Spécification

Pour spécifier l'architecture d'une application réalisée par une méthode orientée objets, il existe plusieurs formalismes graphiques tels que OOA (Object Oriented Analysis), OMT (Object Modeling Technique), UML (*Unified Modeling Language*), Booch (modèle de Grady Booch), HOOD, ...
La méthode UML est un sur ensemble des méthodes Booch et OMT.

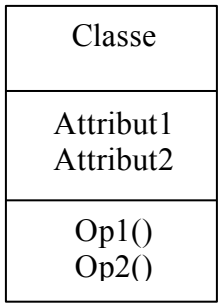
L'élément de base de l'activité de spécification dans toutes ses méthodes et la distinction et la reconnaissance des objets / classes (et de leurs propriétés).

Ces formalismes sont très proches les uns des autres, en particulier OOA, OMT et UML. Sans nous intéresser à ces méthodes en détails (voir les autres cours), nous utiliserons le formalisme graphique UML pour la spécification des applications à réaliser.

Le formalisme UML

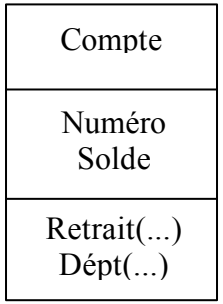
Ce formalisme est très proche du formalisme OMT.

Classe et objet



un objet est le résultat de l'instanciation d'une classe. Cette instanciation définit la relation **"est-un".**

Exemple :

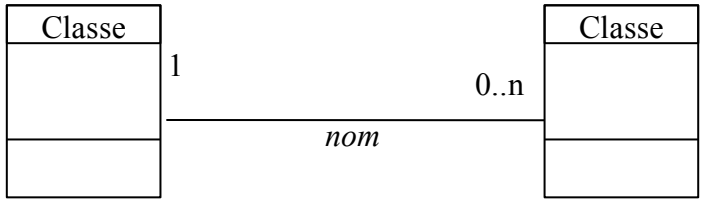


NB : une classe **abstraite** est une classe non instantiable; elle est utilisée seulement pour l'héritage.

Association

Une association représente un lien structurel entre classes d'objets.

La plupart des associations sont binaires (connectent 2 classes).



Une association est proche d'une agrégation (voir plus loin) mais d'une sémantique plus faible (l'agrégation est une des méthodes d'organisation dominantes de la pensée humaine).

Les associations s'ajoutent aux attributs pour donner les liaisons requises demandées par un objet.

Nom de l'association : la liaison entre les classes peut être nommée.

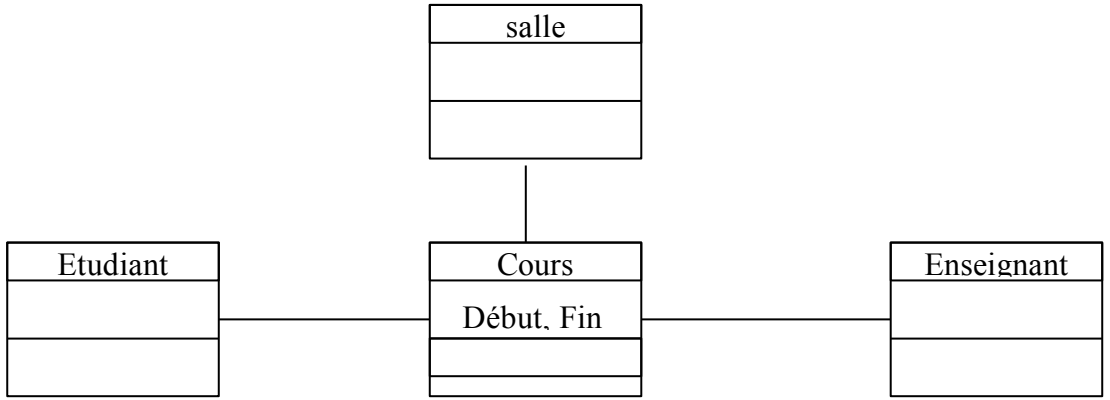
Multiplicité : chaque classe peut participer à l'association en de 0 à N exemplaires.

Exemple d'association

Ici, un plan de vol particulier doit être exécuté par exactement un*Avion*; tout *Avion* peut exécuter zéro ou plusieurs plans de vol.

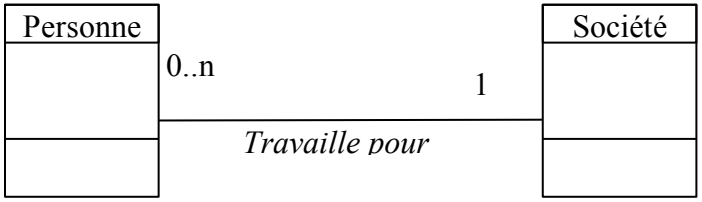
La plupart des associations sont binaires mais on peut avoir besoin d'associations n-aires.

Exemple d'association ternaire:



Nommage des associations

On peut nommer une association en général par une forme verbale (telle que *travaille pour*, *est employé par*, ...):



Le sens de lecture du nom peut être précisé par les signets '<' ou '>' ou par un triangle.

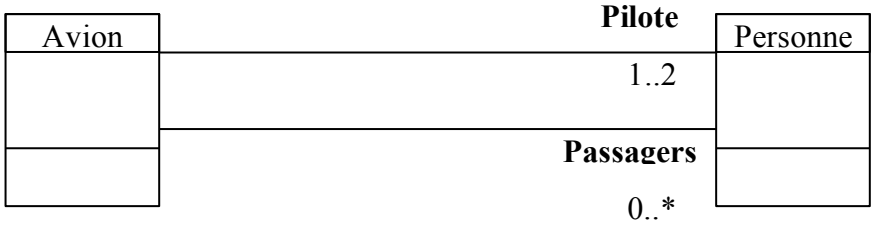
Le nom d'une association permet une clarification de lecture du diagramme et n'apparaît pas dans le code généré pour le diagramme.

Rôles et nommage des rôles

L'extrémité d'une association est appelée **rôle**. Chaque association binaire possède deux rôles. Le rôle décrit comment une classe voit une autre classe au travers une association. Un rôle est nommé au moyen d'une forme nominale. Le nom d'un rôle se distingue du nom de l'association.

Dans une association ou agrégation, on peut attribuer un rôle (un nom identifiant) à chaque extrémité de la liaison.

En général, on ne mélange pas l'usage du nom de l'association avec le nommage des rôles. On préfère le nommage des rôles au nom de l'association, en particulier lorsque deux classes sont reliées par plusieurs associations.



Multiplicité des associations

Chaque rôle d'une association porte une indication de **multiplicité** qui montre combien d'objets de la classes considérée peuvent être liés à un objet de l'autre classe. La multiplicité est une information portée par le rôle sous la forme de :

1 : un et un seul (valeur par défaut)

0..1 : zéro ou un

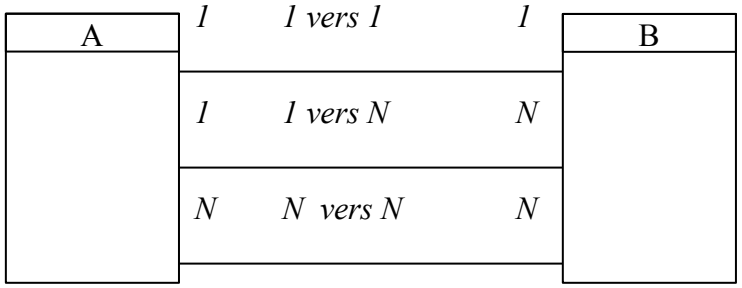
M..N : de M à N (entiers naturels)

* : de zéro à plusieurs

0..* : idem (noté également 0..N)

1..* : d'un à plusieurs (ou 1..N)

