TABLE DES MATIERES

1. Objectif

Ce document décrit une approche du C qui n'est pas très académique. En effet, dés que l'on parle de langages objets, on se tourne vers le Java ou le C++. Le C devient ringard et peu adapté à ses concepts. Or, il est parfaitement possible d'adapter le C pour développer selon les concepts Objets.

Ce document décrit un standard de programmation (permet de réaliser un développement homogène), puis la partie suivante décrit les règles de passage d'un modèle objet (comme UML) à un développement en C.

2. Règles de programmation

2.1 Règle de nommage

2.1.1 Les fichiers sources

Le nom d' un fichier source doit être significatif du contenu.

2.1.2 Les noms des variables

2.1.2.1 Les variables locales

Le nom d'une variable doit définir son contenu.
La première lettre doit être en majuscule et les autres en minuscule.
Tous les mots composants une variable sont séparés par '_'.

Exemple : `Nom_voiture, Nom_magasin, Nb_contraintes,...`

2.1.2.2 Les variables globales

Il est interdit d'avoir des variables globales.

2.1.2.3 Les noms de variables interdites

Les variables nommées ' temp' , ' prov' , … sont à proscrire ainsi que les variables proches des mots clés comme ' For' , ' Socket' , ' New' , ...

2.1.3 Les noms des constantes

Le nom d'une constante doit définir son contenu.
Toutes les lettres sont en MAJUSCULE.
Tous les mots composants une constante sont séparés par '_'.

Exemple : `TAILLE_MAXI, ...`

2.1.4 Les macro-commandes

Le nom d'une macro-commande est en MAJUSCULE.
Tous les mots composants une macro-commande sont séparés par '_'.
Elles permettent de simplifier une séquence de traitement.

Exemple :

```
EST_NUMERIQUE(a)        (('0'<=a)&&(a<='9'))
EST_CORRECTE(Valeur)    (Valeur!=(t_Valeur *)NULL)
EXISTE(Valeur)          (Valeur!=(t_Valeur *)NULL)
```

2.1.5 Les noms d'une fonction et d'une procédure

2.1.5.1 Une action

Le nom d'une fonction ou d'une procédure correspondant à une action doit commencer par un verbe à l'infinitif.

La première lettre est en majuscule.

Toutes les autres lettres sont en minuscule.

Tous les mots composants le nom d'une fonction ou d'une procédure sont séparés par '_'.

Exemple : `Traiter_demande()`

2.1.5.2 Une vérification

Le nom d'une fonction ou d'une procédure correspondant à une action doit commencer par un verbe conjugué au présent.

La première lettre est en majuscule.

Toutes les autres lettres sont en minuscule.

Tous les mots composants le nom d'une fonction ou d'une procédure sont séparés par '_'.

Exemple : `Est_entier()`

2.1.5.3 Les noms des paramètres

Les noms des paramètres respectent la règle des noms de variables décrit ci-dessus.

Ils sont cependant préfixés par :

- 'i_' pour un paramètre d'entrées,
- 'o_' pour un paramètre de sorties,
- 'io_' pour un paramètre d'entrée -sortie.

Exemple : `i_Indice, io_Information, o_Resultat, …`

2.1.5.4 Les procédures et les fonctions sans aucun paramètre

Ces procédures et ces fonctions doivent être suffixées par (void).

Exemple : void Ma_procedure(void);

2.1.6 Les index de boucles

Nous autorisons l'utilisation des variables i, j et k pour les index de boucles.

2.1.7 Les noms des types

2.1.7.1 Les types structurés

Les noms des types structurés respectent la règle des noms de variables décrit ci-dessus. Ils sont préfixés par 't_'.

Exemple : `t_Vehicule`

2.1.7.2 Les types énumérés

Les noms des types énumérés respectent la règle des noms de variables décrit ci-dessus. Ils sont préfixés par 'e_'.

Les valeurs du type énuméré sont écrites comme les constantes.

Il faut utiliser ces types aussi souvent que possible.

Exemple : `enum { ROUGE, VERT, BLEU } e_Couleur`

2.1.7.3 Les types unions

Les types unions ne doivent pas être utilisés, cela nuit à la lisibilité du code.

2.1.8 Les abréviations

Elles sont tolérées dans la limite des abréviations communes.

Exemple :
```
Nb : Nombre , Cpt : Compteur, Maxi : Maximum,
Ref : Reference, Mini : Minimum...
```

2.2 Présentation du code

2.2.1 Règles générales

Le programme doit être aéré (ne pas être avare sur les sauts de ligne).

Les différentes parties d' une fonction ou d' une procédure sont indiquées par des commentaires explicatifs accompagnés éventuellement d' un algorithme permettant de préciser le fonctionnement.

Les conditions ou les itérations sont décrites dans un commentaire la précédant sauf si le code est suffisamment explicite.

L' indentation doit être d'une tabulation (correspondant à 2 espaces).

Les accolades ouvrantes et fermantes respectives doivent être positionnées à la même colonne (l' une en dessous de l' autre sur la même colonne).

Il faut caster les objets ayant des types différents pour les comparer ou les affecter (***ex***: comparaison d' un entier avec un réel, affectation du résultat d' une allocation ou comparaison d' un pointeur avec NULL).

Ne pas effectuer d'affectations dans une condition.

Ne pas utiliser les instructions ' break' (sauf dans un switch) et ' continue' .

2.2.2 Description d'un header

2.2.2.1 Un entête

L' entête d' un header est une zone de commentaire permettant de décrire brièvement le contenu du module ainsi que les différentes évolutions subies par le module, par qui et quand.

Exemple :

```
/* ------------------------------------------------------------------------ */
/*                     Gestion de la classe t_Information                   */
/* ------------------------------------------------------------------------ */
/* OBJET : Structure de donnees dynamiques permettant de gerer un ensemble  */
/*         d'informations (Cle, Valeur)                                     */
/* ------------------------------------------------------------------------ */
/*     DATE     |   AUTEUR    |             COMMENTAIRES                    */
/* ------------------------------------------------------------------------ */
/*  08/11/1999  |     A.L     | Creation des fichiers                       */
/*  01/12/1999  |     A.L     | Creation de la classe t_Information         */
/*  04/01/2000  |     A.L     | Mise a disposition des outils               */
/*  17/03/2000  |     E.P     | Methodes de lecture ensemble infos          */
/*  09/08/2001  |     A.L     | Extraction, Ajout, Suppression d'une partie */
/* ------------------------------------------------------------------------ */
/* CARACTERISTIQUE TECHNIQUE DU MODULE :                                    */
/*                                                                          */
/* LIMITES : Une ligne dans un fichier texte a lire ou a ecrire ne peut     */
/*           exceder 512 octets (ATTENTION : Effet de bord possible)        */
/*                                                                          */
/* ------------------------------------------------------------------------ */
```

2.2.2.2 L'encadrement

Le header est encadré par une directive qui teste si une macro est déjà définie ou pas. Cet encadrement a pour rôle de ne pas avoir de multiples inclusions d' un header.

Exemple :

```
#IFNDEF _al_<nom fichier>_h
#DEFINE _al_<nom fichier>_h
…
contenu du header
…
#ENDIF
```

2.2.2.3 Les inclusions

Cette zone contient l' ensemble des inclusions nécessaires aux fonctionnements du module.

Exemple :

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

2.2.2.4 Les constantes

Les constantes nécessaires aux types sont définies par DEFINE dans cette partie.

Exemple :

```
#define     TAILLE_MAXI 10
```

2.2.2.5 Les types

Cette zone contient la description de tous les types utilisables.

Exemple :

```
typedef struct t_Pays
{
  char Nom[TAILLE_MAXI+1];
  long Nb_habitants;
} t_Pays;
```

2.2.2.6 Les fonctions et les procédures utilisables

Le header contient les prototypes des différentes fonctions et procédures d' accès au module. Ce sont les outils manipulés par les autres modules. Le header ne contient que les prototypes d' accès au module. La description de son utilisation, la liste des paramètres en entrées et/ou en sortie ainsi que le résultat du traitement doivent figurer dans le fichier source C.

