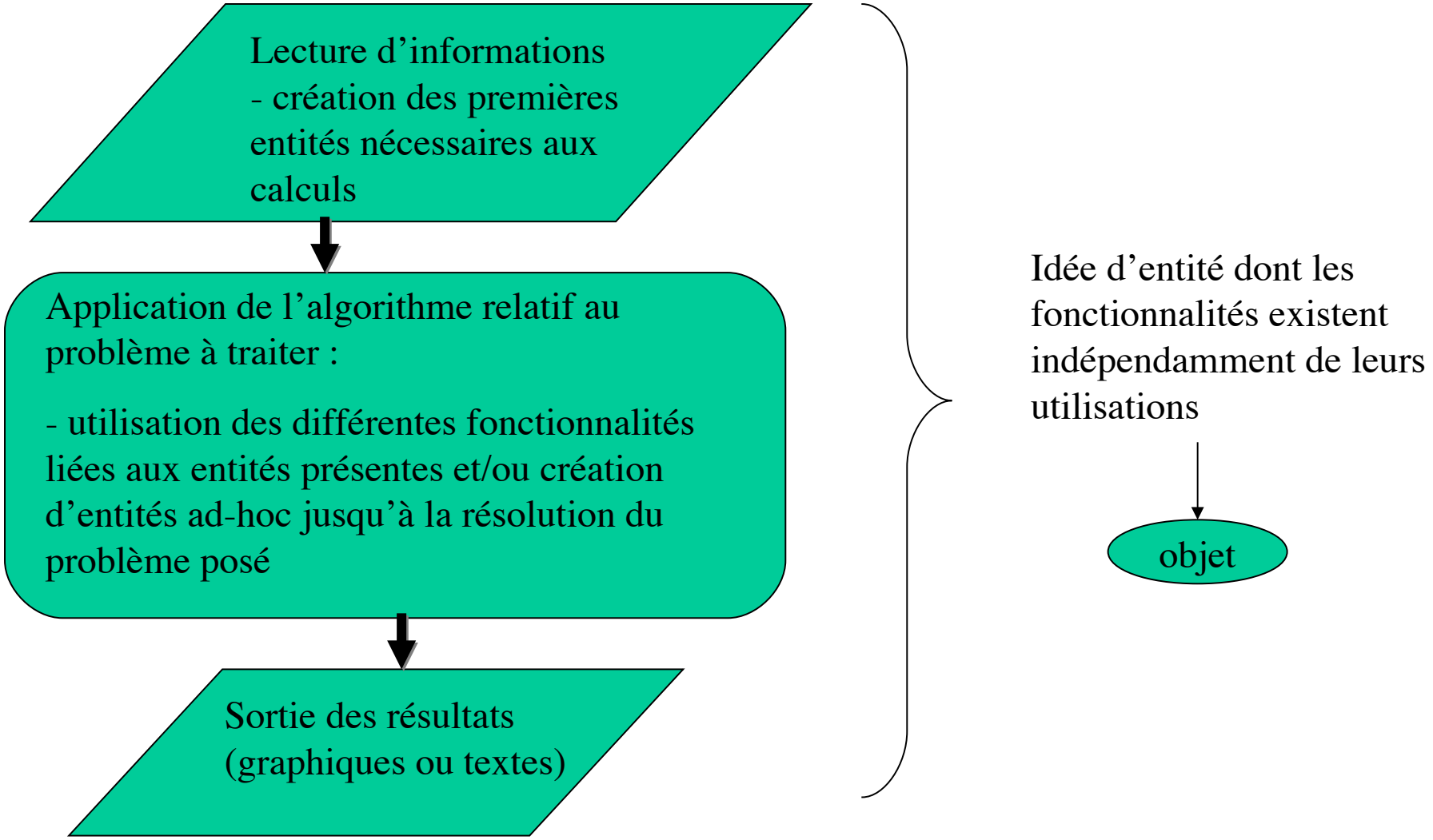
Formation pour développeurs : Logiciel Herezh++

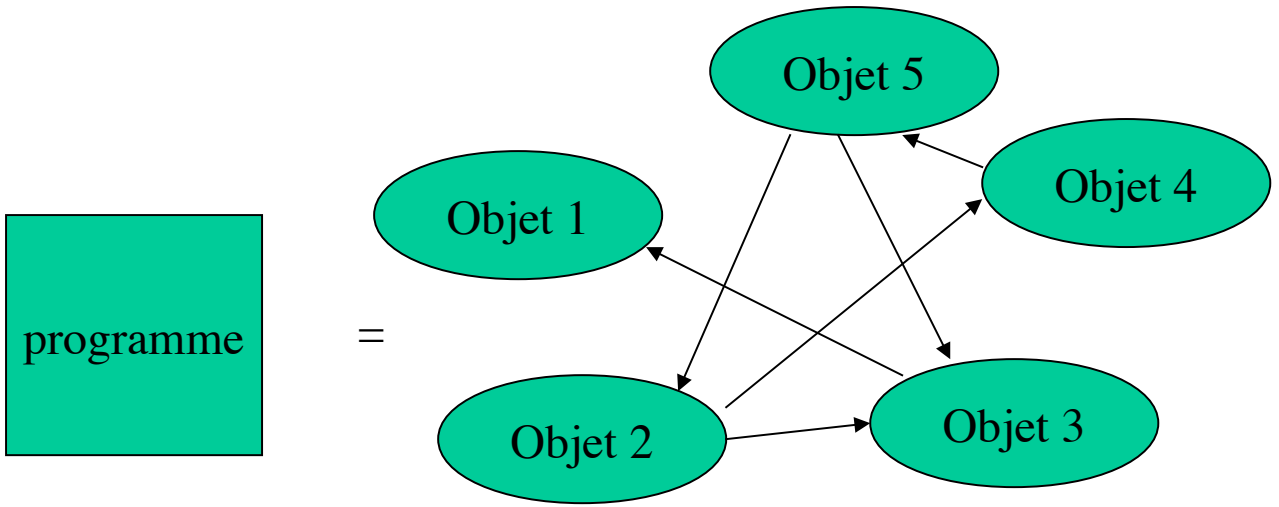
Initalisation des connaissances

Gérard Rio
Professeur émérite Université Bretagne sud
Juin 2023

Un premier schéma de fonctionnement



Langage objet, conception objet



L'objet c 'est une entité d'une classe d'objet. Exemple : un réel appartient à l 'ensemble des réels