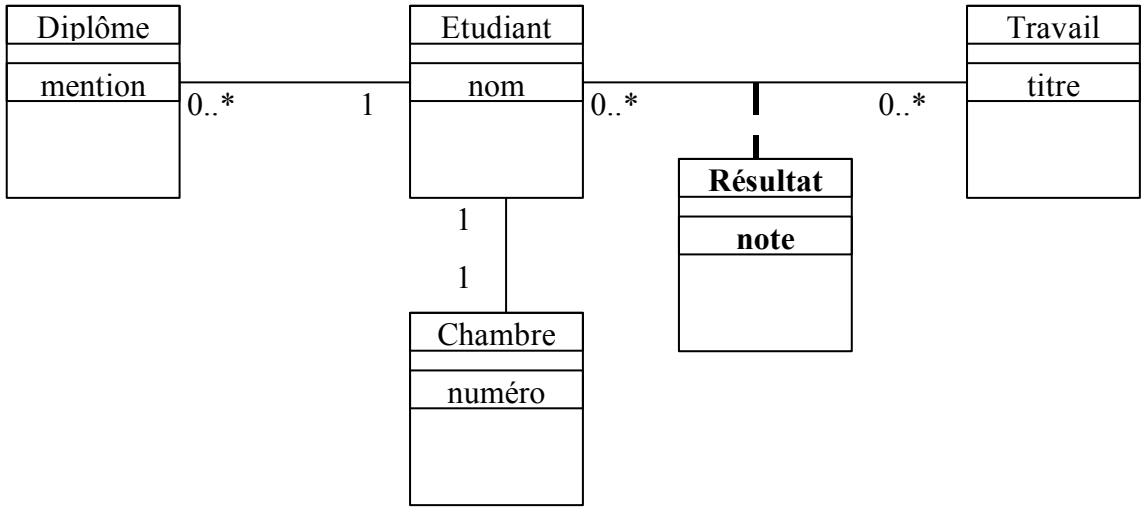
Les valeurs de multiplicité sont souvent employées pour décrire de manière graphique les associations. Les formes les plus courantes sont les associations *1 vers 1*, *1 vers N* et *N vers N*.



Classe-association (attribut de lien)

Pour les associations 1 à 1, les attributs de l'association peuvent toujours être déplacés dans une des classes qui participent à l'association. Dans le cas d'une association 1 vers N, le déplacement est généralement possible vers la classe du coté N.

Toutefois, et en particulier dans le cas d'une association N vers N, il est fréquent de promouvoir l'association au rang d'une classe pour augmenter la lisibilité ou en raison de la présence de l'association vers d'autres classes.



L'association entre la classe *Etudiant* et la classe *Travail* est de type *N vers N*. La classe *Travail* décrit le sujet et la solution par l'étudiant n'est pas conservée.

Dans le cas des contrôle des connaissances, chaque étudiant compose individuellement sur un travail donné et la note obtenue ne peut être stockée ni dans un Etudiant en particulier (car il effectue de nombreux travaux dans sa carrière) ni dans un Travail donné (car il y a autant de notes que d'étudiant comme par exemple dans la liste des notes du test pour un ensemble d'étudiants, par exemple le test d'informatique pour les 1ère année, le test d'informatique pour les élèves de la 2nd année, ...).

La classe **Résultat** ci-dessus est appelée une **classe-association**. Cette classe enferme l'attribut *note* de l'association entre Etudiant et Travail.

Un attribut de lien (classe-association) est une propriété des liens d'une association.

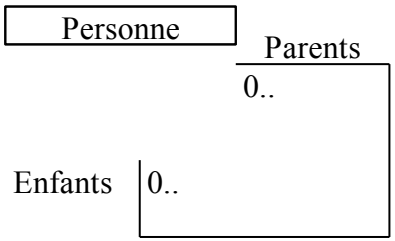
Contraintes sur les relations

Des contraintes peuvent être définies sur une relation. La multiplicité est un exemple de contraintes sur le nombre de liens qui peuvent exister entre deux objets. On représente une contrainte dans le diagramme entre accolades :

Ici, *{ordonné}* indique que la collection des comptes d'une personne est ordonnée.

D'autres contraintes (cf Merise) peuvent exister en UML.

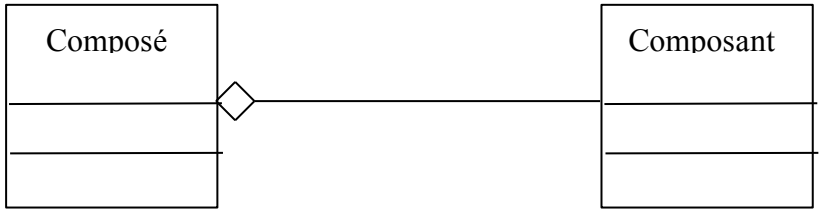
Les associations peuvent relier une classe à elle-même comme dans le cas des structures récursives. Ce type d'association est appelé association **réflexive**. Le nommage des rôles prend toute son importance pour distinguer les instances qui participent à la relation. L'exemple suivant montre la classe des personnes et la relation qui unie les parents et leurs enfants en vie.



Ici, toute personne possède de zéro à deux parents et de zéro à plusieurs enfants. Le nommage des rôles est essentiel à la clarté du diagramme.

Agrégation : structure Composé-Composant

Une agrégation représente une relation non symétriques dans laquelle une des extrémités joue un rôle prédominant par rapport à l'autre extrémité. En général, l'agrégation concerne un seul rôle d'une association.



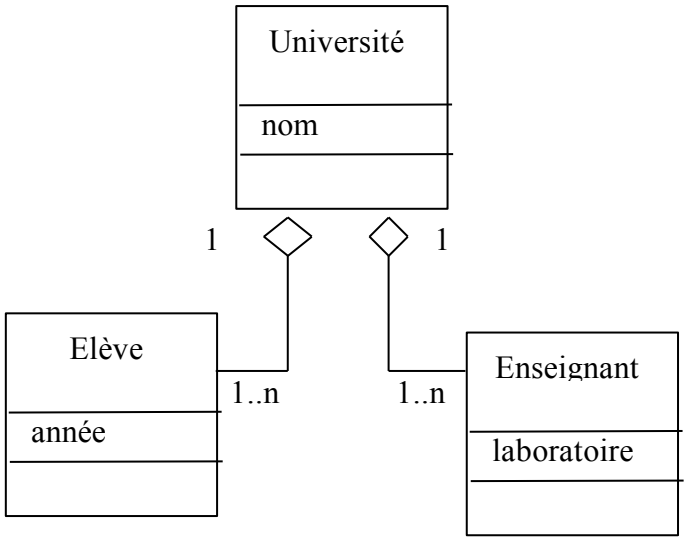
Dans une structure composé-composant, un petit losange apparaît du coté du composé. Le segment est dessiné depuis le composant vers le composé et le losange se trouve du coté du composé.

Les **critères** qui impliquent une agrégation sont :

- une classe fait partie d'une autre classe
- les valeurs d'attributs d'une classe se propagent dans les valeurs d'attributs d'une autre classe.
- une action sur une classe implique une action sur une autre classe
- les objets d'une classe sont subordonnés aux objets d'une autre classe.

L'inverse n'est pas toujours vrai : l'agrégation n'implique pas nécessairement les critères ci-dessus. Dans le doute, les associations sont préférables.

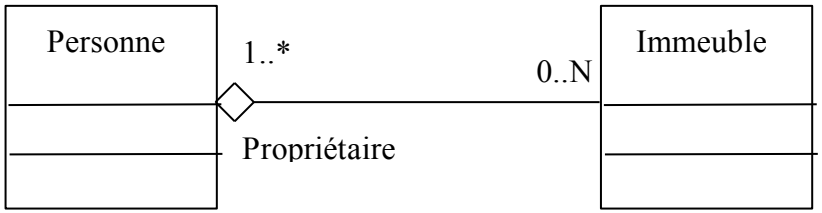
Exemple :



La multiplicité par défaut = 1

Une agrégation définit la relation "***partie-de***".

La multiplicité du coté de l'agrégat (composé) peut être supérieure à 1. Par exemple :



Ce diagramme montre que des personnes peuvent être copropriétaires des mêmes immeubles.

Différentes notions d'agrégation

Différentes notions :

Assemblage - Parties

Collection - Membres

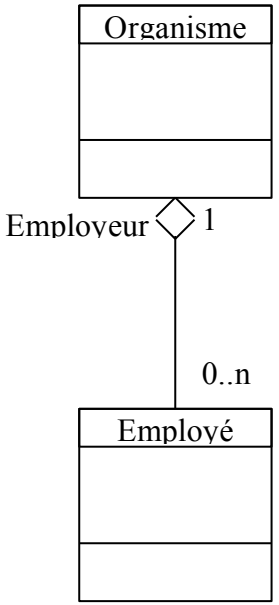
Contenant - Contenu

Exemple de Composé-Composant -- *Assemblage-Parties*:

Un avion est composé de 1 à 4 moteurs. Un moteur appartient à 0 ou un avion.

Exemple de Composé-Composant -- *Contenant - Contenu* (on suppose que les pilotes sont ˆ l'intérieur**) :**

Exemple de Composé-Composant - *Collection-Membres* :



Remarque : la notion d'agrégation ne suppose aucune forme de réalisation particulière. La **contenance physique** est un cas particulier de l'agrégation, appelé **composition**.

