnels par des systèmes orientés objets. Ces derniers sont censés offrir des moyens de modélisation plus puissants et plus souples que leurs prédécesseurs relationnels trop pauvres sémantiquement pour les nouvelles applications bases de données.

Il existe deux grandes classes de systèmes de bases de données objets :

1) Ceux basés sur des langages de programmation objet comme C++ ou Smaltalk, auquels on leur rajoute des possibilités de gestion de la persistance d'objets et de gestion dynamique de schémas.

2) Ceux basés sur des systèmes relationnels comme Postgres ou Iris et auxquels on a rajouté une couche externe offrant les fonctionnalités du modèle objet.

On peut faire une analogie entre les systèmes de gestion de bases de données relationnels et orientés objets. L'équivalent du schéma relationnel est la hiérarchie des classes (ou le graphe d'héritage). L'équivalent d'une relation est la classe, et l'équivalent d'un tuple dans les systèmes relationnels est l'instance ou l'objet.

En plus des langages de programmation objets qui permettent un accès navigationnel, les systèmes de bases de données offrent également, des langages de requête déclaratifs généralement proche de SQL, auquel on rajoute, des extensions pour être adapté au modèle de données objet.

Supposons par exemple que le schéma de notre base de données contienne une classe *employe* ayant comme attributs (publiques) le numéro de sécurité social (*nss*), le *salaire* et un champs référence (*responsable*) vers la classe *employe* représentant son responsable direct. Cette classe hérite les propriétés traditionnelles de la classe *personne*, comme le *nom*, le *prenom*, et un champs multivalué représentant les *enfants* de la personne, etc ..

On pourra alors consulter la base à l'aide de requêtes (du type SQL étendue) :

```
select e.nom, e.salaire                    // sélectionner le nom et le salaire de e
from employe e                             // sachant que e est un employé
where (e.salaire > e.responsable.salaire)  // à condition que son salaire soit supérieur
                                           // au salaire de son responsable.
```

Cette requête va afficher les noms et salaires des employés ayant un salaire supérieur à celui de leur responsable.

Remarquer comment la variable *e* qui est du type *employe* peut référencer des attributs hérités d'une autre classe (*personne*) dans l'expression : *e.nom*

Une autre remarque concernant cet exemple, est la généralisation de la notation pointée: '*e.responsable.salaire*' est appelée une **expression de chemin**, car elle permet de traverser le graphe de référence des objets de la base (on l'appelle aussi jointure par référence).

Voici une autre requête où on désir connaître les numéros de sécurité sociale et les noms des employés ayant au moins un enfant âgé de plus de 18 ans:

select e.nss e.nom
form employe e, personne p
where (e.enfants.Contains(p) and p.age() > 18)

On a supposé que la classe *personne* possède une méthode *age()* et un attribut (publique) *enfants* de type collection(*personne*). *Contains(p)* est une méthode générique permettant de vérifier l'appartenance de *p* à la collection.

8) CONCEPTION ORIENTEE OBJET

L'un des principaux buts des concepts de l'orienté objet est de faciliter le développement de grandes applications. Mais à eux seuls, ils ne sont pas suffisant pour aborder de tels systèmes complexes, il faut leur associer une méthode de conception tirant avantage de l'approche objet.

Beaucoup de méthodes de conception ont été développées et sont en général très proche l'une de l'autre. Certaines sont axées sur la partie analyse du problème, d'autres sur la partie conception. Certaines privilégient le comportement des objets (orientés traitement) d'autres se concentrent sur la modélisation de l'état des objets (orientés données).

Dans tous les cas il n'est pas interdit qu'un programmeur mélange plusieurs méthodologies pour concevoir son système. L'essentiel dans toute méthode de conception est d'avoir le bon sens, d'être organisé et de rester cohérent dans sa stratégie.