Il faut précéder le prototype de la fonction par la classe "extern".

Exemple :

```
extern int Est_entier(char *i_Chaine);
```

2.2.2.7 Exemple de header

```
#ifndef _al_information_h
#define _al_information_h

/* ------------------------------------------------------------------------ */
/*                     Gestion de la classe t_Information                   */
/* ------------------------------------------------------------------------ */
/* OBJET : Structure de donnees dynamiques permettant de gerer un ensemble  */
/*         d'informations (Cle, Valeur)                                     */
/* ------------------------------------------------------------------------ */
/*     DATE     |   AUTEUR    |             COMMENTAIRES                    */
/* ------------------------------------------------------------------------ */
/*  09/08/2001  |     A.L     | Extraction, Ajout, Suppression d'une partie */
/* ------------------------------------------------------------------------ */
/* CARACTERISTIQUE TECHNIQUE DU MODULE :                                    */
/*                                                                          */
/* LIMITES : Une ligne dans un fichier texte a lire ou a ecrire ne peut     */
/*           exceder 512 octets (ATTENTION : Effet de bord possible)        */
/* ------------------------------------------------------------------------ */

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

```
/* ------------------------------- */
/* definition des messages d'erreurs */
/* ------------------------------- */

#define INF0001 "INF0001: Erreur d'allocation memoire"
…

/*--- DEFINITION DES CONSTANTES ---*/

#DEFINE LONGUEUR_IDENTIFIANT    10

/*--- DEFINITION DES MACROS COMMANDES ---*/

#DEFINE EST_NUMERIQUE(a)  (('0'<=a)&&(a<='9'))

/*--- DEFINITION DES STRUCTURES ---*/

typedef struct t_Information
{
      char Identifiant[LONGUEUR_IDENTIFIANT+1];
      int Valeur;
} t_Information;

typedef t_Liste_generique t_Liste_information;

/*--- DEFINITION DES PROTOTYPES UTILISABLES ---*/

extern int Est_message(char *i_Chaine);

#ENDIF
```

2.2.3 Description d'un fichier source C

2.2.3.1 L' entête du corps

L' entête d'un corps est une zone de commentaire permettant de décrire brièvement le contenu du module et les remarques techniques ainsi que les différentes évolutions subies par le module, par qui et quand.

Exemple :

```
/* ------------------------------------------------------------------------ */
/*                  Gestion de la classe t_Information                      */
/* ------------------------------------------------------------------------ */
/* OBJET : Structure de donnees dynamiques permettant de gerer un ensemble  */
/*         d'informations (Cle, Valeur)                                     */
/* ------------------------------------------------------------------------ */
/*     DATE     |   AUTEUR    |             COMMENTAIRES                    */
/* ------------------------------------------------------------------------ */
/*  08/11/1999  |     A.L     | Creation des fichiers                       */
/*  01/12/1999  |     A.L     | Definition des methodes et de la classe     */
/*              |             |                                       t_Info*/
/*  01/12/1999  |     A.L     | Creation de la classe t_Information         */
/*  04/01/2000  |     A.L     | Mise a disposition des outils               */
/*  23/03/2000  |     A.L     | Remplacement de la classe t_Info par une    */
/*              |             |     structure simplifiee                    */
/*  09/08/2001  |     A.L     | Extraction, Ajout, Suppression d'une partie */
/*              |             |                                             */
/* ------------------------------------------------------------------------ */
```

2.2.3.2 L' inclusion

Le module doit inclure le header qui le caractérise.

Exemple :

```
#include "al_outils.h"
```

2.2.3.3 Description des types internes au module

Les types communs sont décrit dans le header.
Les types propres au fichier source C sont décrits ici. Ces types ne seront exploitables que par le fichier source.

2.2.3.4 Les fonctions et les procédures internes au module (non accessible par d' autres modules)

Cette partie comprend le codage des fonctions et des procédures propres au module. Elle doit comprendre un entête (description des fonctionnalités) qui précise la portée de la fonction ou procédure et un corps (codage C).
Les prototypes doivent être précédés de la classe '**static**'.

Exemple :

```
/* ------------------------------------------------------------------ */
/* DESCRIPTION : Permet de déterminer si la chaine est un entier ou pas */
/* ------------------------------------------------------------------ */
/* ENTREES : Chaine : chaine de caractères à analyser                 */
/* ------------------------------------------------------------------ */
/* SORTIES : Valeur : conversion de la Chaine                         */
/* ------------------------------------------------------------------ */
/* RETOUR  : 0 : Chaine n'est pas un entier, Valeur n'est pas modifiée  */
/*           1 : Chaine est un entier, Valeur contient la conversion    */
/* ------------------------------------------------------------------ */

static int Est_entier(char *i_Chaine,int *o_Valeur)
{
 . . .
}
```

2.2.3.5 Les fonctions et les procédures d' accès au module

Cette partie comprend le codage des fonctions et des procédures dont les prototypes sont décris dans le header. Les prototypes ne sont précédés d' aucune classe d' appartenance.

Exemple :

```
/* ------------------------------------------------------------------ */
/* DESCRIPTION : Permet de déterminer si la chaine est un message     */
/* ------------------------------------------------------------------ */
/* ENTREES : Chaine : chaine de caractères à analyser                 */
/* ------------------------------------------------------------------ */
/* RETOUR  : 0 : Chaine n'est pas un message                          */
/*           1 : Chaine est un message                                */
/* ------------------------------------------------------------------ */

int Est_message(char *i_Chaine)
{
  . . .
}
```

2.2.4 Déclarations des variables

Une variable doit être initialisée dés sa déclaration. Si elle doit prendre des valeurs particulières pour indiquer un sens, ces valeurs doivent être décrites dans un commentaire à côté. Dans ce cas, si le nombre de valeurs pouvant être prises sont dénombrables, il est préférable d'utiliser un type énuméré.

Exemple :

```
int i = 0 ;
t_Vehicule Vehicule = (t_Vehicule *)NULL;/* NULL: pas de véhicules */
t_Individu *Individu=Instancier_individu("LESERT Aymeric") ;
```

2.2.5 Les fonctions et les procédures

2.2.5.1 Ordre d' écriture ou de déclaration des fonctions et des procédures

Il faut *TOUJOURS* spécifier la fonction avant de l' utiliser.

REMARQUE : Dans le cas de récursivité croisée, il faudra déclarer le prototype de la fonction appelée avant la fonction appelante.

2.2.5.2 Les fonctions et les procédures utilisables

Elles comportent 4 parties :

- Déclaration et initialisation des variables
- Vérifie la cohérence des données (programmation défensive)
- Traite les données
- Retourne les résultats

La partie 3 ne doit pas effectuer de « return ».

Exemple :

```
/* ------------------------------------------------------------------- */
/* DESCRIPTION : Permet de déterminer si la chaine est un message      */
/* ------------------------------------------------------------------- */
/* ENTREES : Chaine : chaine de caractères à analyser                  */
/* ------------------------------------------------------------------- */
/* RETOUR  : 0 : Chaine n'est pas un message                           */
/*           1 : Chaine est un message                                 */
/* ------------------------------------------------------------------- */

int Est_message(char *i_Chaine)
{
  /* déclaration et initialisation des variables */

  int Longueur = 0 ;
  int Nb_informations = 0 ;


  /* contrôle de la coherence des données */

  Longueur = strlen(i_Chaine) ;
  if ((0>Longueur)||(Longueur>200))
      return(-1);

  /* traitement */

  . . .