L 'ensemble est appelé : **la classe**
Un objet est appelé : une instance de la classe

En fait, utilité de hiérarchiser les différentes classes, par exemple :

- les classes simples peuvent être totalement indépendantes les unes des autres

- les classes complexes peuvent maintenir des relations entres elles

Description rapide d'une classe type

- Comporte :
- des entités publiques ➔ accessibles par tous les autres objets
- des entités privées ➔ accessibles que par l'objet lui-même
- des entités protégées ➔ accessibles que par les classes dérivées

Par entité on entend : données et/ou méthode

Exemple d'une classe élément :

- aucune donnée publique (signifie quelles sont totalement encapsulées)
- méthodes publiques (par ex) :
  * calcul et renvoi de la raideur et du résidu local (utilise uniquement ses données propres...

- données privées (par ex) :
  * les nœuds qui composent l'élément...
- méthodes protégés (par ex) :
  * un calcul intermédiaire quelconque...

Nécessité de maîtriser la qualité d'une instance : par exemple on veut éviter une création incomplète ou incohérente d'une instance.

existence d'un ou plusieurs constructeurs.

Idem au niveau de la destruction d'un objet :

existence d'un destructeur attaché à la classe

Quelques propriétés générales intéressantes sur les classes

Surcharge d'opérateurs

Il est pratiquement possible de définir une opération quelconque entre deux objets ou sur un objet que l'on peut ensuite utiliser dans une expression complexe :
Exemple simple : opération sur des objets géométriques (vecteurs, tenseurs, plans, droites...) Une expression du type :

$$\vec{v} = \frac{((\vec{c}.\vec{d})\,\vec{e} \times (\vec{f}.\vec{g})\,\vec{h})}{\| \vec{t} \|} + \vec{j}$$

Peut s 'écrire :
vecteur v = (((c*d)*e).vectoriel((f*g)*h))/t.norme() +j;
Ou encore si le produit vectoriel est défini avec le symbole &
vecteur v = (((c*d)*e & ((f*g)*h))/t.norme() +j;

le compilateur utilise le même signe * pour réaliser des opérations différentes : produits scalaires, produit d'un scalaire avec un vecteur

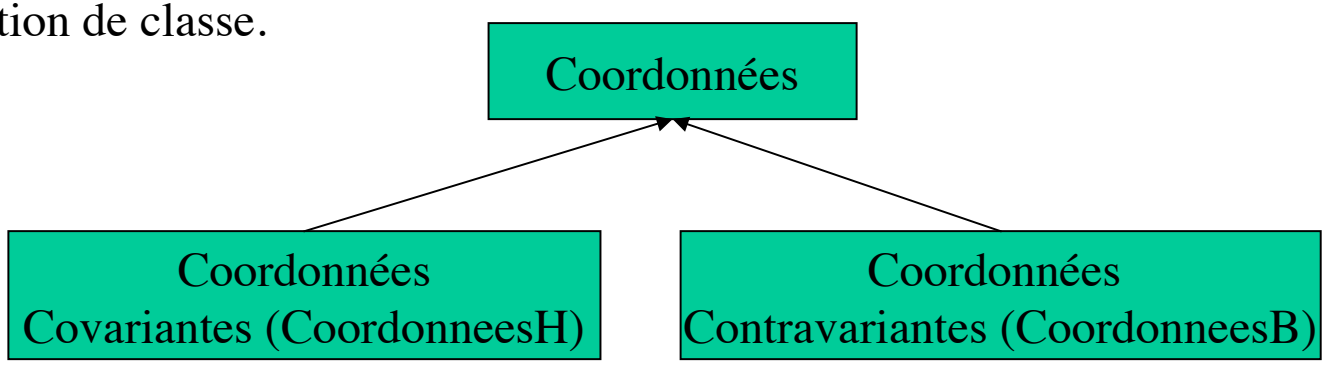
Autre exemple : surcharge de l'opérateur () :
Si A est un vecteur : A(i) peut être défini comme l'opération qui ramène la ième composante mais qui de plus vérifie que i est compris entre 1 et la dimension de l'espace, sinon on a affichage d'un message d'erreur ! (gestion d'erreur contrôlée)

Possibilité d'adresser d'une manière standard les composantes d'une matrice stockée en bande : quand on écrit M(i,j) le programme interprète par M(i,lb-(i-j)), avec lb = largeur de bande

Héritage (simple ou multiple) :

Possibilité de construire une classe complexe en héritant d'une ou plusieurs classes déjà existantes. On bénéficie au départ des méthodes et données déjà existantes.
Permet une spécialisation de classe.

Exemple:



Le « typage » fort permet au compilateur de reconnaître les opérations autorisées des autres.