En général les méthodes d'analyses/conceptions orientées objets doivent permettre (au minimum) ce qui suit :

1- Identification des objets formant le système.
2- Classification de ses objets, en identifiant les propriétés en commun.
3- Définir les comportements des objets (spécification des méthodes).
4- Etablir les liens d'héritage entre classes et par factorisation des propriétés communes aux classes, générer des classes abstraites dans les niveaux supérieurs du graphe d'héritage.
5- Etablir le graphe de composition entre les différents objets complexes et leurs composants (utilisation des collections dans le cas de champs multivalués).
6- Etablir le graphe d'association entre objets et recenser ainsi les différents envoies de messages générés (ce qui permet la communication entre les objets du système).

Le processus de développement est composé de trois étapes:
- Analyse : définir l'étendue du problème à résoudre.
- Conception : créer une structure d'ensemble pour le système.
- Mise en œuvre : écrire et tester le code.

Ce processus est itératif et contient généralement beaucoup de retour arrières afin de s'assurer de la bonne compréhension du problème et que les bons choix ont été fait.

- Un exemple:

Considérons un exemple pratique, utilisant la méthode UML (Unified Modelling Language) proposée par Rumbaugh, Booch et Jacobson en 1995. Il s'agira de concevoir et développer un système de prospectus électronique sur les cours enseignés dans un institut de formation:

- A partir d'une série de discussions et d'interviews avec les futurs utilisateurs du systèmes ou n'importe quelle personne plus ou moins concernée, on pourra résumer notre **analyse du domaine** en texte décrivant les propriétés et fonctionnalités attendues du nouveau système.

Le prospectus électronique doit fournir des informations sur les différentes filières de l'institut, aux étudiants, aux enseignants et aux responsables administratifs.

Chaque filière est composé d'un ensemble de modules étalés sur deux années, chacun possède un nom et une description. Certains modules sont théoriques, d'autres sont pratiques sous forme de stages dans des salles de TP. Les modules théoriques sont caractérisés par un sommaire contenant le plan des chapitres et par un volume horaire hebdomadaire.

Le système doit fournir des informations sur les enseignants, les filières et les modules. Il doit ainsi permettre de connaître quels sont les modules enseignés, dans une filière donnée, ou alors connaître le numéro de bureau, l'adresse électronique (e-mail) et la spécialité d'un enseignant donné, ou encore connaître pour un enseignant la liste des modules qu'il enseigne. Pour chaque filière il existe un enseignant jouant le rôle de responsable de filière et qui aura la charge de mettre à jour les descriptions des modules. Les responsables administratifs ont la possibilité de mettre à jour le prospectus en y insérant et/ou supprimant des enseignants et des filières.

- En analysant ce texte sémantiquement et de manière préliminaire on peux établir un diagramme montrant les **acteurs**, les **services** et les **frontières** du système : (Fig VIII-1)

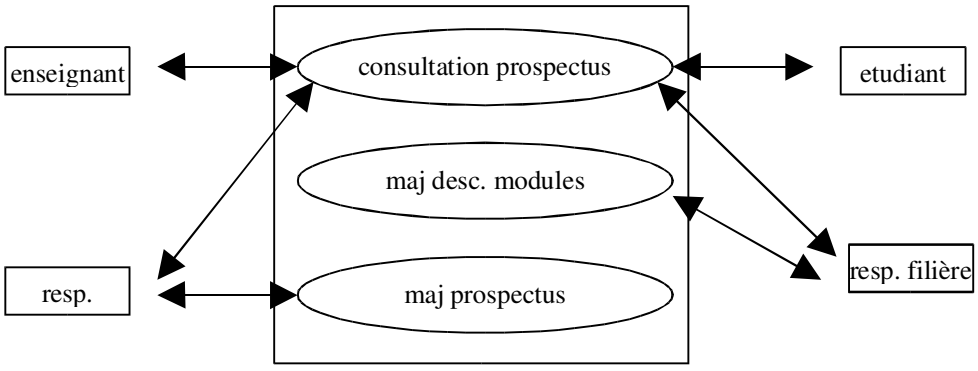


Fig VIII-1 : Les acteurs, cas d'utilisation et les frontières du système

Dans cette figure on voit les différents types d'utilisateurs finaux du système (acteurs ou objet externes) et leurs liens avec les services fournis.