  /* retour */

  return (Nb_informations) ;
}
```

2.2.5.3 Les fonctions et les procédures internes au fichier source

Elles peuvent comporter 3 ou 4 parties :

- Déclaration et initialisation des variables
- Vérifie la cohérence des données (programmation défensive, si nécessaire)
- Traite les données
- Retourne les résultats

La partie 3 ne doit pas à effectuer de « return ».

2.3 Exploitation de la présentation du code

Grâce à la normalisation décrite ci-dessus, il est concevable de mettre en place un outils permettant de réaliser une documentation on-line (au format HTML afin de les consulter avec un browser).

3. Règles de transformation d'un modèle objet en C

3.1 Objectif

Pour modéliser des programmes complexes, la méthodologie objet s'impose aujourd'hui. Cependant, pour des raisons techniques, il est parfois nécessaire de développer en C.

A prime abords, le C n'est pas un langage adapté à une modélisation objet. Mais cela est possible si nous nous équipons de régles de transformation d'un modèle objet en C afin d'avoir une programmation des différents objets homogènes et cohérents.

3.2 Les classes

3.2.1 Propriétés

Les propriétés d'une classe sont un ensemble d'attributs caractérisant un objet (***Ex***: la couleur d'un stylo : couleur est un attribut de l'objet stylo, il est alors aisé d'instancier un stylo bleu ou vert). Nous considérerons que les propriétés sont privées (Accessible uniquement avec des accesseurs). En C, une propriété est un composant d'une structure.

Le nom des propriétés respecte la règle des noms de variables.

Les propriétés doivent être décrites après les méthodes.

3.2.2 Méthodes privées, protégées et publiques

Les méthodes sont un ensemble de fonctions ou de procédures agissant sur un objet. Elles correspondent aux méthodes décrites dans le modèle (***Ex*** : la méthode écrire pour un stylo). En C, nous ne pouvons pas faire la distinction entre une méthode publique, privée ou protégée. Nous considérerons que les méthodes sont publiques. Les méthodes se matérialiseront par des pointeurs sur des fonctions définies en interne dans un module.

Le premier paramètre d'une méthode est un pointeur sur un objet de type de la classe contenant la méthode (elle est nommée « **this** »). Cette pratique permet d'accéder aux propriétés de la classe.

Les méthodes doivent être décrites avant les propriétés.

Chaque méthode a une fonction associée. Une fonction associée a une méthode est nommée avec le nom de la méthode in fixée par le nom de la classe.

Exemple :

Soit la méthode Ecrire de la classe Stylo.
La fonction associée à la méthode se nommera : Ecrire_stylo.

3.2.3 Méthodes obligatoires

3.2.3.1 Le(s) constructeur(s)

Cette méthode permet d' instancier ***proprement*** une classe. Elle **alloue** une nouvelle instance pour laquelle elle initialise les propriétés de la classe et positionne les méthodes sur les bonnes fonctions. Le constructeur se nomme "**Instancier_<*Nom de la classe*>**" (équivalent d'un new)

Cette méthode n'est pas une méthode de l'objet, elle es t une fonction d'accès au fichier source C. Le constructeur doit **obligatoirement** comporter les phases :

- Allocation d'un objet (et teste le bon fonctionnement de l'allocation)
- Affectation des méthodes
- Initialisation des propriétés
- Instanciation des agrégations ou des classes méres
- Retour du nouvel objet alloué

REMARQUE : Il n' est pas interdit d' avoir plusieurs constructeurs. Dans ce cas, il faut préciser la nature du constructeur : "**Instancier_<*Nature*>_<*Nom de la classe*>**". Un constructeur peut éventuellement faire appel à un autre constructeur de la même classe

(***Exemple*** : **Instancier_copie_stylo**)

3.2.3.2 Le destructeur

Cette méthode permet de libérer ***proprement*** une instance. Elle **libère** l'ensembles des objets instanciés dans sa classe et la mémoire allouée lors du constructeur. Elle **initialise** le pointeur sur l'instance à NULL. Le destructeur est une méthode et se nomme "**Liberer**" (équivalent d'un delete).

Cette méthode est une méthode de l'objet. La particularité de cette méthode est que le premier paramètre est un pointeur sur un pointeur d'instance.

Le destructeur doit **obligatoirement** respecter les phases :

- Suppression des associations liées à l'objet instancier
- Libération des objets agrégations instanciés dans la classe
- Libération de l'instance
- Initialisation de l'instance à NULL

REMARQUE : Il ne peut y avoir qu' un seul destructeur par classe.

3.2.3.3 Les accesseurs

Les propriétés d'une classe ne sont accessibles directement que dans les méthodes de la classe. Quand un objet d'une classe est utilisé, il faut accéder a ux propriétés de la classe par des accesseurs.

BUT : Encapsulation maximum (garantie l'intégrité de l'objet).

3.2.4 Les méthodes conseillées

Pour chaque classe, il faudra être capable de :

- instanciation par copie (***Instancier_copie_***<***nom de la classe***>)
- comparer deux instances d' une même classe (***Comparer***)
- comparer une instance avec une valeur autre qu' une instance du même type (***Comparer_valeur***)
- tracer le contenu de l' instance d' une classe : dans le journal interne (***Tracer***) ou à l' écran (***Ecrire***)
- contrôler la cohérence de l' instance (***Est_coherente***)

3.2.5 Les méthodes surchargées d' un objet

Elles comportent 5 parties :

- Déclare et initialise les variables
- Vérifie la cohérence des données (programmation défensive)
- Appelle les méthodes de la classe mère surchargée
- Traite les données
- Retourne les résultats

3.2.6 Les méthodes réécrites d' un objet

Elles comportent 4 parties (comme une fonction et une procédure) :

- Déclare et initialise les variables
- Vérifie la cohérence des données (programmation défensive)
- Traite les données
- Retourne les résultats

3.2.7 Règle de nommage des méthodes dans le fichier source C

Comme il n' est pas possible de décrire les méthodes à l' intérieur d' une classe, il faut définir une fonction ou une procédure par méthode. Le nom de la fonction ou de la procédure est postfixé par le nom de l' objet.

Exemple : Soit la méthode Liberer d' un stylo, la fonction associée sera Liberer_stylo.

3.2.8 Exemple simple

STYLO
Couleur
Ecrire(Texte) : void Lire_Encre() : e_Couleur

définition du type en C dans le header

```
typedef enum { ROUGE, VERT, BLEU }e_Couleur ;

/* --- DESCRIPTION DE LA CLASSE STYLO --- */

typedef struct t_Stylo
{
  /* le destructeur */

  void (*Liberer)(struct t_Stylo **this) ;

  /* les méthodes */

  void (*Ecrire)(struct t_Stylo *this,char *i_Texte);
  e_Couleur (*Lire_encre)(struct t_Stylo *this) ;

  /* une propriété */

  e_Couleur Couleur;
} t_Stylo;

extern t_Stylo *Instancier_stylo(e_Couleur);
```

définition de la classe en C dans le fichier source C

```
/* --- DESCRIPTION DES FONCTIONS ASSOCIEES AUX METHODES --- */

void Ecrire_stylo(t_Stylo *this,char *i_Texte)
{
  Change_couleur(this->Couleur);
  printf("Texte : <%s>\n",i_Texte);
  return ;
}


e_Couleur Lire_encre_stylo(t_Stylo *this)
{
  return(this->Couleur);
}

/* le destructeur */
void Liberer_stylo(t_Stylo **this)
{
  AL_FREE(*this);
  *this=(t_Stylo *)NULL;
  return;
}
```

```
/* le constructeur */
t_Stylo *Instancier_stylo(e_Couleur i_Couleur)
{
  /* allocation d'une instance */

  t_Stylo *this=(t_Stylo *)AL_MALLOC(sizeof(t_Stylo)) ;
  if (this==(t_Stylo *)NULL)
    return((t_Stylo *)NULL) ;

  /* affectation des méthodes */

  this->Ecrire=Ecrire_stylo ;
  this->Lire_encre=Lire_encre_stylo;
  this->Liberer=Liberer_stylo;