La composition

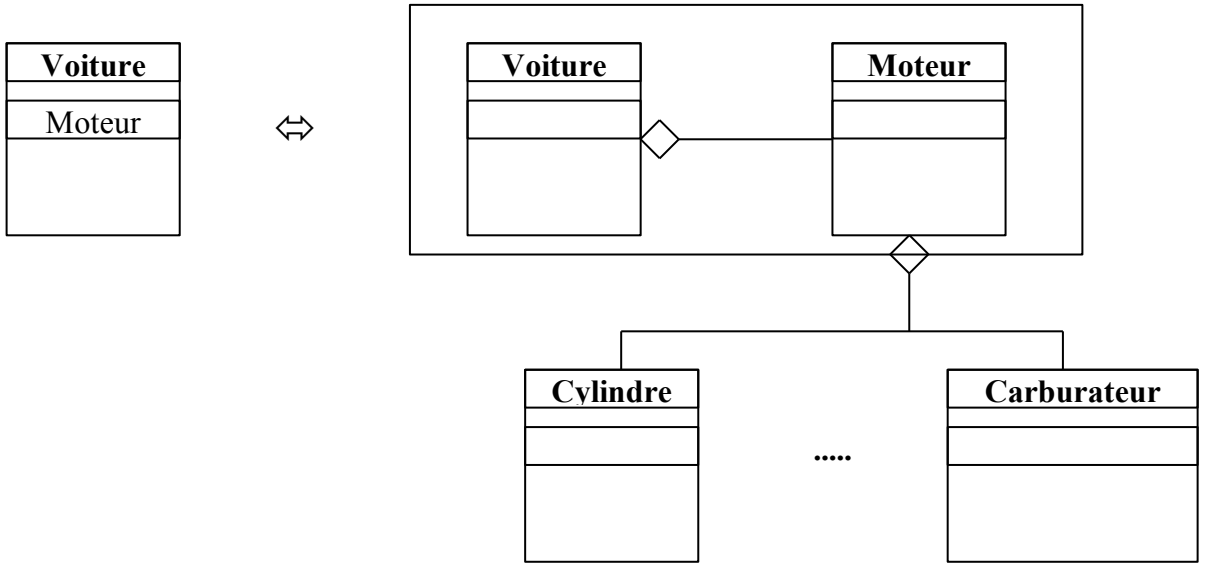
Les attributs constituent un cas particulier d'agrégation réalisée par valeur : ils sont physiquement contenus par l'agrégat. Cette forme d'agrégation est appelée composition.

NB : en UML, la composition (agrégation par valeur) se présente dans les diagramme par un losange de couleur noire. Pour simplifier les diagrammes, nous omettrons ce losange noir.

La composition implique une contrainte sur la valeur de multiplicité du coté de l'agrégat : elle ne peut prendre que les valeurs 0 ou 1.

La composition et les attributs sont interchangeables (sémantiquement équivalents). On utilise une composition à la place d'attribut quand cet attribut participe à d'autres relations dans le modèle.

Exemple :

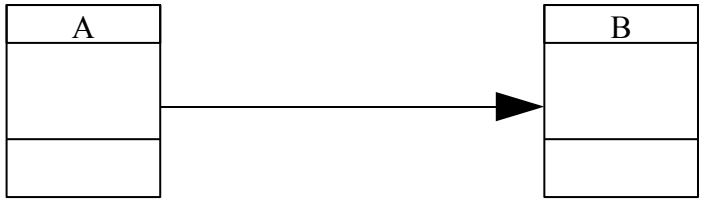


Dans le diagramme de gauche, *Moteur* est un attribut de *Voiture*. Dans le diagramme de droite, on établit une composition entre *Voiture* et *Moteur* car la classe *Moteur* participe à d'autres relations.

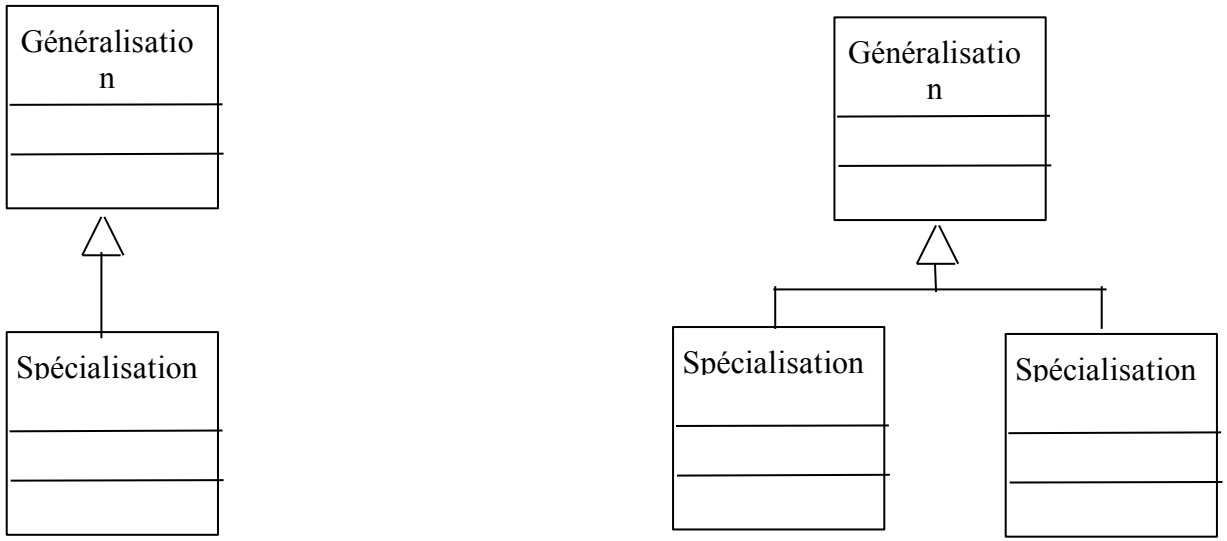
La navigation

Par défaut, une association est navigable dans les deux sens. Dans certains cas, seule une seule direction de navigation est utile.

Dans l'exemple suivant, les objets instances de A voient les objets instances de B mais les objets instances de B ne voient pas les objets instances de A.

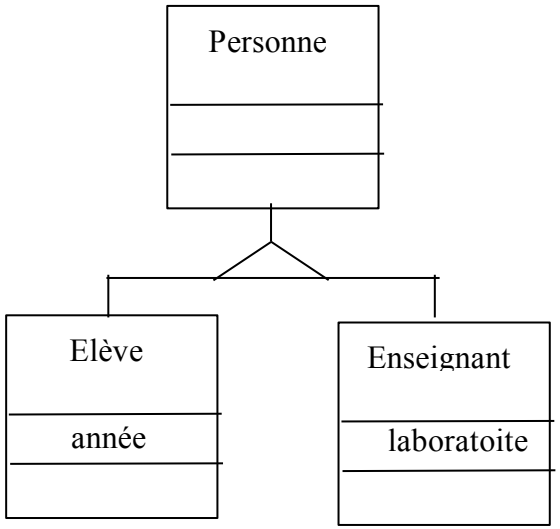


Structure Gén-Spéc (Héritage)



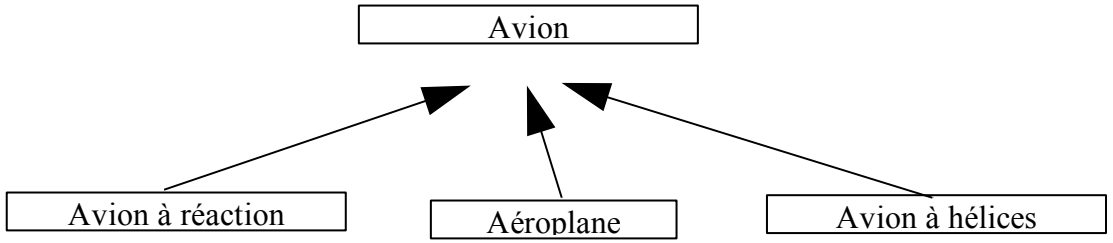
NB : La Généralisation - Spécialisation (Gen-Spec) définit la relation "**sorte-de**"

Exemple :