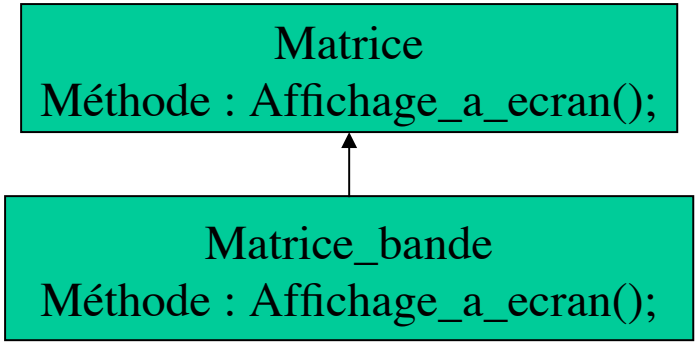
```
CoordonneesH a,b,c; CoordonneesB e,f g; double h; // déclarations des variables
c=a+b; h=a*e;   // additions et produit scalaire autorisés
c=a+e; h=a*b;  // erreur, les opérations n'ont pas été définies car d'aucun sens mathématique
```

- Dans CoordonneesH on dispose de toutes les méthodes définies dans Coordonnées
- Possibilité d'ajouter dans CoordonneesH de nouvelles méthodes (fonctions) spécifiques

Polymorphisme

Possibilité de redéfinir une méthode déjà existante : on dit surcharger (comme pour les opérateurs tels que l'additions, la multiplication …)

Exemple:



- Lorsque l'on définit une instance de la classe Matrice (c-a-d matrice carrée) la méthode associée **Affichage_a_ecran()** affiche une matrice carrée à l'écran.
- Lorsque l'on définit une instance de la classe Matrice_bande la méthode associée **Affichage_a_ecran()** affiche une matrice bande à l'écran.

Dans le cas de l'opérateur () c'est particulièrement utile :

ex : Matrice A; A(2,5) = 3; // adressage classique d'une matrice carrée
Matrice_bande B; B(2,5) = 3; // adressage de l'élément (2,lb-(2-5))

NB: remarquer le coloriage qui améliore la lisibilité : rouge pour les commentaires, vert pour les types, ocre pour les méthodes, bleu pour les mots réservés du système ou les types prédéfinis

Quelques éléments choisis de base du langage

Types prédéfinis

Il existe des types (ou classe) simples prédéfinis :

Entier ➔ int, exemple : int a,b,c; // les variables a,b,c sont du type entier

Réelle ➔ double, exemple : double e,f,h; // les variables e,f,h sont du type réel

Toutes les opérations classiques existent : +, -, *, /
Et également les opérations simplifiées : +=, -=, *=, /=
exemple : a += 3; // signifie a est remplacé par a+3

Les pointeurs:

on peut définir un pointeur de n'importe quoi

```
double * g; // g est un pointeur de double, son contenu indique l'emplacement d'un réel
double e=4;  g=&e; // g pointe sur e :  *g (c-a-d la valeur pointée) vaut 4
Double j=8; g=& j; // *g vaut maintenant 8
*g= *g +4; // exemple d'expression utilisant un pointeur
```

Les références :

Ce sont des pointeurs qui pointent toujours sur la même variable ou instance
Exemple : double & a; // une référence

Spécialisation :

```
const int a=3; // signifie que a est une constante valant 3
void Affiche_a_ecrant() const; // signifie que la méthode ne modifie pas les variables de
                  // l'instance, void signifie que cette méthode ne retourne pas de variable
int& const toto(); // ramène une référence sur une variable qui devra être utilisée comme
                     // constante
```

Entrées sorties : exemple de l'écran et du clavier

Une classe spécifique d'entrée sortie, et la définition des opérateurs >> et <<

```
cout << « ceci est un essai »; // écriture à l'écran du texte « ceci est un essai »
int a;
cout << « donnez la valeur d'un entier »;
cin >> a;  // lecture de la variable a
```

Allocation dynamique de mémoire

exemple :

```
int dimension = 200;
Vecteur * pt = new Vecteur(dimension); // allocation d'un nouveau vecteur
```

Permet pendant l'exécution d'allouer une taille variable à un objet à l'aide d'un constructeur adéquate.

Possibilité également de supprimer la place mémoire lorsqu'elle n'est plus utile.

exemple :

```
delete pt; // suppression de l'objet pointé
```

Tests et boucles classiques

```
if (condition) { action_si_condition_est_vrai;} else {sinon_autres_actions;};

for (int i=1;i<= i_maxi;i++) {ensemble_d 'actions;}
```

Encapsulage : Possibilité de bien contrôler qui peut modifier une variable et qui peut uniquement la lire

Ex : supposons une instance A de la classe Matrice qui comporte une variable interne : la dimension, on veut que l'utilisateur de A puisse connaître la dimension mais pas la modifier. Tout d'abord la variable dimension est déclarée interne à A.

Solution 1 :
On crée une méthode spécifique qui ramène une copie de la variable :

```
Int Dimension()
  { return dimension;};
```

Solution 2 :
On crée une méthode spécifique qui ramène une référence constante sur la variable interne

```
Int & const Dimension()
  { return dimension;};
```

Utilisation :

```
  cout << « la dimension est » << A.Dimension();
```

Dans les deux cas seul A peut modifier la dimension, on dit que l'on a encapsulé la variable dimension.
L'encapsulage permet d'améliorer considérablement la sûreté de fonctionnement d'un programme