  /* initialisation des propriétés */

  this->Couleur=i_Couleur;

  /* retour de l'objet instancié */

  return(this);
}
```

utilisation en C

```
void ma_fonction()
{
  TStylo *Stylo_Bleu=Instancier_stylo (Bleu);
  …
  Stylo_Bleu->Ecrire(Stylo,"Coucou");
  …
  Stylo_Bleu->Liberer(&Stylo);

  return;
}
```

3.3 Les associations

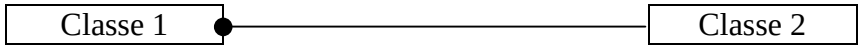
3.3.1 Descriptions

Les associations sont des liens entre 2 (ou plus) d' objets.

3.3.2 Multiplicité d' une association (cardinalité)

3.3.2.1 (Zéro, un ou plusieurs) et (Exactement Un)

Exemple :

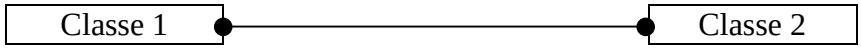


La Classe 1 doit ajouter une propriété qui est une référence (un pointeur) sur **une** instance de la classe 2 (Ce pointeur ne doit pas être NULL).

La classe 2 doit avoir une liste de références sur des objets de la classe 1.

3.3.2.2 (Zéro, un ou plusieurs) et (Zéro, un ou plusieurs)

Exemple :



La Classe 1 doit ajouter une liste de références sur **des** instances de la classe 2.
La Classe 2 doit ajouter une liste de références sur **des** instances de la classe 1.

3.3.2.3 Zéro ou Un

Dans ce cas, il faut créer une référence sur une instance associée. Cette référence peut ne pas être renseignée (mise à NULL). (Cf Exactement Un)

3.3.2.4 Spécifié numériquement

Dans ce cas, comme le nombre d' éléments est connu à l' avance, il faut créer un tableau de références sur les instances associées de la taille spécifiée numériquement.

3.3.3 Ordre d' une association

La notion d' ordre est un facteur qui entre en compte à partir du moment où une association peut comprendre une cardinalité n. Cette information constitue une propriété de la liste ou du tableau : Ordre de classement des informations.
La liste ou le tableau peut être classé par ordre croissant, décroissant, pile ou file.

3.3.4 Association n-aire

Il n' y aura pas d' associations n-aire. Si une association n-aire se dessine, il faut la décomposer en plusieurs associations binaires.

3.3.5 Attributs d' une association

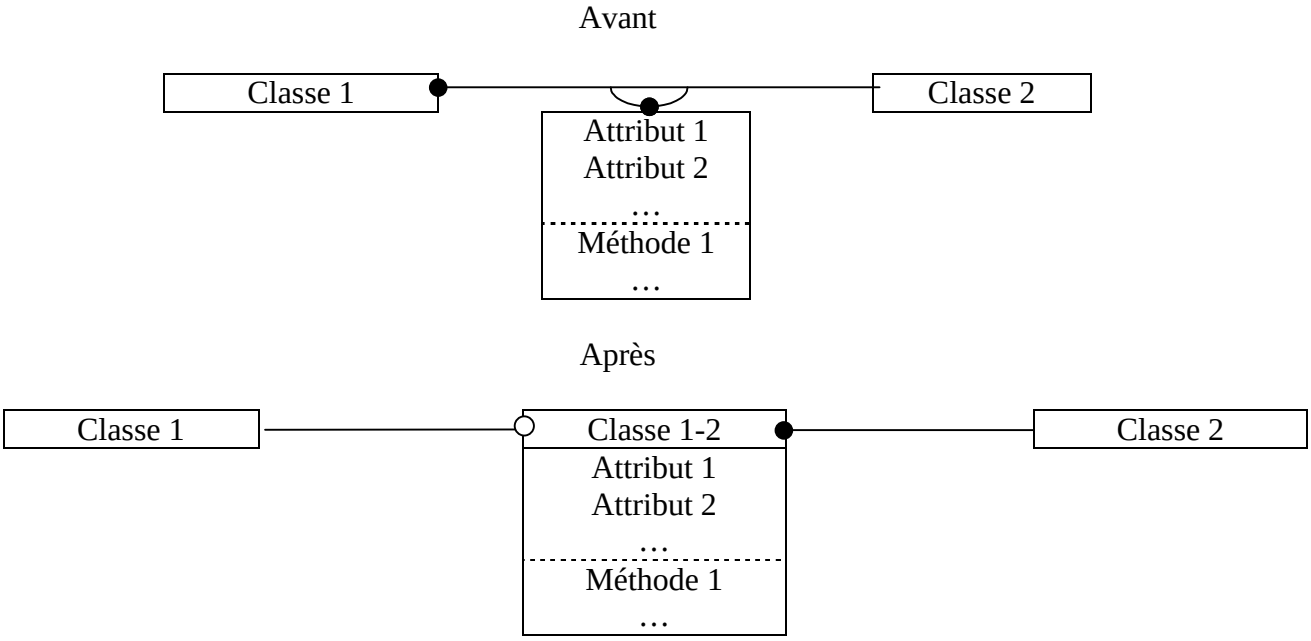
Si une association dispose d' attributs, il faut la transformer en une classe possédant comme propriétés les attributs de l' association et positionner deux associations entre cette nouvelle classe et les deux classes associées. Après quoi, il faut appliquer les règles décrites ci-dessus.

3.3.6 Méthodes obligatoires pour la gestion des associations

Il faut définir les méthodes suivantes :

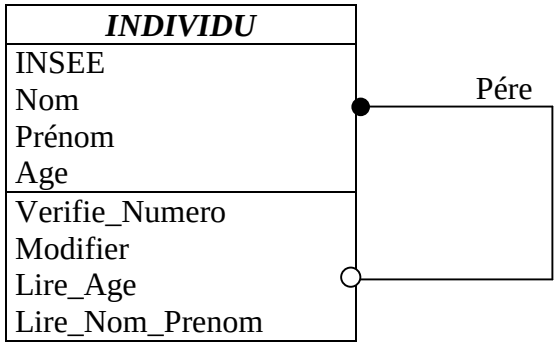
- **Ajouter_reference_<nom de la classe associée>** : ajoute une référence à une liste
- **Modifier_reference_<nom de la classe associée>** : modifie une référence unique
- **Supprimer_reference_<nom de la classe associée>** :supprime une référence d' une liste

Exemple :



3.3.7 Exemple d' association

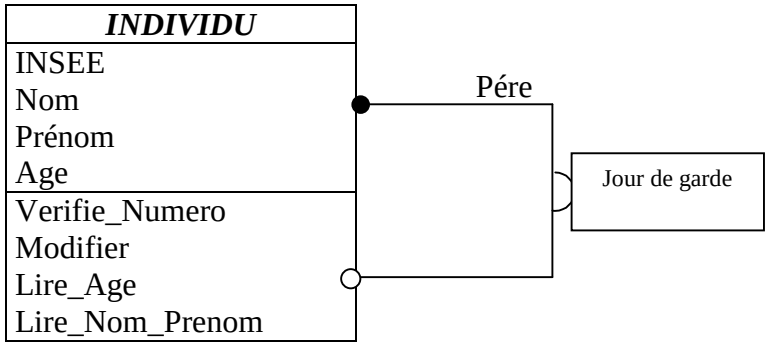
3.3.7.1 Description d' une relation père- enfant (sans attributs)




```
typedef struct t_Individu
{
  /* méthodes obligatoires */

  void (*Liberer)(struct t_Individu **);
  int (*Comparer) (struct t_Individu *,struct t_Individu *);
  void (*Tracer)(struct t_Individu *,e_Trace,char *);
  e_Booleen (*Est_coherente)(struct t_Individu *);
  void (*Interroger)(struct t_Individu *,t_Requete *,t_Reponse *);