- Afin d'identifier les objets du systèmes on pourra procéder par une simple analyse textuelle à la recherche des groupes nominaux. On peut par exemple commencer par souligner les noms ou groupes nominaux:

Le prospectus électronique doit fournir des informations sur les différentes filières de l'institut, aux étudiants, aux enseignants et aux responsables administratifs.

Chaque filière est composé d'un ensemble de modules étalés sur deux années, chacun possède un nom et une description. Certains modules sont théoriques, d'autres sont pratiques sous forme de stages dans des salles de TP. Les modules théoriques sont caractérisés par un sommaire contenant le plan des chapitres et par un volume horaire hebdomadaire.

Le système doit fournir des informations sur les enseignants, les filières et les modules. Il doit ainsi permettre de connaître quels sont les modules enseignés, par année, dans une filière donnée, ou alors connaître le numéro de bureau, l'adresse électronique (e-mail) et la spécialité d'un enseignant donné, ou encore connaître pour un enseignant la liste des modules qu'il enseigne. Pour chaque filière il existe un enseignant jouant le rôle de responsable de filière et qui aura la charge de mettre à jour les descriptions des modules. Les responsables administratifs ont la possibilité de mettre à jour le prospectus en y insérant et/ou supprimant des enseignants et des filières.

- Ensuite on va éliminer un certain nombre de candidats pour des raisons de redondances, de notions vagues, ou parce qu'ils désignent plutôt des attributs ou des opérations ou alors des constructions d'implémentations, ...

La liste suivante représente les objets candidats :

Prospectus, Filière, Module Théorique, Module Pratique, Enseignant

Les objets comme : Responsable Administratif, Etudiant sont éliminés car ils représentent des objets externes au système. Contrairement à Enseignant qui est en même temps externe (utilisateur du système) et interne (fait partie des informations manipulées par le système).

L'objet Responsable de filière est aussi éliminer car il désigne un rôle joué par l'objet Enseignant.

- A ce moment là, (une fois qu'on établit un ensemble de classes représentant les type d'objets choisis), on va commencer par maintenir un **dictionnaire de données** décrivant chaque classe du système. Dans ce dictionnaire de données on notera, de façon textuelle, la définition de

chaque classe incluant les attributs les associations et les opération permises. Le dictionnaire restent en constante évolution durant toute la phase d'analyse et de conception pour être de plus en plus détaillé et corrigé. Par exemple on pourra avoir une entrée concernant les enseignants :

Enseignant	Un enseignant peut enseigner plusieurs modules et peut jouer le rôle de responsable de filière. Parmi les attributs d'un enseignant il y a le numéro de bureau, l'adresse e-mail et sa spécialité.

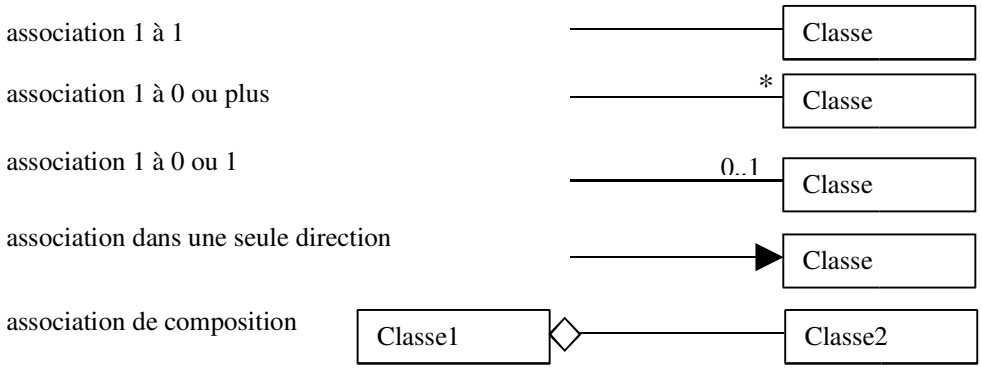
- A ce niveau on commencera par identifier les associations entre les différentes classes. Les associations sont importantes car elle déterminent les routes que peuvent prendre les messages entre objets.