Exemple de déclaration de classe

```
class Vecteur
{  public :   // méthodes publiques c -a-d accessibles de l'extérieur de la classe
     Vecteur (int n=0);  // Constructeur fonction de la taille du vecteur, initialisation à zéro par défaut
    Vecteur (const Coordonnee& a); // Constructeur fonction d'un Point

   // etc. Définition d'un ensemble de constructeur

 ~Vecteur ();              // DESTRUCTEUR :
              //--- METHODES publiques :
int Taille () const;       // Retourne la taille du vecteur
void Affiche ()  const ;    // Affichage des composantes du vecteur
void Change_taille (int n);    // Change la taille du vecteur (la nouvelle taille est n)
double Max_val_abs ()  const ; // Calcul du maximum en valeur absolue des composantes du vecteur
double Norme ()  const ; // Calcul de la norme euclidienne des composantes du vecteur
Vecteur& Normer () ; // norme le vecteur
Vecteur operator+ (const Vecteur& vec) const ; // Surcharge de l'operateur + : addition entre deux vecteurs
Vecteur operator- (const Vecteur& vec) const ; //Surcharge de l'operateur - : soustraction entre deux vecteurs
Vecteur operator* (double val)  const ;  //  multiplication d'un vecteur par un scalaire
double operator* (const Vecteur& vec) const ; //  multiplication entre deux vecteurs
double operator() (int i) const;  // Retourne la ième composante du vecteur : accès en lecture uniquement

// Etc. Toutes les méthodes et opérateurs nécessaires aux vecteurs (dans Herezh++ 29 méthodes et opérateurs)

  protected : //  données et méthodes protégées accessibles que par les méthodes de Vecteur
      int taille;      // taille du vecteur
     double* v;    // pointeur sur les composantes du vecteur
};
```

Exemple d'implantation des constructeurs, méthodes, etc.

```
// Constructeur permettant de définir un vecteur de taille n initialise a 0
Vecteur::Vecteur (int n)
{
  #ifdef MISE_AU_POINT      // uniquement utilisé en mise au point
   if ( n<0 )
            {       cout << "\nErreur : taille invalide !\n";
                    cout << "VECTEUR::VECTEUR (int ) \n";
                    Sortie(1); // appel d'une méthode générale permettant de gérer l'arrêt du programme
            };
  #endif
            if ( n==0 )
            // cas où la taille sélectionnée est nulle : initialisation identique à l'appel
            // du constructeur par défaut
            {       taille=0;
                    v=NULL;
            }
            else
            {       v=new double [n]; // allocation de la place mémoire
                    taille=n;
                    for (int i=0;i<n;i++)
                    // affectation des composantes du vecteur à 0
                            v[i]=0.0;
            };
};
```

Explications

Vecteur::Vecteur (int n) : signifie le constructeur Vecteur défini dans la classe Vecteur

#ifdef et #endif sont des ordres de précompilation

#ifdef MISE_AU_POINT : signifie si la variable MISE_AU_POINT existe alors exécuter les ordres jusqu'à rencontrer #endif

Ceci permet d'exécuter certaines parties du code sous forme conditionnelle, par exemple ici le Test ne sera exécuté qu'en phase de mise au point. Ainsi en phase normale le test sera évité pour aller plus vite (on suppose que des choix préalables évitent le cas n négatif)

```
#ifdef MISE_AU_POINT      // uniquement utilisé en mise au point
   if ( n<0 )
            { cout << "\nErreur : taille invalide !\n";
              cout << "VECTEUR::VECTEUR (int ) \n";
              Sortie(1); // appel d'une méthode générale permettant de gérer l'arrêt du programme
            };
  #endif
```

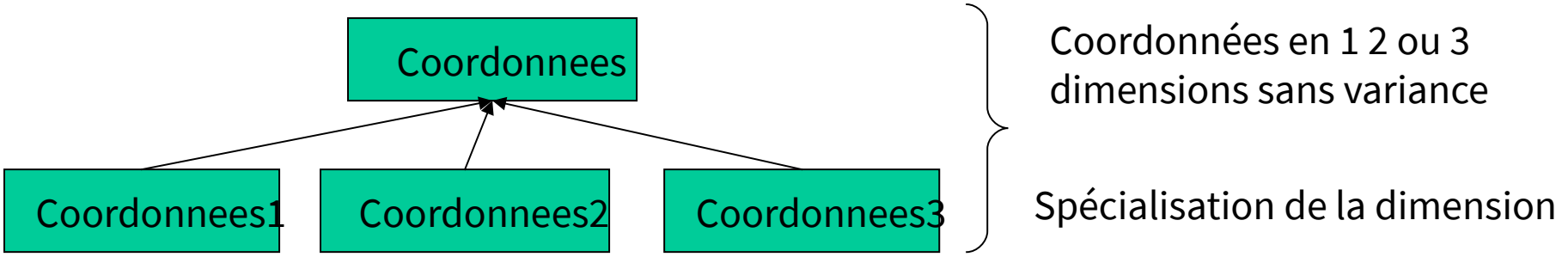
v=NULL; Signifie que le pointeur pointera sur rien