  /* déclaration des méthodes */

  e_Booleen (*Verifie_numero) (struct t_Individu *);
  void (*Modifier)(struct t_Individu *,char *,char *,int);
  int (*Lire_age)(struct t_Individu *);
  void (*Lire_nom_prenom)(struct t_Individu *,int,char *,int,char *);
```

```
  /* déclaration des propriétés */

  char INSEE[16];
  char Nom[32];
  char Prenom[64];
  int Age;

  /* déclaration de l'association */

  t_Liste_generique *Ref_enfant;
  struct t_Individu *Père; /* NULL quand pas de père */
} t_Individu;

extern t_Individu *Instancier_individu(char *,char *,char *,int);

extern t_Individu *Instancier_copie_individu(t_Individu *);
```

3.3.7.2 Description d' une relation père- enfant (avec attributs)




```
typedef struct t_Individu
{
  /* méthodes obligatoires */
  void (*Liberer)(struct t_Individu **);
  int (*Comparer) (struct t_Individu *,struct t_Individu *);
  void (*Tracer)(struct t_Individu *,e_Trace,char *);
  e_Booleen (*Est_coherente)(struct t_Individu *);
  void (*Interroger)(struct t_Individu *,t_Requete *,t_Reponse *);

  /* déclaration des méthodes */

  e_Booleen (*Verifie_numero) (struct t_Individu *);
  void (*Modifier)(struct t_Individu *,char *,char *,int);
  int (*Lire_age)(struct t_Individu *);
  void (*Lire_nom_prenom)(struct t_Individu *,int,char *,int,char *);
  int (*Demander_pere)(struct t_Individu *,struct t_Individu **);

  /* déclaration des propriétés */

  char INSEE[16];
  char Nom[32];
  char Prenom[64];
  int Age;
```

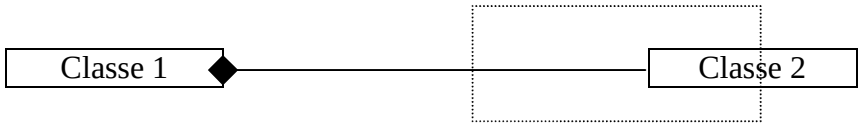
```
  /* déclaration de la relation */

  t_Liste_générique *Ref_garde;
  void *Garde_par; /* Pointeur sur une instance de t_Garde */
} t_Individu;

extern t_Individu *Instancier_individu(char *,char *,char *,int);

extern t_Individu *Instancier_copie_individu(char *,char *,char *,int);

typedef struct t_Garde
{
  /* méthodes obligatoires */
  void (*Liberer)(struct t_Garde **);

  /* déclaration des liens de la relation */
  t_Individu *Pére;
  t_Individu *Enfant;

  /* déclaration des propriétés */

  e_Booleen Garde;
} t_Garde;

/* constructeur */

extern t_Garde *Instancier_garde (t_Individu *,t_Individu *,e_Booleen
i_Garde);
```

3.4 L' agrégation

L' agrégation sert à définir l' ensemble des composants attachés à une classe.

3.4.1.1 Exactement Un

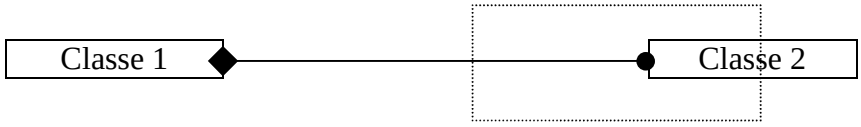
Exemple :



La Classe 1 doit ajouter une propriété qui est une référence (un pointeur) sur **une** instance de la classe 2 (Le pointeur ne peut pas être NULL).

3.4.1.2 Zéro, Un ou plus

Exemple :



La Classe 1 doit ajouter une liste de références sur **des** instances de la classe 2 (Liste éventuellement vide).

3.4.1.3 Zéro ou Un

Dans ce cas, il faut créer une référence sur une instance agrégée. Cette référence peut ne pas être renseignée (Le pointeur peut être éventuellement NULL).

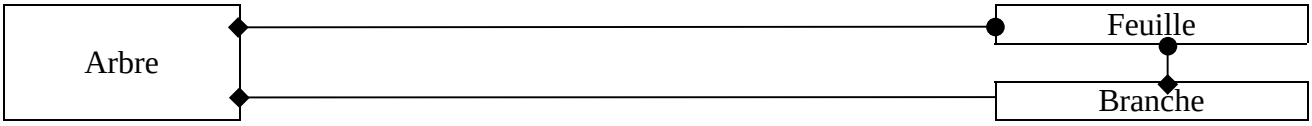
3.4.1.4 Spécifié numériquement

Dans ce cas, comme le nombre d' éléments est connu à l' avance, il faut créer un tableau de références sur les instances agrégées de la taille spécifiée numériquement.

3.4.1.5 Algorithme de l' instanciation

Lors de l' instanciation d' une classe, utilisant des agrégations, il ne faut pas oublier d' initialiser à NULL les composants si cela est nécessaire ou de les instancier.

3.4.1.6 Exemple




```
typedef struct t_Feuille
{
  …
} t_Feuille;

typedef struct t_Branche
{
   …
  t_Liste_feuille *Feuilles;  /* ensemble de feuilles situées sur
                                 la branche */
   …
} t_Branche;

typedef struct t_Arbre
{
   …
  t_Liste_feuille *Feuilles;  /* ensemble des feuilles sur l'arbre */
  t_Branche *Branche;         /* la seule branche de l'arbre */
   …
} t_Arbre;
```

3.5 L' héritage

L' héritage est un mécanisme permettant de mettre en commun un ensemble de propriétés et de méthodes liées à des classes plus spécifiques les unes que les autres (***Ex*** : un fonctionnaire hérite des informations d' un individu, un professeur hérite des informations d' un fonctionnaire, donc d' un individu).

Comme nous considérons que les propriétés d' une classe sont des propriétés privées et les méthodes sont protégées, nous devons accéder aux propriétés de la classe mère avec les accesseurs de la classe mère.

3.5.1 L' héritage simple sans opérations abstraites

L' héritage sans opérations abstraites entre la fille et la mère est géré comme une agrégation de cardinalité (1,1). La différence entre une agrégation et un héritage est la nécessité d' instancier la mère en même temps que la fille (c' est à dire que lors de l' instanciation d' une fille, il faut instancier obligatoirement une mère : surcharge du constructeur de la mère).

Les méthodes et les propriétés de la classe mère sont considérées privées

3.5.2 L' héritage multiple

L' héritage multiple sans opérations abstraites entre la fille et les mères est géré comme autant d' agrégation de cardinalité (1,1) entre la fille et les mères. Dans ce cas, il ne faut pas oublier d' instancier automatiquement toutes les mères lorsqu' une fille est instanciée.

3.5.3 Les classes abstraites

Si une classe manipule des méthodes abstraites (prototype défini au niveau de la mère dont le corps doit être écrit spécifiquement pour les classes filles), c' est une classe abstraite. La classe mère est déclarée comme une classe simple et s' utilise comme un héritage simple.

Si la classe abstraite ne contient que des méthodes abstraites, il n' est pas nécessaire d' appliquer une agrégation. Il faut toujours respecter la règle suivante :

Les méthodes de la classe fille doivent être décrites dans le même ordre que les méthodes de la classe mère en tête de la définition de la structure

3.5.4 Les classes héritières particulières

Il existe des classes héritières qui ne possèdent pas plus de méthodes et d' attributs que la classe mère. Quelques méthodes sont spécifiques, mais dans l' ensemble, l' aspect fonctionnel et technique est le même. Dans ce cas là, il n' est pas nécessaire de créer les classes filles, il suffit de positionner un indicateur qui spécifie la nature de la classe mère.

Exemple : Soit la classe mère Liste_générique, qui décline en liste générique croissante et décroissante, toutes les fonctions sont quasiment identiques à l' exception des méthodes de rangement. Dans ce cas, il n' est pas nécessaire de décrire les classes liste générique croissante ou décroissante, il suffit de positionner une nature dans la classe mère.

3.5.5 Surcharge des méthodes

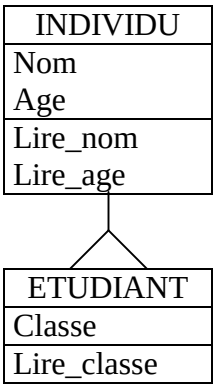
La méthode surchargée doit faire appel à la méthode de la mère à surcharger avant d' effectuer le traitement propre à la classe fille.

3.5.6 Remarque

Pour le codage des héritages en C, il ne faut pas accéder directement aux informations de la mère. Il faut utiliser les accesseurs de la mère (Définition des propriétés privées).

3.5.7 Exemple d' héritage d' une classe simple sans opération abstraite

3.5.7.1 Le modèle objet



3.5.7.2 Dans le header