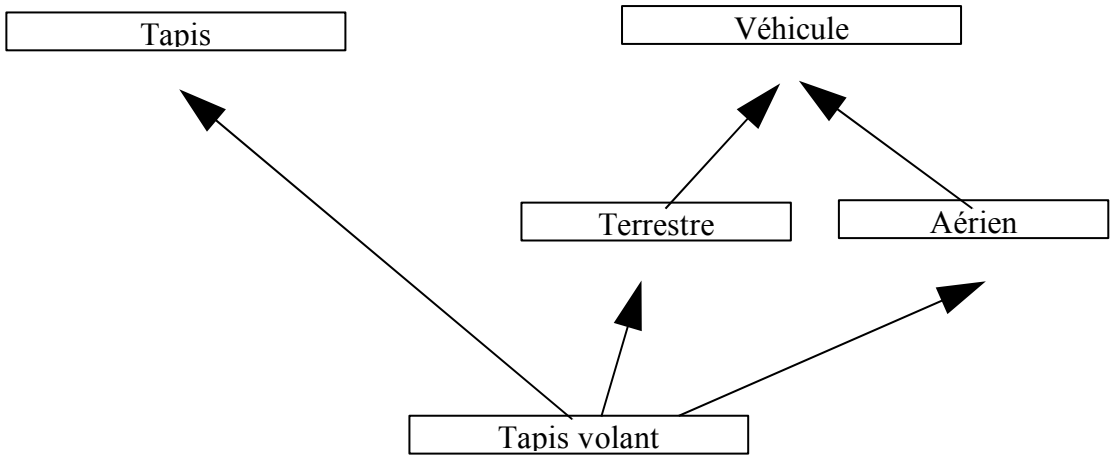


La classe Personne est appelée **super classe** et les classes Elève et Enseignant sont des **sous classes**.

Un autre exemple de hiérarchie Gén-Spéc :

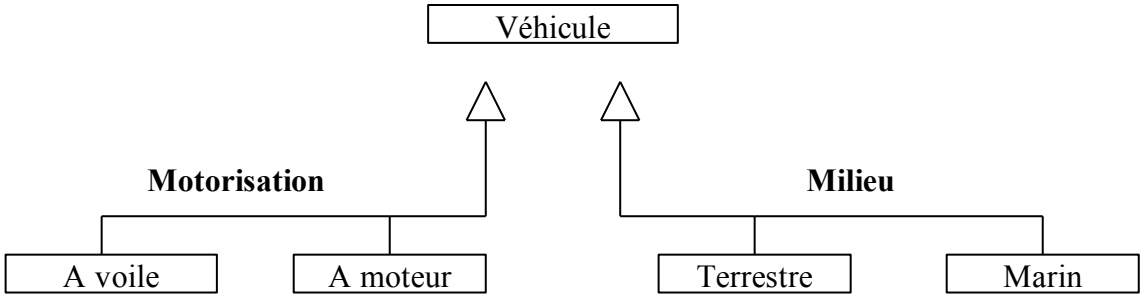


Exemple d'héritage multiple :



Une classe peut être spécialisée selon plusieurs critères simultanément. Dans ce cas, chaque critère est indiqué dans le diagramme :

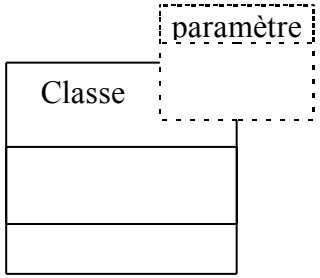
Exemple :



Classes paramétrables

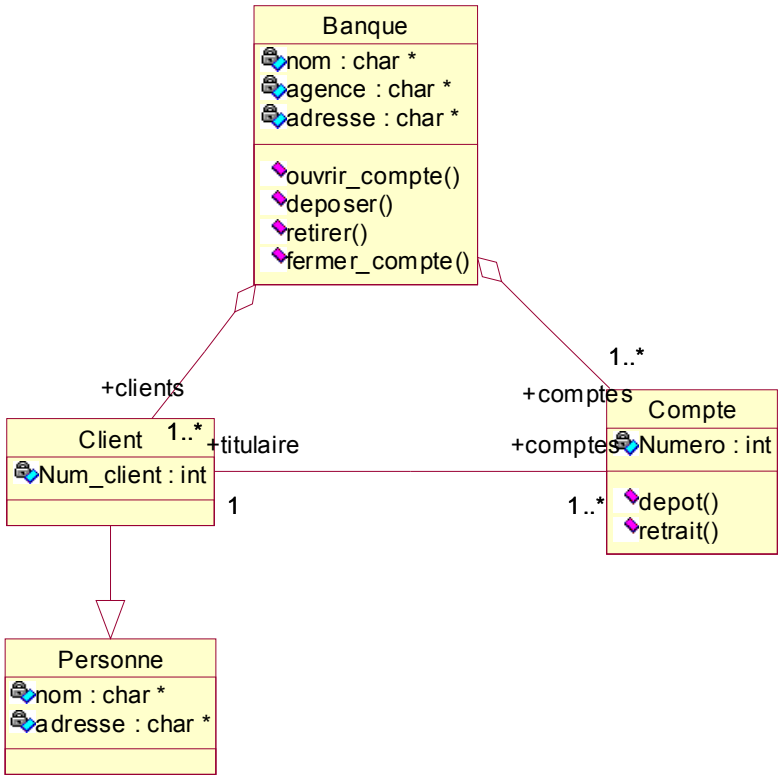
Dans le cas des classes génériques, une classe peut avoir un paramètre :

Une classe paramétrable se présente sous la forme de :



Un exemple : Banque

Modélisation d'une banque, ses clients et ses comptes. Un client est une sorte de Personne et peut avoir plusieurs comptes.



Un autre exemple : Société

Une entreprise emploie des Personnes.

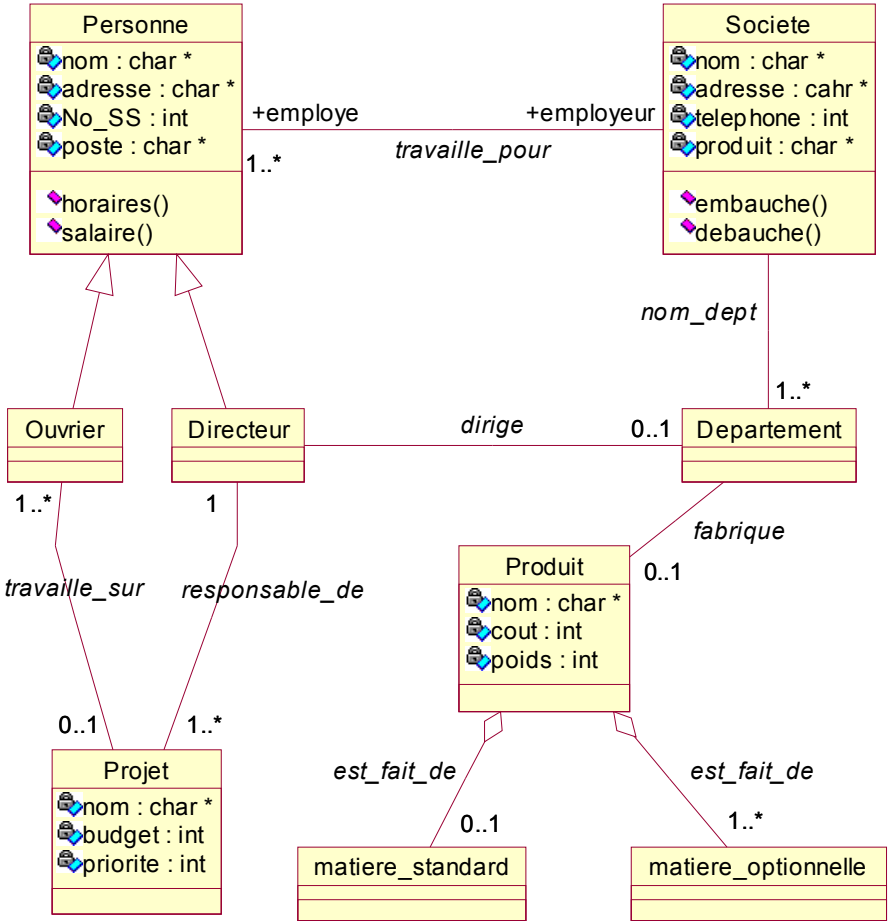
Elle est divisée en Départements et chaque Département est responsable de fabrication de 0 ou 1 Produit.

Les Produits sont fabriqués par des matières Standards et/ou matières optionnelles.

Les employés sont divisés En ouvrier ou en Directeur.

Un ouvrier participe éventuellement à un Projet.

Un Projet est dirigé par un Directeur.