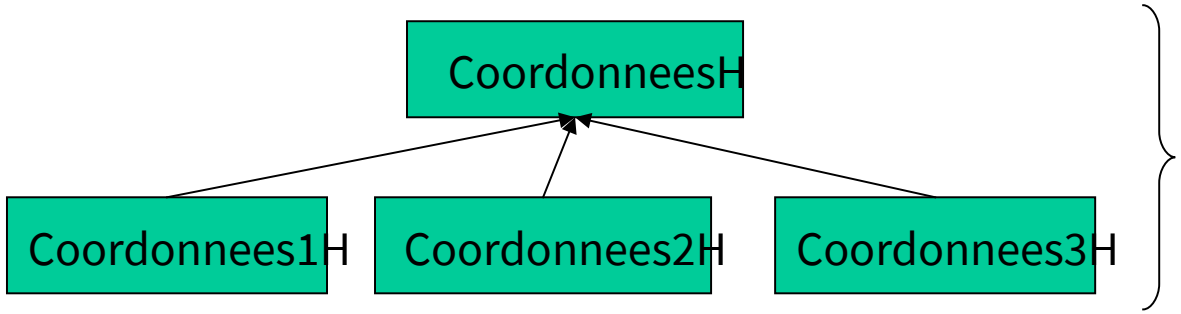
Exemple de la surcharge d'un opérateur

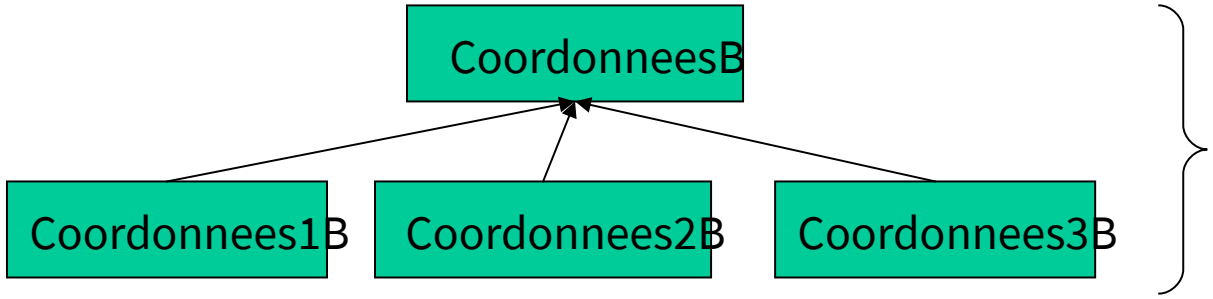
```
// Surcharge de l'opérateur * : multiplication d'un vecteur par un scalaire
Vecteur Vecteur::operator* (double val)  const
{
  #ifdef MISE_AU_POINT
   if ( taille==0 )
            {          cout << "\nErreur : taille nulle !\n";
                       cout << "VECTEUR::OPERATOR* (double ) \n";
                       Sortie(1);
            };
  #endif
            Vecteur result(taille);
            for (int i=0;i<taille;i++)
            // stockage dans le vecteur result du produit des composantes du vecteur courant
            // avec val
                       result.v[i]=val*v[i];
            return result;
};
```

Les objets sont hiérarchisées :

objets de bases : exemples des coordonnées







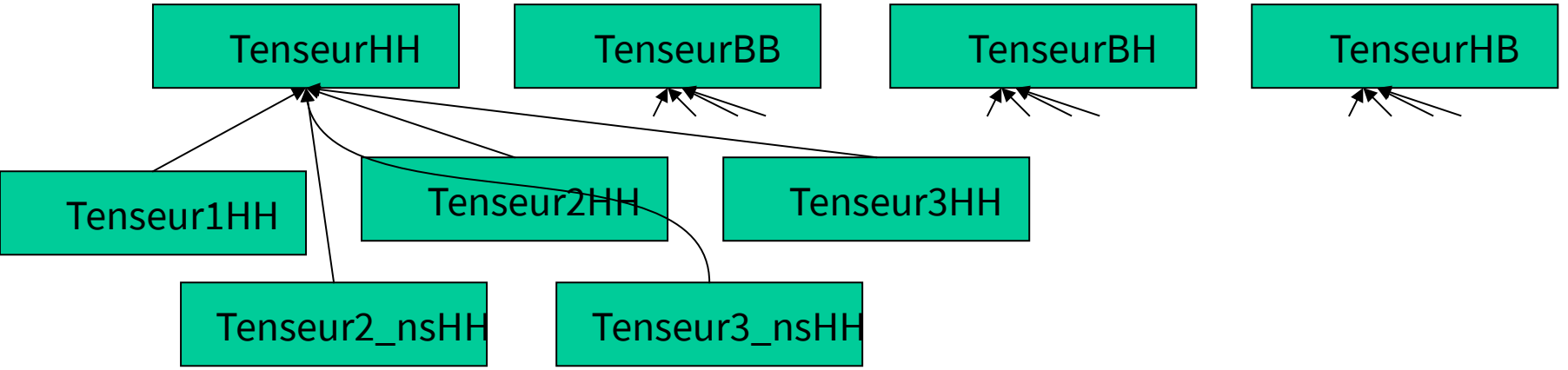
Les types de bases les plus courants sont :

- les coordonnées, ex: coordonnées des points dimension 1 2 ou 3 avec ou pas la variance

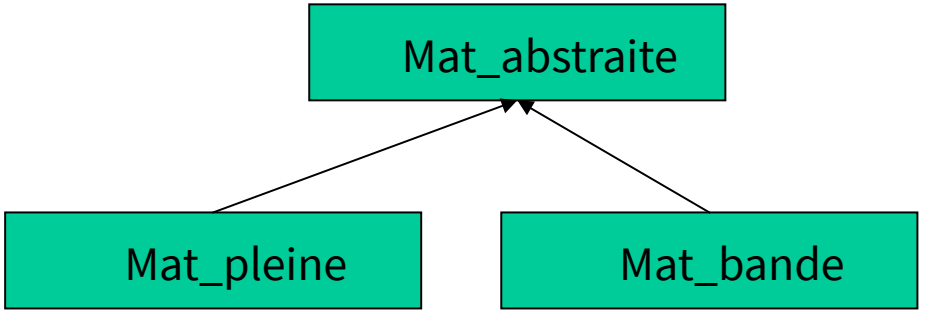
- les vecteurs, ex : vecteur second membre, dimension variable

Vecteur

- les tenseurs a priori d'ordre 2, dimension 1 2 ou 3, avec la variance 2 fois covariants, 2 fois contravariants, mixtes, symétriques ou pas



- les matrices carrées ou rectangulaires, ex : matrice de raideur locale
- les matrices stockées en bande, ex : matrice de raideur globale



Notion de Template

Un template c 'est un patron (ou un caneva) d 'un classe. Pour l 'utiliser, il suffit ensuite de définir le type (ou les types) auxquels on veut appliquer le template