```
#ifndef _individu_h
#define _individu_h

typedef struct t_Individu
{
  void (*Liberer)(struct t_Individu **); /* destructeur */
  char *(*Lire_nom)(struct t_Individu *);
  int (*Lire_age)(struct t_Individu *);

  char Nom[32];
  char Prenom[64];
} t_Individu;

extern t_Individu *Instancier_individu(char *,int);

typedef struct t_Etudiant /* derive de t_Individu */
{
  void (*Liberer)(struct t_Etudiant **); /* destructeur */
  char *(*Lire_nom)(struct t_Etudiant *); /* methode surchargée */
  int (*Lire_age)(struct t_Etudiant *); /* methode surchargée */

  char *(*Lire_classe)(struct t_Etudiant *);

  t_Individu *Individu;
  char Classe[256];
} t_Etudiant;

extern t_Etudiant *Instancier_etudiant(char *,int,char *);

#endif
```

3.5.7.3 Dans le fichier source

```
#include "individu.h"

void Liberer_individu(t_Individu **this)
{
  AL_FREE(*this);
  *this=(t_Individu *)NULL;
  return;
}

char *Lire_nom_individu(t_Individu *this)
{
  return(this->Nom);
}

int Lire_age_individu(t_Individu *this)
{
  return(this->Prenom);
}

t_Individu *Instancier_individu(char *i_Nom,int i_Age)
{
  t_Individu *Nouveau=(t_Individu *)AL_MALLOC(sizeof(t_Individu));

  /* creation d'une nouvelle instance */
  if (Nouveau==(t_Individu *)NULL)
    return((t_Individu *)NULL);

  /* positionnement des methodes */
  Nouveau->Liberer=Liberer_individu;
  Nouveau->Lire_nom=Lire_nom_individu;
  Nouveau->Lire_age=Lire_age_individu;

  /* positionnement des proprietes */
  (void) strcpy(Nouveau->Nom,i_Nom);
  Nouveau->Age=i_Age;
  return(Nouveau);
}


void Liberer_etudiant(t_Etudiant **this)
{
  if (*this!=(t_Etudiant *)NULL)
  {
      /* surcharge du destructeur de la classe mere */
     (*this)->Individu->Liberer(&(*this)->Individu);

     AL_FREE(*this);
  }

  *this=(t_Etudiant *)NULL;
  return;
}
```

```
char *Lire_nom_etudiant(t_Etudiant *this)
                                          /* surcharge de Lire_nom_individu */
{
  return(this->Individu->Lire_nom(this->Individu));
}

int Lire_age_etudiant(t_Etudiant *this)
                                          /* surcharge de Lire_age_individu */
{
  return(this->Individu->Lire_age(this->Individu));
}

char *Lire_classe_etudiant(t_Etudiant *this)
{
  return(this->Classe);
}

t_Etudiant *Instancier_etudiant(char *i_Nom,int i_Age,char *i_Classe)
{
  t_Etudiant *Nouveau=(t_Etudiant *)AL_MALLOC(sizeof(t_Etudiant));
  if (Nouveau==(t_Etudiant *)NULL)
  {
    return((t_Etudiant *)NULL);
  }

  /* instanciation de la classe mere */
  Nouveau->Individu=Instancier_individu(i_Nom,i_Age);
  if (Nouveau->Individu==(t_Individu *)NULL)
  {
    AL_FREE(Nouveau);
    return((t_Etudiant *)NULL);
  }

  /* positionnement des methodes de la classe fille */
  Nouveau->Lire_nom=Lire_nom_etudiant;
  Nouveau->Lire_age=Lire_age_etudiant;
  Nouveau->Lire_classe=Lire_classe_etudiant;
  Nouveau->Liberer=Liberer_etudiant;

  /* positionnement des proprietes de la classe fille */
  (void) strcpy(Nouveau->Classe,i_Classe);

  return(Nouveau);
}
```

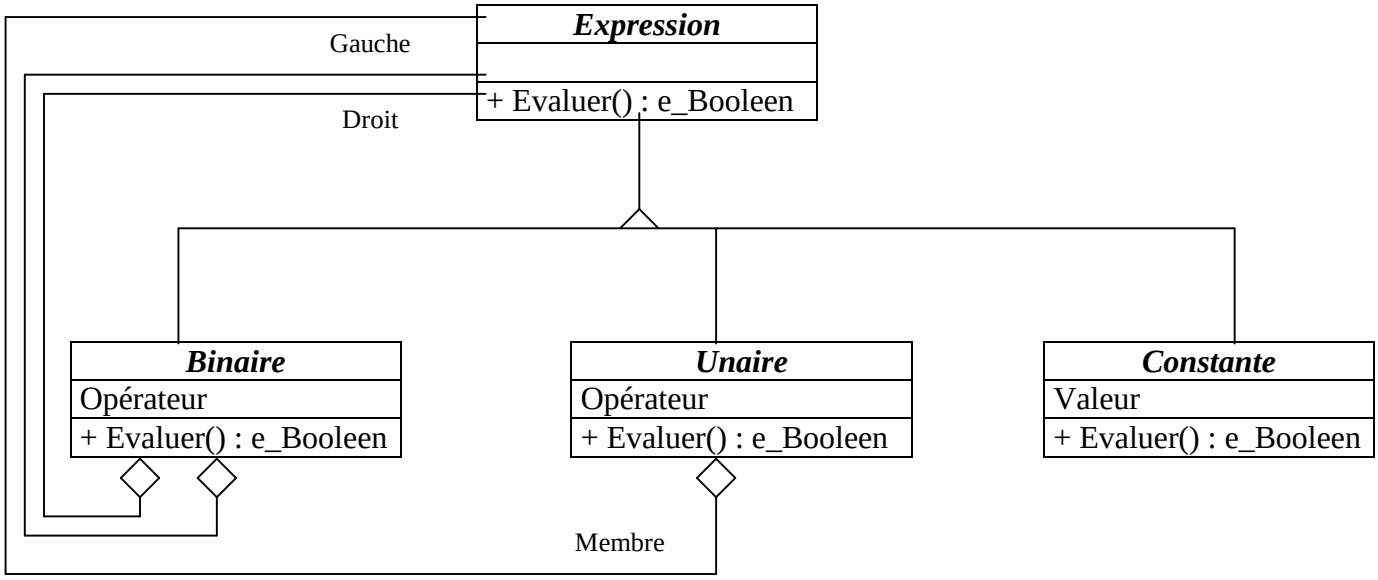
3.5.7.4 Exemple d' utilisation

```
#include "individu.h"

void main(void)
{
  t_Individu *Eleve=(t_Individu *)
                              Instancier_etudiant("DUPONT Jean",10,"DEUG");
  if (Eleve==(t_Individu *)NULL)
  {
    printf("Erreur d'allocation memoire …\n");
    return;
  }
```

```
  printf("Nom : %s\n",Eleve->Lire_nom(Eleve));
  printf("Age : %d\n",Eleve->Lire_age(Eleve));