Personne
nom : char *
adresse : char *
No_SS : int
poste : char *
horaires()
salaire()
+employe
1..*
travaille_pour
+employeur
Societe
nom : char *
adresse : cahr *
telephone : int
produit : char *
embauche()
debauche()
nom_dept
1..*
Ouvrier
Directeur
dirige
0..1
Departement
1..*
1
travaille_sur
responsable_de
fabrique
0..1
Produit
nom : char *
cout : int
poids : int
0..1
1..*
Projet
nom : char *
budget : int
priorite : int
est_fait_de
0..1
matiere_standard
est_fait_de
1..*
matiere_optionnelle

Génération de code

Des outils de spécification tels que "Rational Rose" ou "wclass" permettent de spécifier une application et de générer automatiquement le code pour ces spécifications.

Nous allons étudier le code généré pour les diagrammes de spécification vus ci-dessus. A chaque classe dU diagramme correspond une séquence de code générée. La combinaison de ces séquences donnera une application. L'utilisateur doit néanmoins compléter les fichiers générés, en particulier pour les fonctions membres qu'il définit.

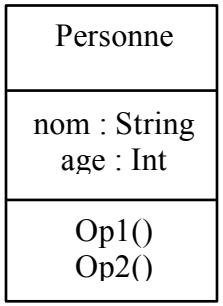
Les outils de génération de code automatiques génèrent, pour une classe X, l'ensemble des constructeurs, destructeur, copie - constructeur et certains opérateurs (=, ==, !=, ..). Les fonctions membres déclarées sont également prises en compte.

Le code généré tient également compte des associations, agrégations, ...

Ci-dessous, nous examinons le code habituellement généré par les éditeurs de spécification objets.

La classe *Personne* utilise la classe prédéfinie string (voir cours).

Classes



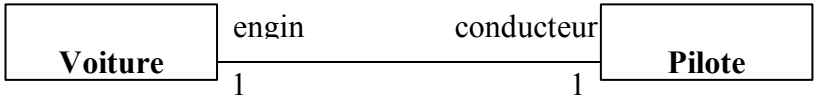

```
// Le fichier Personne.hpp généré automatiquement
class Personne
{string nom;
  int age;
  public :
        Personne();                                  // constructeur
        Personne(const Personne &right);             // copie
constructeur
        ~Personne();                                 // destructeur
        const Personne & operator=(const Personne &right);
        int operator==(const Personne &right) const;
        int operator!=(const Personne &right) const;
        void op1();
        void op2();
        const String get_nom() const;
        void set_nom(const String value);
        const int get_age() const;
        void set_age(const int value);
};
```

Remarque : Le générateur de code produit également un fichier "Personne.cpp" qui contient le code des fonctions déclarées dans "Personne.hpp". Dans certains générateurs de codes, les corps de ces fonctions sont laissés vides (l'utilisateur doit les compléter) alors que d'autres proposent des fonctions complètes.

Association 1 vers 1

Une association 1 vers 1 est réalisée à l'aide des pointeurs placés dans les parties privées des classes qui participent à l'association.

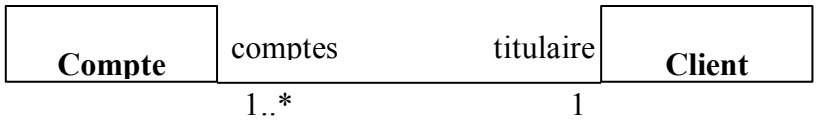
Dans l'exemple ci-dessous, "engin" est le rôle du coté voiture et "conducteur" est le rôle du coté Pilote. La multiplicité de l'association est 1 vers 1.




```
class Voiture
{ public :
    const Pilote * get_conducteur() const;
    void set_conducteur(Pilote * value);
 private :
    Pilote * conducteur;
};
```

```
class Pilote
{public :
    const Voiture * get_engin() const;
    void set_engin(Voiture * value);
private :
    Voiture * engin;
};
```

Association 1 vers N (ou N vers 1)



Le générateur de code réalise l'association par des pointeurs placés dans les parties privées des classes qui participent à l'association. La multiplicité 1..* (ou 1..*) est réalisée par un ensemble de taille non contrainte de pointeurs (type *Ensemble_Infini_de_pointeurs<X>* : un ensemble de taille non limitée de pointeurs sur X où X est un type). Le type ensemble est un type non ordonné.

```
class Client
{ ......
 private :
      Ensemble_Infini_de_Pointeurs<Compte>  comptes;
 public :
      const Ensemble_Infini_de_Pointeurs<Compte> get_comptes () const;
      void set_comptes (Ensemble_Infini_de_Pointeurs<Compte> value);
      ....
```

```
};


class Compte
{ Client * titulaire;
 public :
        const Client * get_titulaire() const;
        void set_titulaire (Client * value);
        ....
};
```

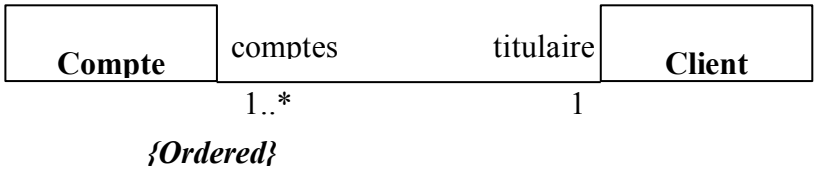
Remarques :

- Dans le cas de la classe *Client* ci-dessus, certains générateurs de codes utilisent le type prédéfini *Liste* à la place d'ensemble lorsque le type *Ensemble* n'est pas disponible. Le type *Liste* est cependant utilisé pour une collection ordonnée de taille illimitée.

- Grâce à la possibilité de création de types génériques (ou template) en C++ (voir plus loin), le types Ensemble ou Liste utilisé ci-dessus est en fait un type (classe) paramétré. Une classe paramétrée est un modèle de classes. Pour cela, on définit par exemple une classe "Liste de X" (notée Liste<X>) qui implante les opérations habituelles sur les listes sans connaître le type X de ses éléments. On instancie ensuite X par un type, par exemple le type entier (noté Liste<int>) et on obtient une liste d'entiers. Ainsi, en précisant un type pour X, on peut disposer d'une liste contenant n'importe quelle autre classe (par exemple, Liste<Personne>, Liste<Compte>, Ensemble<Voiture>, ...)

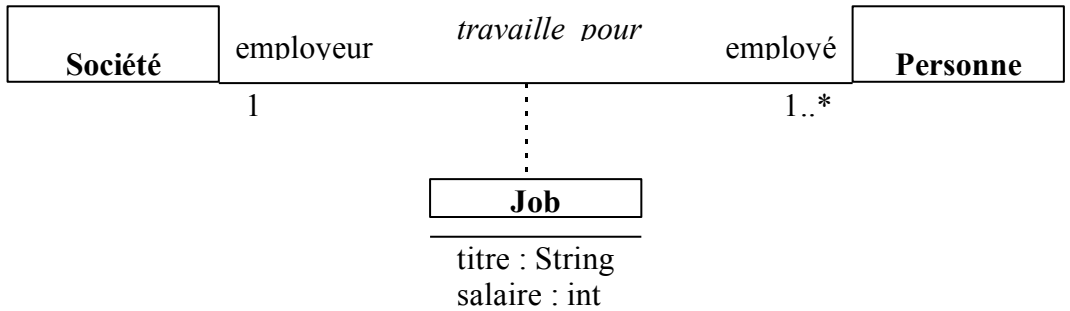
Pour les templates en C++, voir le cours plus loin.

Association 1 vers N avec une contrainte



Dans cet exemple, on impose la contrainte {Ordered} à l'association au niveau du rôle "comptes". Le code généré est tout à fait semblable à l'exemple précédent. La seule différence est que la classe Client doit contenir une collection ordonnée de "Compte". Ceci est implanté par l'utilisation d'une Liste à la place de l'ensemble dans la classe Client : *Liste_Infini_de_pointeurs<Compte>*.

Classe - association



```
class Job
{private:
        String titre;
        int salaire;
        Personne *employé;
        Société *employeur;
 public :
        // fonctions get & set employé
        // fonctions get & set employeur
        // fonctions get & set titre
        // fonctions get & set salaire
};
```

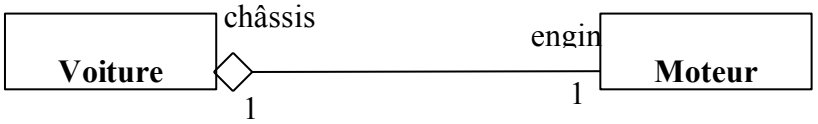
```
class Société
{private :
        UnboundedSetByReference<Job> travaille_pour;
 public :
        const UnboundedSetByReference<Job> get_travaille_pour() const;
        void set_travaille_pour (UnboundedSetByReference<Job> value);
};
class Personne
{private :
        Job * travaille_pour;
 public :
        const Job * get_travaille_pour() const;
        void set_travaille_pour(Job * value);


};
```

On remarque que dans le cas d'une classe - association, la relation entre les classes Société et Personne devient indirecte et passe par la classe Job.