Permet de définir une famille de classes différentes uniquement par le type des données auxquels elles s 'appliquent

- STL: les listes d'un type défini. Ex: liste de conditions limites

List<T>

Un template de liste, permet la définition d 'une liste de n 'importe quel type T

- les tableaux d'ordres 1 ou 2 d'un type défini, ex: un tableau de noeud

Tableau<T>

un template de tableau , permet de définir un tableau de n 'importe quel type T

Ex: le nom des degrés de liberté

```
enum Enum_ddl { X1 = 1, X2 , X3, EPAIS , TEMP , UX, UY, UZ , V1 , V2 , V3, PR,
        SIG11,SIG22,SIG33,SIG12,SIG23,SIG13,ERREUR,NU_DDL };
```

Exemple d 'utilisation :

```
Tableau <Enum_ddl> tab_ddl(4); // définition d 'un tableau de 4 noms de degré de liberté
Tab_ddl(1) = EPAIS; // une affectation
```

Notion de Template

Un template c'est un patron (ou un caneva) d'une classe. Pour l'utiliser, il suffit ensuite de définir le type (ou les types) auxquels on veut appliquer le template

Permet de définir une famille de classes différentes uniquement par le type des données auxquelles elles s'appliquent

Notion de classe virtuelle

C'est une classe qui permet de formaliser et de standardiser l'interface avec une famille d'objet.
Par exemple supposons que l'on veuille définir des méthodes qui doivent travailler avec des matrices d'une manière identique quel que soit le type de stockage (carré, bande...). Il suffit d'utiliser que les fonctions définies dans la classe virtuelle Mat_abstraite. Ces fonctions peuvent en général avoir une implémentation différente pour chaque classe dérivée, mais elles comportent la même procédure d'appel pour toutes les implémentations.

C'est au cours de l'exécution que le programme choisira la bonne fonction à appliquer
➔ **polymorphisme dynamique**

Exemple:

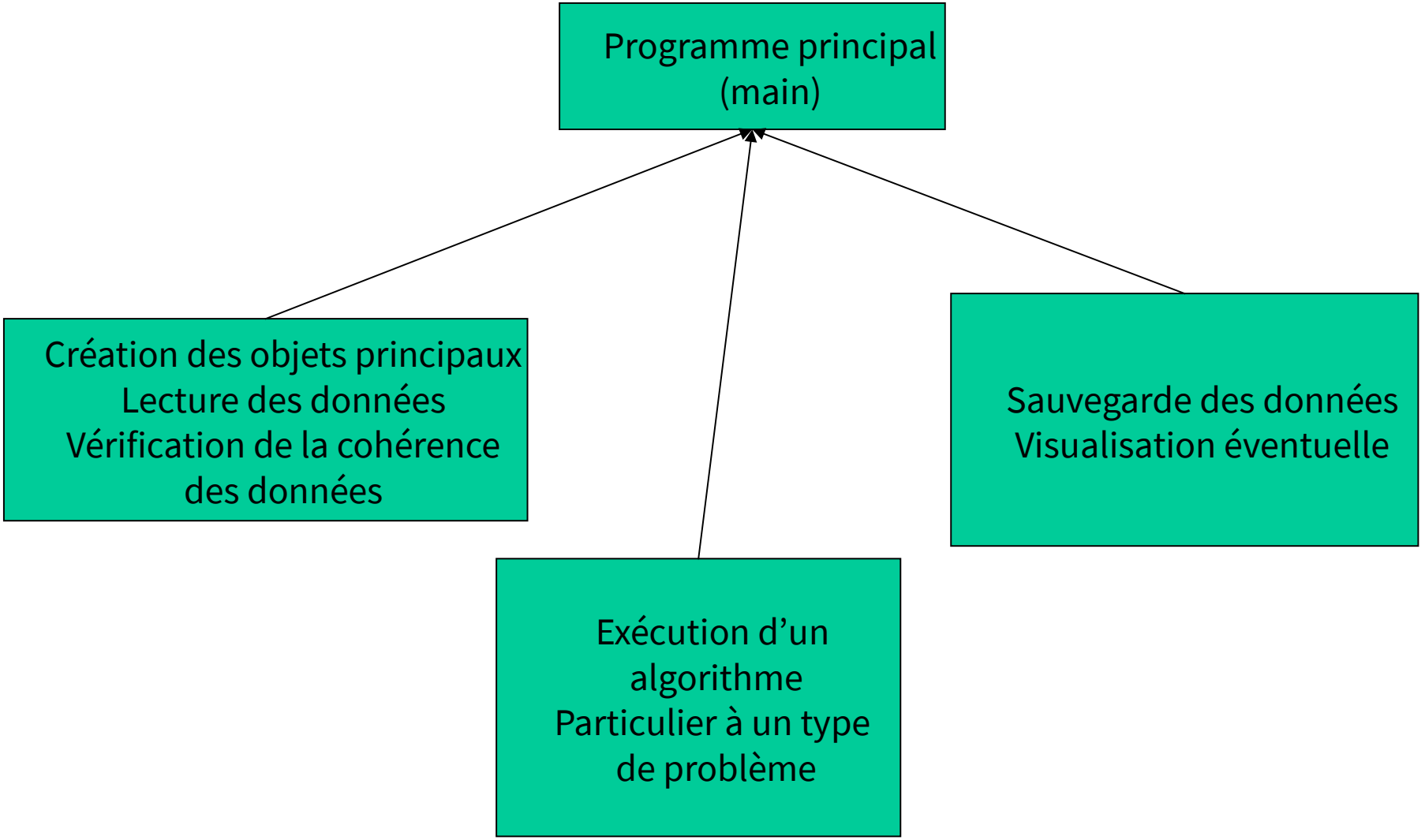
```
Class Mat_abstraite
 { …..
   // Résolution du système Ax=b  avec en sortie un new vecteur
   virtual Vecteur Resol_syst (const Vecteur& b) =0;  // fonction virtuelle pure
  ……
   }

Class Mat_bande : public Mat_abstraite // hérite de Mat_abstraite
{…..
   // Résolution du système Ax=b  avec en sortie un new vecteur
 // implémentation particulière au stockage bande
   Vecteur Resol_syst (const Vecteur& b);
  ……
   }

Class Mat_pleine : public Mat_abstraite // hérite de Mat_abstraite
{…..
   // Résolution du système Ax=b  avec en sortie un new vecteur
 // implémentation particulière au stockage carré classique
   Vecteur Resol_syst (const Vecteur& b);
  ……
   }
```