On peut identifier la liste des associations candidates en identifiant, dans le texte décrivant le problème, les verbes et groupes verbaux relatifs aux phrases contenant les objets déjà choisis :

Chaque filière est formée par un ensemble de modules...
Le système doit fournir des informations sur les enseignants, les filières et les modules...
...connaître pour un enseignant la liste des modules qu'il enseigne...
Pour chaque filière il existe un enseignant jouant le rôle de responsable de filière...
... qui (le responsable de filière) ... de mettre à jour les descriptions des modules...
... de mettre à jour le prospectus .. y insérant/y supprimant des enseignants et des filières.

De cette liste de phrases on peut voir qu'il y a une relation d'agrégation entre Filière et Module. Une relation d'association entre Enseignant et Module. Une autre association entre Enseignant (certains d'entre eux) et Filière existe aussi. Certaines de ces phrases candidates désignent plutôt des actions comme par exemple : "*Le système doit fournir des informations..."* ou bien *"... de mettre à jour les description des modules..."*. Ce type d'association sera représenté par des opérations dans le **diagramme des classes** décrivant les différents liens entre classes, qu'on verra par la suite.

Dans la notation UML, les associations sont représentées par des lignes entre les objets concernés et doivent être nommées de manière appropriée. Les cardinalités des associations doivent aussi apparaître dans le diagramme. Les agrégations (association de composition) sont identifiées dans le diagramme par un losange au niveau de l'objet composant. Voici la légende utilisée pour les différentes cardinalités:



- L'identification des attributs fera appel à notre expérience dans la modélisation de ce type d'application, car dans le texte décrivant le problème, on ne trouvera pas tous les attributs nécessaire. Il faudra peut être chercher ces informations auprès des personnes concernés, puisqu'on a déjà établit la liste des classes candidates, ceci facilitera la recherche de ces données.

- Il reste maintenant à raffiner l'organisation du graphe d'héritage en utilisant les mécanismes de généralisation permettant de factoriser les propriétés communes à certaines classes pour abstraire une nouvelle super-classe dans le graphe. Dans notre exemple les propriétés communes entre Module Théorique et Module Pratique serviront à définir une classe abstraite Module d'où dériveront (par héritage) les deux premières.

Le diagramme de classes est donné en figure VIII-2.

- Maintenant il reste à modéliser la dynamique, c'est à dire trouver les différents envoies de messages engendrés entre les objets lors du déroulement de l'application. Pour ce faire, on imagine les différents type scénarios possibles entre les acteurs et le système et pour chacun d'eux, on construit des **diagrammes de séquences** montrant l'interaction des objets pour réaliser la tâche en question. Généralement on commence d'abord par modéliser les scénarios "normaux", les cas exceptionnels sont laissés à la fin.

Par exemple voici ce qui pourrait être un scénario "normal" où un responsable administratif désire ajouter une filière dans le prospectus:

Le prospectus (l'objet) crée un nouveau objet de type Filière et demande au responsable d'entrer le nom de la nouvelle filière. Ensuite le prospectus demande au responsable administratif de sélectionner parmi la liste des enseignants de l'institut, celui qui sera responsable de la filière. Le prospectus établit alors un lien entre la filière et l'enseignant choisi comme étant son responsable.