Agrégation

Plusieurs types d'agrégation selon la multiplicité.

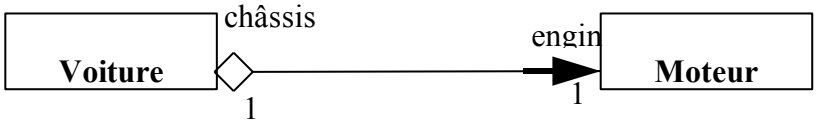
Agrégation 1 vers 1




```
class Voiture                          class Moteur
{Moteur * engin;                           {Voiture * chassis;
 public :                              public :
     const Moteur * get_engin() const;         const Voiture * get_chassis();
     void set_engin(Moteur * value);           void set_chassis(Voiture * value);
     ....                                  ....
};                                     };
```

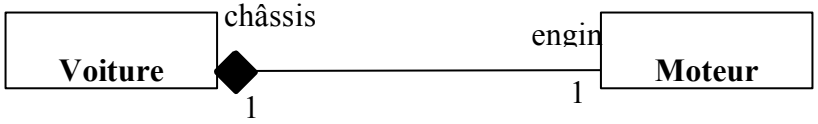
Agrégation à navigabilité restreinte

Si l'agrégation est à navigabilité restreinte comme dans l'exemple :



dans ce cas, la classe Moteur ne contiendra pas l'attribut Voiture et les fonctions get & set de cet attribut disparaissent.

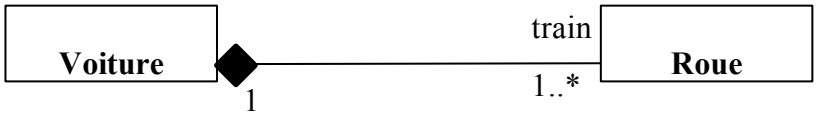
Agrégation par valeur (composition)



La seule différence est que dans la classe agrégat (ici Voiture), l'attribut Moteur (le composant) est représenté par valeur au lieu de référence. Il n'y aura pas de changement dans la classe composant (ici Moteur).

```
class Voiture                                   class Moteur
{Moteur  engin;                                 {Voiture * chassis;
 public :                                       public :
     const Moteur  get_engin() const;               const Voiture * get_chassis();
     void set_engin(const Moteur  value);           void set_chassis(Voiture * value);
     ....                                           ....
};                                              };
```

Agrégation par valeur 1 vers N



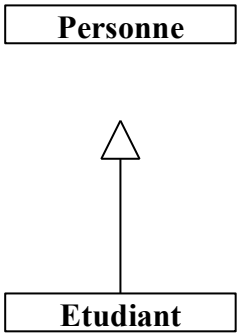
Il n'y aura pas de changement dans la classe composant (Roue). Par contre, la classe composée (agrégat) change et on utilise un ensemble infini par valeur de composants :

```
class Voiture
{Ensemble_Infini_par_Valeur<Roue>  train;
public :
    const Ensemble_Infini_par_Valeur<Roue> get_train() const;
    void set_train(const Ensemble_Infini_par_Valeur<Roue> value);
    ....
};
```

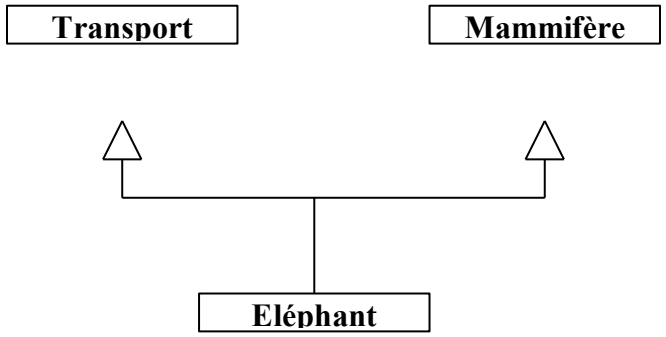
Le type Ensemble_Infini_par_Valeur<Roue> est un type générique (template) et représente un ensemble non ordonné infini de Roue (par opposition à un ensemble de pointeurs sur Roue).

Héritage

On peut avoir deux types d'héritages : simple et multiple.

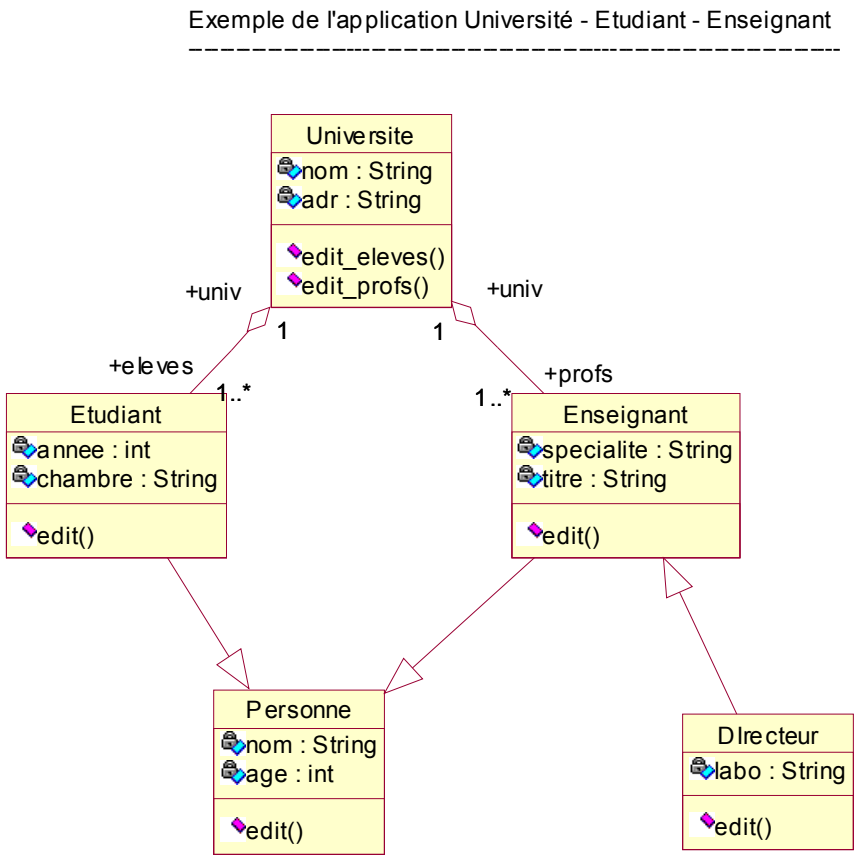



```
class Etudiant : public Personne
{
    .....
};
```




```
class Eléphant : public Transport, public Mammifère
{
      .....
};
```

Exemple


Exemple de l'application Université - Etudiant - Enseignant
Universite
nom : String
adr : String
edit_eleves()
edit_profs()
+univ
1
+univ
1
+eleves
1..*
+profs
1..*
Etudiant
annee : int
chambre : String
edit()
Enseignant
specialite : String
titre : String
edit()
Personne
nom : String
age : int
edit()
DIrecteur
labo : String
edit()

Templates (fonctions et classes génériques)

Les *Templates* permettent de définir des fonctions capables d'effectuer des opérations sur des types de données qui seront définis au moment de l'utilisation.

Par exemple, on peut envisager la fonction *max* capable de calculer le maximum de deux valeurs V1 et V2 de type T :

```
T max(T v1, T v2)
{if (v1 > v2) return v1 ;
 else retuen v2;
}
```

Pour que cette fonction soit utilisable, il faut que l'opérateur '>' soit défini sur le type T. La fonction *max* pourra alors admettre n'importe quel couple de paramètres et calculer leur maximum.

De même, on peut envisager une fonction de tri de vecteur de données qui accepte des vecteurs de différents types.

Habituellement (particulièrement en C), le problème de définir une fonction qui réalise un calcul sur des types de données différentes est résolu par :

1- définition de plusieurs fonctions ou la surcharge de fonction. L'inconvénient dans ce cas est la gestion et la maintenance de ces fonctions.