  /* je veux acceder à la méthode Lire_classe de l'étudiant */
  /* ATTENTION : ceci peut être réalisé car je sais que
                                                    c'est un étudiant */

  printf("Classe : %s\n",((t_Etudiant *)Eleve)->Lire_classe(Eleve));

  Eleve->Liberer(&Eleve);

  return;
}
```

3.5.8 Exemple d' une expression booléenne

3.5.8.1 Modèle objet



3.5.8.2 Description du header

```
#ifndef _expression_h
#define _expression_h

typedef enum { OUI=1, NON=0, VRAI=1, FAUX=0, TRUE=1, FALSE=0 } e_Booleen;
typedef enum { ET, OU } e_Operateur_binaire;
typedef enum { NON } e_Operateur_unaire;

typedef struct t_Expression_booleenne
{
  /* methode abstraite */
  /* ----------------- */
  void (*Liberer)(struct t_Expression_booleenne **);
  e_Booleen (*Evaluer)(struct t_Expression_booleenne *);

} t_Expression_booleenne;
```

```
typedef struct t_Constante
{
  /* methode abstraite */
  /* ----------------- */
  void (*Liberer)(struct t_Constante **);
  e_Booleen (*Evaluer)(struct t_Constante *);

  /* propriete de la classe */
  /* ---------------------- */
  int Valeur;
} t_Constante;

extern t_Constante *Instancier_constante(int Valeur);

typedef struct t_Unaire
{
  /* methode abstraite */
  /* ----------------- */
  void (*Liberer)(struct t_Unaire **);
  e_Booleen (*Evaluer)(struct t_Unaire *);

  /* propriete */
  /* --------- */
  e_Operateur_unaire Operateur;
  t_Expression_booleenne *Membre;
} t_Unaire;

extern t_Unaire *Instancier_unaire(char Operateur,t_Expression_booleenne
                                                                  *Membre);

typedef struct t_Binaire
{
  /* methode abstraite */
  /* ----------------- */
  void (*Liberer)(struct t_Binaire **);
  e_Booleen (*Evaluer)(struct t_Binaire *);

  /* propriete */
  /* --------- */
  e_Operateur_binaire Operateur;
  t_Expression_booleenne *Gauche;
  t_Expression_booleenne *Droite;
} t_Binaire;

extern t_Binaire *Instancier_binaire(char Operateur,
                                     t_Expression_booleenne *Gauche,
                                     t_Expression_booleenne *Droite);
#endif
```

3.5.8.3 La description du fichier source C

```
#include <stdio.h>
#include "expression.h"

e_Booleen Evaluer_constante(t_Constante *this)
{
  return(this->Valeur);
}
```

```
void Liberer_constante(t_Constante **this)
{
  /* liberation de l'instance */
  AL_FREE(*this);

  /* positionnement de l'instance a NULL */

  *this=(t_Constante *)NULL;
  return;
}

t_Constante *Instancier_constante(int Valeur)
{
  /* allocation */
  t_Constante *Nouveau=(t_Constante *)AL_MALLOC(sizeof(t_Constante));
  if (Nouveau==(t_Constante *)NULL)
    return((t_Constante *)NULL);

  /* positionnement de methodes */
  Nouveau->Evaluer=Evaluer_Constante;
  Nouveau->Liberer=Liberer_Constante;

  /* positionnement de la valeur */

  Nouveau->Valeur=Valeur;

  return(Nouveau);
}

e_Booleen Evaluer_unaire(t_Unaire *this)
{
  if(this->Operateur==NON)
  {
    return(!this->Membre->Evaluer(this->Membre));
  }
  return(0);
}

void Liberer_unaire(t_Unaire **this)
{
  (*this)->Membre->Liberer(&(*this)->Membre);

  AL_FREE(*this);
  *this = (t_Unaire *)NULL;
  return;
}

t_Unaire *Instancier_unaire(e_Operateur_unaire Operateur,
                                              t_Expression_booleenne *Membre)
{
  /* allocation */

  t_Unaire *Nouveau=(t_Unaire *)AL_MALLOC(sizeof(t_Unaire));
  if (Nouveau==(t_Unaire *)NULL)
    return((t_Unaire *)NULL);

  /* positionnement de methodes */

  Nouveau->Evaluer=Evaluer_Unaire;
  Nouveau->Liberer=Liberer_Unaire;
```

```
  /* positionnement de la valeur */

  Nouveau->Operateur=Operateur;
  Nouveau->Membre=Membre;

  return(Nouveau);
}

e_Booleen Evaluer_binaire(t_Binaire *this)
{
  e_Booleen Valeur=FAUX;

  switch(this->Operateur)
  {
    case ET : Valeur=this->Gauche->Evaluer(this->Gauche)&&
                     this->Droite->Evaluer(this->Droite);
              break;
    case OU : Valeur=this->Gauche->Evaluer(this->Gauche)||
                     this->Droite->Evaluer(this->Droite);
              break;
  }

  return(Valeur);
}

void Liberer_binaire(t_Binaire **this)
{
  (*this)->Gauche->Liberer(&(*this)->Gauche);
  (*this)->Droite->Liberer(&(*this)->Droite);

  AL_FREE(*this);

  *this=(t_Binaire *)NULL;
  return;
}

t_Binaire *Instancier_binaire(e_Operateur_binaire Operateur,
                                                t_Expression_booleenne
*Gauche,
                                                t_Expression_booleenne
*Droite)
{
  /* allocation */
  t_Binaire *Nouveau=(t_Binaire *)AL_MALLOC(sizeof(t_Binaire));
  if (Nouveau==(t_Binaire *)NULL)
    return((t_Binaire *)NULL);

  /* positionnement de methodes */

  Nouveau->Evaluer=Evaluer_binaire;
  Nouveau->Liberer=Liberer_binaire;

  /* positionnement de la valeur */

  Nouveau->Operateur=Operateur;
  Nouveau->Gauche=Gauche;
  Nouveau->Droite=Droite;

  return(Nouveau);
}
```

3.5.8.4 Un programme d' exemple

```
#include <stdio.h>
#include "expression.h"

void main()
{
e_Booleen Valeur;
t_Expression_booleenne *Expr=(t_Expression_booleenne *)Instancier_binaire(ET,
                          (t_Expression_booleenne *)Instancier_unaire(NON,
                          (t_Expression_booleenne *)Instancier_constante(1)),
                          (t_Expression_booleenne *)Instancier_constante(0));

Valeur = Expr->Evaluer(Expr);

printf("Expr = %s\n",Valeur?"VRAI":"FAUX");

Expr->Liberer(&Expr);
return;
}
```

3.6 Les patrons ou les modèles

3.6.1 Description d' un patron

Un patron est une classe générique offrant un ensemble de méthodes communes quelque soit le type de l' objet manipulé (***Exemple*** : une liste générique). Les fonctionnalités de la gestion d' une liste de produits ou d'individus seront les mêmes (Ajout, Suppression, Parcours, …).

Un patron gérera un objet comme étant une référence sur un objet quelconque (void *). Pour manipuler cet objet, il faudra définir un certain nombre de méthodes propres à l' objet défini lors de l' instanciation d' un patron. Le patron n' est donc pas responsable de l' implémentation de ces méthodes, seule la classe de l' objet doit définir ces méthodes. En d'autres termes, ce sont un ensemble de classes partageant un même ensemble de méthodes.

Exemple : Une liste croissante générique

Cet exemple manipule n' importe quel objet mais le patron doit savoir comment sont classés les objets. Il a donc besoin d' une fonction de comparaison entre 2 objets de la liste pour les classer.

3.6.2 Exemple d' un patron : Liste_Générique_Croissante

```
typedef struct t_Liste_generique_croissante
{
  /* destructeur */
  void (*Liberer)(struct t_Liste_generique_croissante **);