Utilisation :

```
Vecteur result = A. Resol_syst (SM); // A est une matrice quelconque, SM est un vecteur
// cette ligne fonctionne de manière identique, quel que soit le type de matrice A
```


Programme principal
(main)
Création des objets principaux
Lecture des données
Vérification de la cohérence
des données
Exécution d'un
algorithme
Particulier à un type
de problème
Sauvegarde des données
Visualisation éventuelle

Le programme principal **(version simplifiée historique : année 2000)**

```
int main ()
   {
// écriture de l'entête du programme
      cout << "\n=================================================\n";
      cout << "||              HEREZH++ version 03                ||\n";
      cout << "=================================================\n";
      cout << "\n \n";

      // définition d'un projet
      Projet projet;
// définition des quatre grandes étapes
      projet.Lecture();
      projet.Calcul();
      projet.SortieDesResultats();
      projet.Visualisation_interactive();

      cout << "\n ================================================\n";
      cout << " |              fin du calcul              |\n";
      cout << " ================================================";
      cout << endl;

   };
```

La classe projet (informations partielles)

```
Projet::Projet () // Constructeur par défaut
{   ...// définition du type d'algorithme : par exemple
    algori = new AlgoriNonDyna();
     // def des classes générales
     lesRef = new LesReferences ;  //def des références
     lesMaillages = new LesMaillages (entreePrinc,paraGlob,lesRef);// def des maillages
     lesLoisDeComp = new LesLoisDeComp ; // def des lois de comportement
     diversStockage = new DiversStockage ;  // stockage divers
     charge = new Charge ;  // chargement
     lesCondLim = new LesCondLim ;  // conditions limites
     resultats = new Resultats(entreePrinc) ; // sortie des résultats
  …….. };
void Projet::Lecture () // lecture des données
{...lesMaillages->LectureLesMaillages(); // lecture des maillages et des références
    lesLoisDeComp->LecturelesLoisDeComp(*entreePrinc,*lesRef); // lecture des lois
    diversStockage->Lecture1(*entreePrinc,*lesRef);  // première lecture des stockages divers
    charge->Lecture1(*entreePrinc,*lesRef);  // lecture des actions extérieures de chargement
    lesCondLim->Lecture1(*entreePrinc,*lesRef);  // lecture des ddl bloqués
    diversStockage->Lecture1(*entreePrinc,*lesRef);  // seconde lecture du stockage divers
    lesCondLim->Lecture2(*entreePrinc,*lesRef);  // lecture des ddl d'initialisation
    charge->Lecture2(*entreePrinc);  // lecture du type d'application du chargement
    algori->Lecture(*entreePrinc);  // lecture des paramètres de contrôle de l'algorithme
    resultats->Lecture(lesRef);  // lecture des paramètres de gestion de la sortie des résultats
……..};
```

```
    // résolution du problème
    void Projet::Calcul ()
    {  …
      // exécution de l'algorithme
       algori ➔ Execution
          (paraGlob,lesMaillages,lesRef,lesLoisDeComp,diversStockage,charge,
                  lesCondLim,lescontacts,resultats);
      ….
     }
// sortie des résultats
void Projet::SortieDesResultats ()
    { // tout d'abord sortie des résultats fichiers
      resultats ➔ SortieInfo(paraGlob,lesMaillages,lesRef,lesCondLim,lescontacts);
    };
// visualisation (éventuelle) via le format vrml
void Projet::Visualisation_interactive ()
    { // dans le cas où l'utilisateur veux également une visualisation interactive
  // via le format vrml on active l'algorithme ad-hoc
      if (paraGlob ➔ SousTypeCalcul(visualisation))
       algori ➔ Visu_vrml
          (paraGlob,lesMaillages,lesRef,lesLoisDeComp,diversStockage,charge
          ,lesCondLim,lescontacts,resultats);
      };
```

Relations entre classes :

- relation d ’héritage (déjà vu)

- relation d’inclusion : une classe a pour données une ou plusieurs instances d’une autre classe, exemple :

```
class Droite
   {  public :
      …..
      protected :       // VARIABLES PROTEGEES :
      Coordonnee A; // un point de la droite
      Vecteur U;       // vecteur directeur de la
   droite
      ...
    };
```

une classe Droite contient une instance de la classe vecteur et de coordonnée.