2- définition d'une fonction avec des arguments pointeurs génériques. L'inconvénient est l'absence de contrôle de types et la conversion des pointeurs est à la charge de l'utilisateur.

D'autres solutions comportant des inconvénients peuvent être envisagées.

La solution proposée par les Templates en C++ permet de définir un modèle de fonctions.

Fonctions génériques

Exemple de définition en C++ :

```
template <class T>
T max(T x, T y)   {return (x > y ? x : y ; }


class thing
{       .....
        int operator>(const thing t1, const thing t2) { .... }
};


void main()
{thing a, b, c;
 int x, y, z;
 a = max(b, c);          // appel de la fonction "max" avec des paramètres de type "thing"
 x = max(y, z);          //          ==                    ==          "int"
}
```

Dans cet exemple, la fonction *max* accepte deux paramètres de n'importe quel type et calcule leur maximum.

La présence de la définition de l'opérateur '>' pour le types utilisés est importante. La classe "thing" ci-dessus le définit et le type prédéfini de base "int" dispose de cet opérateur.

La différence entre la fonction générique *max* et la surcharge de la même fonction est évidente : pour surcharger *max*, il faut connaître tous les types qui peuvent figurer en paramètre de max. Ce qui n'est pas le cas pour la fonction générique.

Le type T est un type formel, on peut remplacer T par n'importe quel autre nom mais le même nom doit apparaître aux mêmes endroits que T. T est appelé le paramètre générique. Le type effectif de T sera précisé lors de l'utilisation de la fonction *max* comme dans la fonction *main* ci-dessus.

On peut avoir plusieurs paramètre générique :

```
template <class U, class V>
int f(U a, V b, ...) { ... }
```

En plus de la définition générique, on peut évidement continuer à utiliser le nom *max* pour définir d'autres fonctions ou d'en surcharger d'autres.

Par exemple, en plus de la définition de max ci-dessus, on peut avoir la définition explicite :

```
char * max(char * x, char * y)
{return (strcmp(x, y) > 0) ? x ; y; }
```

Lors de l'appel de la fonction "max", il y a d'abord une recherche parmi les définitions explicites et/ou surchargées (les définitions non génériques). La recherche d'une fonction générique est entamée si les définitions explicites ou surchargées ne conviennent pas.

Classes génériques

Un utilisateur peut avoir besoin de manipuler des structures de données s'appuyant sur des types différents : par exemple, il aura besoin d'une pile d'entiers, de réels, de char, ...

Les classes génériques permettent de réaliser des structures complexes manipulant des types de bases différents.

Exemple de la classe Pile générique :

```
template <class T> class Pile              // classe Pile générique
{ T * contenu;
  int taille;
  int top;
  public :
        Pile(int t) : taille(t), top(-1) {contenu = new T[t]; }
        ~Pile() {delete [] contenu;}
        void empiler(const T &e) {contenu[++top] = e;}
        T & depiler() {return contenu[top—];}
};
```

```
int main()
{Pile<int> Pi(5); Pi.empiler(3); ..          // définition d'une pile de 5 entiers
 Pile<char> Pc(10); Pc.empiler('x'); ....  // définition d'une pile de 10 caractères
}
```

Remarque : Il est possible d'alléger les notations à l'aide de *typedef* :

```
typedef Pile<double> Pile_dble;


Pile_dble Pd(10);
Pd.empiler(3.5);
.....
```

De même, à l'intérieure d'une classe générique :

```
template <class T> class thing
{...
 public :
 typedef T T;              // définit thing<T>::T
};
```

Les méthodes non inline d'une classe générique utilisent une syntaxe assez lourde. Par exemple, pour définir la fonction *empiler* de la classe Pile générique ci-dessus en dehors de la classe :

```
template <class T> void Pile<T>::empiler(const T& e)
        {...}
```

Une classe générique peut être paramétrée par des types et par des variables. Par exemple, on peut définir la classe Pile générique en précisant un paramètre qui fixe la taille de la pile :

```
template <class T, int length> class Pile          // length sera la taille de la pile
{ T * contenu;
  int taille;
  int top;
  public :
      Pile() : taille(length), top(-1) {contenu = new T[t]; }
      ~Pile() {delete [] contenu;}
      void empiler(const T &e) {contenu[++top] = e;}
      T & depiler() {return contenu[top—];}
};
```

```
int  main()
{Pile<int, 5> Pi; Pi.empiler(3); ..          // définition d'une pile de 5 entiers
 Pile<char,10> Pc; Pc.empiler('x'); ....   // définition d'une pile de 10 caractères
}
```

Dans certains cas, une classe générique peut ne pas convenir à certains types. Par exemple, on aurait certaines difficultés dans le cas d'une pile de char * (problème d'affectation de chaînes, problème de comparaison de chaînes, ...).

Dans ce cas, une solution consiste à définir une instance spécifique de la classe Pile :

```
class Pile<char *>                  // définition spécifique de Pile de chaînes
{
      .....
      // les mêmes opérations mais en utilisant strcpy, strcmp, ..... pour les chaînes
};
```

Dans ce cas, lors d'une instanciation, les instance spécifiques sont d'abord prises en compte avant d'effectuer des instanciations automatiques :

```
void main()
{Pile <char *> Pch(15);      // une pile de 15 chaînes de caractères
 Pch.empiler("Galata"); ...
}
```

Ici, l'instance Pch est définie à l'aide de la définition spécifique de la Pile pour char*.

Une autre solution à ce problème consiste à traiter le cas particulier de l'affectation. En effet, on constate que la seule différence entre la classe Pile générique ci-dessus et la version spécifique pour char* est dans la fonction *empiler* et plus particulièrement dans l'affectation *contenu[++top] = e*.

On peut donc définir une fonction générique appelée *affect* et remplacer l'affectation ci-dessus par un appel à *affect*. Pour traiter le cas de *char**, on définira une version spécifique de la fonction *affect* pour *char**.

L'exemple de la page suivante contient la fonction *affet*.

Remarque : le langage C (et C++) effectue une conversion implicite lors de l'appel d'une fonction. Par exemple, avec :

```
void f(int a, int b) { ....}
```

et l'appel :

```
f(2.5, 6.7);
```

il y a une conversion des réels en entiers avant l'appel de la fonction f.

Par contre, dans le cas des fonctions génériques, il n'y a pas de conversion implicite.

Exemple-1 :

Considérons la fonction générique *max* vue précédemment et l'exemple d'utilisation suivante :

```
int A=1; char B='a'; int C=max(A, B);          // problème de conversion
```

Cet appel à *max* pose problème car il n'y aura pas de conversion implicite de *char* en *int*.

Il faut appliquer une conversion explicite sur la variable B :

```
int A=1; char B='a'; int C=max(A, (int) B);          // conversion explicite
```

Exemple-2 :

soit la fonction générique sqrt :

```
template <class T> T sqrt(T) {...}
```

et la fonction *f* ci-dessous déclare :

```
void f(int i, double d, complexe c)
{complexe z1 = sqrt(i);                    // sqrt(int)
  complexe z1 = sqrt(d);                   // sqrt(double)
  complexe z1 = sqrt(c);                   // sqrt(complexe)
}
```

Si l'utilisateur fait par exemple un appel à *sqrt(double)* avec un argument *int*, une conversion explicite doit être utilisée :

```
void f(int i, double d, complexe c)
{complexe z1 = sqrt((double) i);           // sqrt(double)
  complexe z1 = sqrt(d);                   // sqrt(double)
  complexe z1 = sqrt(c);                   // sqrt(complexe)
}
```

Un exemple

Soit la classe générique Vecteur :

```
template <class T> class Vecteur
{int nb_elements,max_elements;
  T * contenu;              // contiendra les éléments du vecteur
 public :
       Vecteur(int n=0) : nb_elements(0), max_elements(n)
              {if (n==0) contenu=NULL; else contenu=new T[n];}
       void ajouter(const T& e)
              {if (nb_elements < max_elements)
                     affect(contenu[nb_elements++],e);
              }
       int size() {return nb_elements;}
       T& operator[](int i)            // renvoyer le ième élément
              {assert (i>=0 && i<nb_elements);
               return contenu[i];
              }
```

```
    friend ostream& operator<<(ostream & f, Vecteur<T> & V)
    {f << '<'<<V.nb_elements<<'>';
     for (int i=0; i< V.nb_elements ; i++) f << V.contenu[i] << ", ";
     return f<<endl;
    }
};
```