  /* méthodes du patrons */

  void (*Ajouter)(struct t_Liste_generique_croissante *,void *);
  void (*Supprimer)(struct t_Liste_generique_croissante *,void *);
  void *(*Rechercher)(struct t_Liste_generique_croissante *,void *);
```

```
  /* méthode propre à l'objet à généraliser */

  int (*Comparer)(void *,void *);

  /* propriétés du patron */

  int Nb_Elements;
  void **Liste_element;

} t_Liste_générique_croissante;

extern t_Liste_generique_croissante *Instancier_Liste_generique_croissante(
                                          int (*Comparer)(void *,void *));
```

3.7 Les classes interfaces

3.7.1 Définition

Une classe interface est une classe dont seules les méthodes sont visibles de l' extérieure par les classes qui les utilisent.

3.7.2 Codage d' une classe interface

3.7.2.1 Dans le header

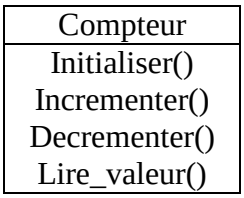
Dans le header de la classe interface, doit apparaître uniquement la description de cette dernière avec les méthodes. Les différentes instanciations possibles doivent y figurer aussi.

3.7.2.2 Dans le fichier source

Dans le fichier source de la classe interface, doit figurer la description de toutes les classes implémentant la classe interface. Ces nouvelles classes peuvent disposer de méthodes supplémentaires et/ou de propriétés.

3.7.3 Exemple d' une classe interface

3.7.3.1 Description



3.7.3.2 Dans le header

```
#ifndef _compteur_h
#define _compteur_h

typedef struct t_Compteur
{
      void (*Liberer)(struct t_Compteur **);
```

```
      void (*Initialiser)(struct t_Compteur *);
      void (*Incrementer)(struct t_Compteur *);
      void (*Decrementer)(struct t_Compteur *);
      int (*Lire_valeur)(struct t_Compteur *);
} t_Compteur;

extern t_Compteur *Instancier_compteur_sequentiel(int Mini, int Maxi);
extern t_Compteur *Instancier_compteur_circulaire(int Mini,int Maxi);

#endif
```

3.7.3.3 Dans le fichier source

```
#include <stdio.h>
#include "compteur.h"

typedef struct t_Compteur_sequentiel
{
      void (*Liberer)(struct t_Compteur_sequentiel **);
      void (*Initialiser)(struct t_Compteur_sequentiel *);
      void (*Incrementer)(struct t_Compteur_sequentiel *);
      void (*Decrementer)(struct t_Compteur_sequentiel *);
      int (*Lire_valeur)(struct t_Compteur_sequentiel *);

      int Compteur_courant;
      int Mini,Maxi;
} t_Compteur_sequentiel;

void Initialiser_sequentiel(t_Compteur_sequentiel *this)
{
      this->Compteur_courant = this->Mini;
      return;
}

void Incrementer_sequentiel(t_Compteur_sequentiel *this)
{
      if (this->Compteur_courant != this->Maxi)
            this->Compteur_courant ++;
      return;
}

void Decrementer_sequentiel(t_Compteur_sequentiel *this)
{
      if (this->Compteur_courant != this->Mini)
            this->Compteur_courant --;
      return;
}

int Lire_valeur_compteur_sequentiel(t_Compteur_sequentiel *this)
{
      return(this->Compteur_courant);
}

void Liberer_compteur_sequentiel(t_Compteur_sequentiel **this)
{
      AL_FREE(*this);
      *this = (t_Compteur_sequentiel *)NULL;
      return;
}
```

```
t_Compteur *Instancier_compteur_sequentiel(int Mini,int Maxi)
{
      t_Compteur_sequentiel *Nouveau=(t_Compteur_sequentiel *)
                                    AL_MALLOC(sizeof(t_Compteur_sequentiel));
      if (Nouveau == (t_Compteur_sequentiel *)NULL)
            return((t_Compteur *)NULL);


      Nouveau->Initialiser=Initialiser_compteur_sequentiel;
      Nouveau->Incrementer=Incrementer_compteur_sequentiel;
      Nouveau->Decrementer=Decrementer_compteur_sequentiel;
      Nouveau->Lire_valeur=Lire_valeur_compteur_sequentiel;
      Nouveau->Liberer=Liberer_compteur_sequentiel;

      if (Mini < Maxi)
      {
            Nouveau->Mini = Mini;
            Nouveau->Maxi = Maxi;
      }
      else
      {
            Nouveau->Mini = Maxi;
            Nouveau->Maxi = Mini;
      }

      return((t_Compteur *)Nouveau);
}

typedef struct t_Compteur_circulaire
{
      void (*Liberer)(struct t_Compteur_circulaire **);
      void (*Initialiser)(struct t_Compteur_circulaire *);
      void (*Incrementer)(struct t_Compteur_circulaire *);
      void (*Decrementer)(struct t_Compteur_circulaire *);
      int (*Lire_valeur)(struct t_Compteur_circulaire *);

      int Compteur_courant;
      int Mini,Maxi;
} t_Compteur_circulaire;

void Initialiser_circulaire(t_Compteur_circulaire *this)
{
      this->Compteur_courant = this->Mini;
      return;
}


void Incrementer_circulaire(t_Compteur_circulaire *this)
{
      if (this->Compteur_courant != this->Maxi)
            this->Compteur_courant ++;
      else
            this->Compteur_courant = this->Mini;

      return;
}
```

```
void Decrementer_circulaire(t_Compteur_circulaire *this)
{
      if (this->Compteur_courant != this->Mini)
            this->Compteur_courant --;
      else
            this->Compteur_courant = this->Maxi;

      return;
}

int Lire_valeur_compteur_circulaire(t_Compteur_circulaire *this)
{
      return(this->Compteur_courant);
}

void Liberer_compteur_circulaire(t_Compteur_circulaire **this)
{
      AL_FREE(*this);
      *this = (t_Compteur_circulaire *)NULL;
      return;
}

t_Compteur *Instancier_compteur_circulaire(int Mini,int Maxi)
{
      t_Compteur_circulaire *Nouveau=(t_Compteur_circulaire *)
                                    AL_MALLOC(sizeof(t_Compteur_circulaire));
      if (Nouveau == (t_Compteur_circulaire *)NULL)
            return((t_Compteur *)NULL);

      Nouveau->Initialiser=Initialiser_compteur_circulaire;
      Nouveau->Incrementer=Incrementer_compteur_circulaire;
      Nouveau->Decrementer=Decrementer_compteur_circulaire;
      Nouveau->Lire_valeur=Lire_valeur_compteur_circulaire;
      Nouveau->Liberer=Liberer_compteur_circulaire;

      if (Mini < Maxi)
      {
            Nouveau->Mini = Mini;
            Nouveau->Maxi = Maxi;
      }
      else
      {
            Nouveau->Mini = Maxi;
            Nouveau->Maxi = Mini;
      }

      return((t_Compteur *)Nouveau);
}
```

3.8 Ordre de déclaration des éléments d' un type

Dans la structure permettant de modéliser une classe, il faut définir :

1. Le destructeur
2. Les méthodes abstraites (publiques)
3. Les méthodes surchargées (protégées)
4. Les méthodes propres à la classe (privées)
5. Les propriétés
6. Les agrégations
7. Les associations

3.9 La gestion des erreurs (mécanisme d' exceptions)

Pour gérer les erreurs survenant dans une classe, un peu comme des exceptions, il faut ajouter une propriété : Erreur de type e_Erreur. Ce type énuméré recense toutes les exceptions utilisées dans cette bibliothéque. Cette propriété doit alors être mise à jour dans toutes les méthodes ou accesseurs (sauf Lire_erreur) pour savoir si le dernier appel est responsable d'une anomalie ou pas.

Pour accéder à cette propriété, il faut alors ajouter un accesseur : Lire_erreur.