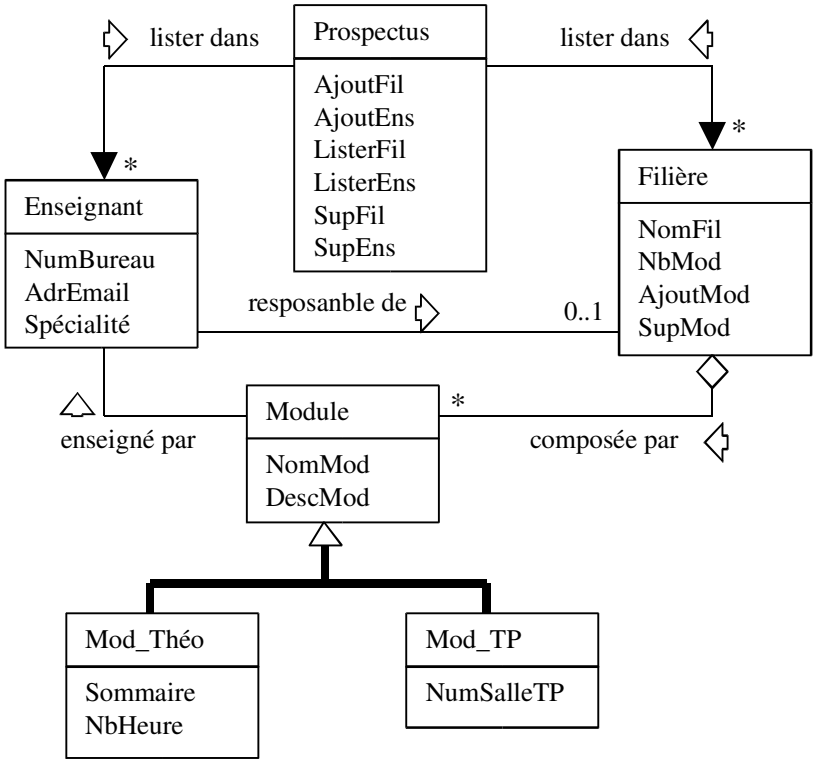


Fig VIII-2 : Diagramme de Classes

A partir de ce scénario on peut établir le diagramme de séquence suivant : (Fig VIII-3)

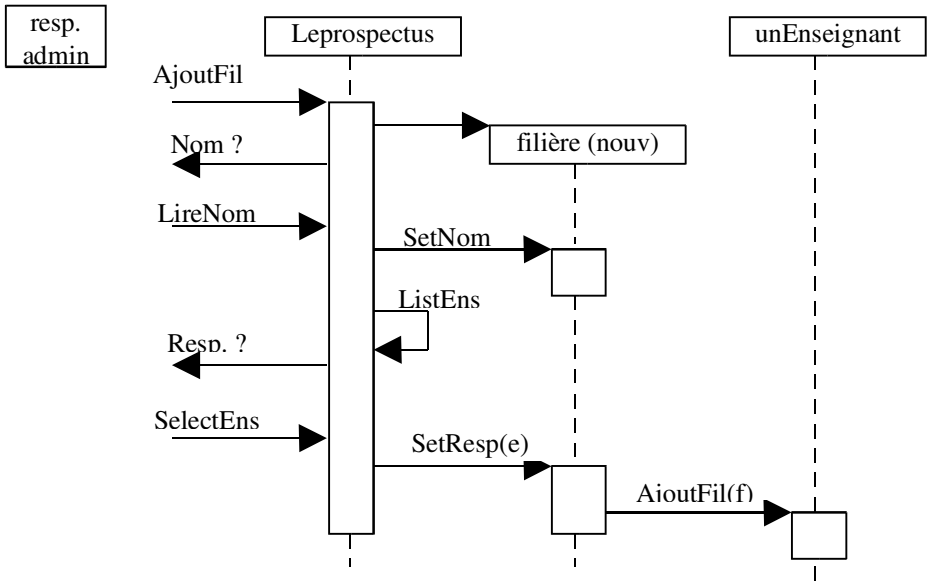


Fig VIII-3: diagramme de séquence montrant les envois de messages

- Après avoir recensé les différents scénarios, il convient maintenant d'écrire le programme (en C++ par exemple) même s'il n'est pas complet dans les premières itérations du cycle de développement du système. On commence, en général, par un prototype à des fins de tests et d'évaluations pour raffiner les prochaine itérations jusqu'à obtenir le produit final.