La fonction **affect** permet une simple affectation dans le cas des types et classes ayant défini l'affectation (opérateur =). Par contre, pour les chaînes de caractères, affect utilisera *strcpy* avec une allocation pour l'opérande gauche.

```
template <class T> void affect(T& g, const T& d)    // pour les types ayant l'opérateur '='
    {g=d;}
```

Pour le type char * :

```
void affect(char * &g, const char * d)                    // pour char *
    {g=new char[strlen(d)+1]; strcpy(g,d);}
```

Pour les besoins de comparaison des éléments, on définit :

```
template <class T> class Comparacteur
{public :
      static int plus_petit( T& a,  T& b)  {return (a<b);}          // pour static, voir plus loin
};
```

Et pour permettre la comparaison des chaînes de caractères, on définira la version spécifique :

```
class Comparacteur<char *>
{public :
      static int plus_petit(const char* a, const char* b)
            {return (strcmp(a,b) < 0); }
};
```

Et enfin la classe des vecteurs triables (on peut les trier dans l'ordre croissant) est dérivée de la classe Vecteur et de la classe Comparateur. L'héritage de Comparateur permet de disposer de l'opération "plus_petit" :

```
template <class T> class Vecteur_triable :          // héritage double
        public Vecteur<T>, public Comparacteur<T>
{public :
    Vecteur_triable(int s) : Vecteur<T>(s)          // constructeur
        {}
};
```

On peut définir la fonction générique de tri des vecteurs par (méthode Bulle) :

```
template <class T> void tri(Vecteur_triable<T> & V)
{int n=V.size();
 for (int i=0; i< n-1; i++)                         // tri par la méthode bulle
    for (int j=n-1; i<j; j--)
        if (V.plus_petit(V[j], V[j-1]))             // échanger les 2 éléments
            echange(V[j], V[j-1]);
}
```

La fonction échange permet de permuter deux éléments existants. Le fait que ces éléments existent nous empêche d'utiliser la fonction affect ci-dessus car affect, dans sa version des char* alloue une zone de mémoire pour son opérande gauche supposé inexistant.

```
template <class T> void echange(T& g,  T& d)          // échange générique
      {T temp=g; g=d;d=temp;}
```

Pour char * :

```
void echange(char * &g, char *& d)                    // échange de deux chaînes
{char * temp=new char[strlen(d)+1]; strcpy(temp,g);   // ou affect(temp, g);
  strcpy(g,d); strcpy(d,temp);
}
```

Un exemple d'utilisation : vecteur d'entiers, vecteur de char *, vecteur de String, ...

Rappel : ici, seul le type char * nécessite des versions spécifiques d'affectation (fonction affect), d'échange (fonction echange) et de la classe spécifique Comparateur (pour l'opérateur '<').

Les autres classes (y compris les classes de bases) disposent de ces opérateurs.

```
void main()
{Vecteur_triable<int> Vi(5);
 Vi.ajouter(15);Vi.ajouter(25);Vi.ajouter(5); Vi.ajouter(2);Vi.ajouter(30); cout << Vi;
 tri(Vi);cout << Vi;

 Vecteur_triable<char *> Vch(5);
 Vch.ajouter("toto"); Vch.ajouter("tata"); Vch.ajouter("titi"); cout << Vch;
 tri(Vch); cout << Vch;

 Vecteur_triable<String> Vstr(5);
 Vstr.ajouter("blabla"); Vstr.ajouter("ratata");  Vstr.ajouter("blobert"); cout << Vstr;
 tri(Vstr); cout << Vstr;
}
```

Remarques :

Classes génériques et fonctions amies :

Une fonction amie (friend) qui fait référence à un paramètre générique doit elle aussi être générique.

Une fonction amie générique n'est amie que des instances de la classe générique qui ont les mêmes paramètres génériques.

Classe générique et attribut de classe :

Les attributs de classe déclarés avec le mot clé **static** sont communs à tous les objets instances d'une instance particulière d'une classe générique.

Dans l'exemple ci-dessus, on a déclaré static la fonction **plus_petit**. Elle sera commune à tous les objets instances d'une instance particulière, par exemple **Comparateur<int>** (de manière directe) ou de **Vecteur_triable<double>** (par héritage). En d'autres termes, tous les vecteurs triables de doubles partagent la même fonction.

Exemple : Vecteur générique

L'exemple ci-dessous définit un vecteur generique. Pour simplifier, nous ne traitons pas le cas de *char**.

L'utilisateur pourra utiliser la classe String (défini en cours) à la place de char *.

Dans cette solution, les insertions ne sont pas ordonnées.

```
#include <iostream.h>
#include "String.hpp"

#define N 5                 // l'incrément N, 2N, 3N, ...

template <class T> class vecteur
{int nb_elements;                   // nombre d'éléments dans vecteur
 int max_elements;              // nombre maximum possible d'éléments (N, 2N, 3N..)
 T* tab;
 public:
 vecteur()
```

```
    {max_elements=N; nb_elements=0;
     tab=new T[N];
    }

vecteur(const vecteur<T>& V)
    {max_elements=V.max_elements;nb_elements=V.nb_elements;
     tab=new T[max_elements];
     for (int i=0; i<nb_elements; i++)        tab[i]=V.tab[i];
    }

vecteur<T>& operator=(const vecteur<T>& V)
    {if (this == &V) return *this;
     this->~vecteur();
     max_elements=V.max_elements; nb_elements=V.nb_elements;
     tab=new T[V.max_elements];
     for (int i=0; i<nb_elements; i++)        tab[i]=V.tab[i];
     return *this;
    }
```

```
int operator==(const vecteur<T>& V)
      {if (nb_elements != V.nb_elements) return 0;
       for (int i=0;  i<nb_elements; i++)
            if(tab[i] != V.tab[i]) return 0;
      return 1;
      }

int operator!=(const vecteur<T>& V)
      {return !(*this == V);}

vecteur<T>& operator+=(const T& ch)
{if (nb_elements < max_elements)
      {tab[nb_elements]=ch;
       nb_elements++; return *this;
      }
 else
      {int i;
```

```
        T* t1=new T[max_elements+N];
        for ( i=0; i<nb_elements; i++)      t1[i]=tab[i];
        this->~vecteur();
        t1[nb_elements]=ch;
        tab=t1;  nb_elements++; max_elements+=N;
        }
 return *this;
}

vecteur<T> operator+(const T& ch)
{vecteur<T> V=*this;   V += ch;   return V;}

const T& operator[](int i) const
{if ((i<0) ||(i>=nb_elements)) return NULL;
 return tab[i];
}

/* si T = int, il y aura une ambiguite par rapport a l'operateur [int]
```

```
int operator[](const T& ch) const          // renvoie l'indice de ch
{for (int i=0; i<nb_elements; i++)                // renvoie -1 si non trouve'
       if (tab[i] == ch) return i;
 return -1;
}
*/

vecteur<T> operator+(const vecteur<T> & V)              // FUSION
 {vecteur<T> V1=*this;
  for (int i=0; i<V.nb_elements; i++) V1 += V[i];          // ou V.tab[i]
  return V1;
  }

~vecteur()         {delete [] tab; }

friend ostream& operator<<(ostream& O, const vecteur<T> & V)
{O << "<" << V.max_elements <<", "<<V.nb_elements<<"> ";
  for (int i=0; i<V.nb_elements; i++) O<<V.tab[i] << " ";
```

```
  return O << endl;
 }

 friend istream& operator>>(istream& Is, vecteur<T> & V)
 {cout << "donner " << V.max_elements << " elements, 1 par ligne \n";
  T ch;
  for (int i=0; i<V.max_elements; i++) {Is >> ch; V += ch;}
  return Is;
 }

};
void main()
{vecteur<int> Vi1; Vi1 += 12; cout << Vi1;
 vecteur<String> V1;
 V1 += "abc";
 V1 += "def";
 V1 += "jkl";
 V1 += "mnp";
```

```
V1 += "opq"; cout << V1;


vecteur<String> V2=V1; cout << V2 ;

vecteur<String> V3; V3=V1; cout << V3;

cout << V1 + "xyz" ;

vecteur<String> V4; V4 = V1 + "uvw" + "xyz" ;

cout << V4;

V1 = V1 + "123"; cout << V1;

if (V1 == V4) cout << "egalite de V1 et V4";

else cout << "difference de V1 et V4";

cout << endl;

cout << "le 3eme element de V1 est = " << V1[3] << endl;

cout << "la fusion de V1 et V3 donne" << V1+V3 << endl;

vecteur<String> V5; cin >> V5; cout << V5;

}
```