En général les classes incluses sont d’un niveau inférieur

- relation de référence : une classe a pour données des références ou des pointeurs sur d’autres instances qui ont une existence propre.
Par exemple une classe élément a pour données un tableau de pointeur d’instance de nœuds

```
class Element
   {  public :
      …..
      protected :       // VARIABLES PROTEGEES :
      Tableau <Nœud*>  tab_noeud; // un point
   de la ...
    };
```

Les instances pointées et la classe qui pointe peuvent être de même niveau hiérarchique

L’emploi des pointeurs permet de ne stocker qu’une fois l’information

Quelques cas particuliers

Allocation dynamique fréquente d'une multitude de petits objets
Ex: tenseur, en particulier dans les opérations intermédiaires

Ex :

$$\sigma_i^j = a_1 \, (\epsilon_k^l \, \delta_l^k) \, \delta_i^j + a_2 \, \epsilon_i^j$$

$$\boldsymbol{\sigma} = a_1 \, (\boldsymbol{\epsilon} : \boldsymbol{Id}) + a_2 \, \boldsymbol{\epsilon}$$

Tenseur3BH sigBH = (a1 * (epsBH && IdBH3)) * IdBH3 + a2 * epsBH ;

Méthodologie:

1) Allocation d'un ensemble de n (500 par exemple) x 9 réels via une liste (STL) attachée à la classe **Tenseur3BH**
2) Constructeur de **Tenseur3BH** = demande d'un nouveau nœud (9 réels) de la liste (un élément des 500: automatique)
3) Destructeur = redonne à la liste le nœud (les 9 réels)

Avantages:

- Appel de la fonction d'allocation système uniquement pour une zone mémoire importante
- Gestion des nœuds de la liste (allocation, désallocation, appel de l'allocation système) transparente (méthode interne à la classe list)

Problème de la persistance des objets intermédiaires par exemple dans le cas d'objet virtuel

Ex: cas de tenseurs dont on ne connaît pas la dimension (ce sont donc des pointeurs de tenseur)

```
sigBH = (a1 * (epsBH && IdBH) ) * IdBH3 + a2 * epsBH ;
```

T1 + T2

Pendant les opérations intermédiaires, les pointeurs continus d'exister, mais les objets pointés peuvent disparaître !

Solution:

Constat: les tenseurs, résultats d'opérations, sont construit à partir d'un appel à new()
ex: en 3D T1 est obtenue par l'opérateur :
Tenseur3HH::operator * (const double & b),
qui fera donc appel en interne à new **Tenseur3HH**;

- Création d'une liste qui sauvegarde tous les pointeurs d'objet intermédiaire
- La liste intermédiaire est supprimée au moment de l'affectation
  (**Tenseur3HH**::operator =(const **TenseurHH** & B))

Coût:
- Ajout de nœuds dans la liste de sauvegarde
- Suppression de liste

→ Opérations très rapides pour les listes (redirection de pointeurs)

Minimisation du nombre d'opération pour introduire un nouvel objet descendant d'un objet générique:
Ex: nouvel élément fini, une nouvelle loi, etc..

Deux solutions

A) Une classe de gestion de l'ensemble des descendants

Ex: **LesLoisDeComp** | Permet par exemple de récupérer ou créer une nouvelle loi (un nouvel objet loi) d'un type donné, au moment de la lecture des lois

Introduction d'une nouvelle loi
= 3 lignes supplémentaires
dans le code existant
Ex: →

```
#include "Poly_hyper3D.h »
    ..
     pt = new Poly_hyper3D (); list_de_loi.push_back(pt);
    ..
case POLY_HYPER3D : pt = new Poly_hyper3D();
```

B) La construction d'une liste statique d'objet, contenant chacun un objet d'un nouveau type

NB: les objets statiques sont construits avant le démarrage du programme

Ex: les éléments.

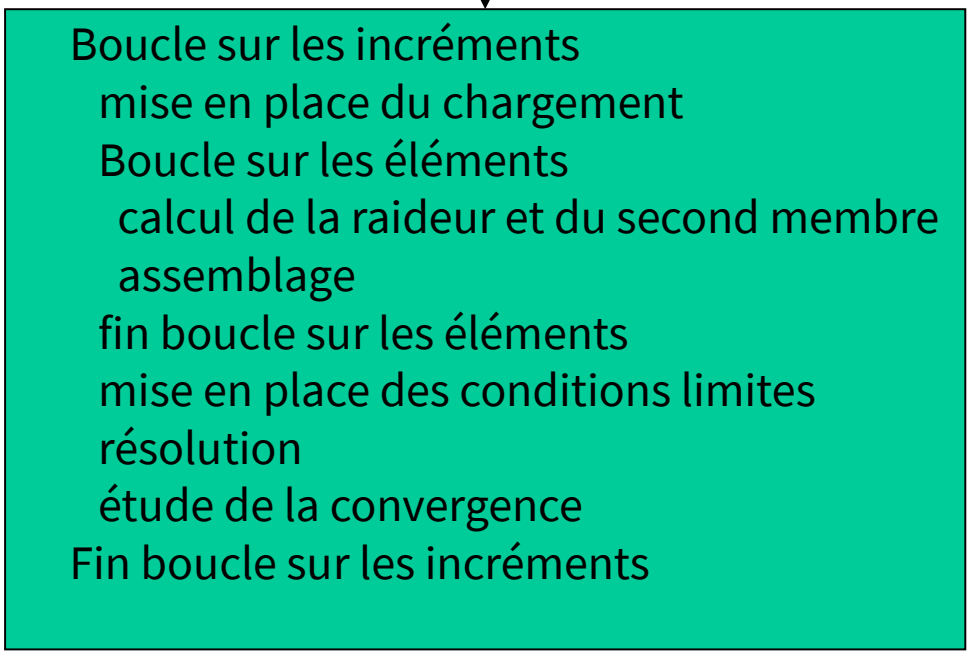
1) l'élément générique (la classe virtuelle) contient une liste statique L
2) Chaque élément contient un objet statique dont le constructeur introduit un représentant de l'objet dans L
3) Chaque élément peut créer un double de lui-même
4) l'élément générique contient des méthodes pour gérer les éléments à partir de la liste L (mais ne connaît pas explicitement les nœuds de L)

Introduction d'un nouvel élément →

Théoriquement: aucune ligne supplémentaire n'est nécessaire
Dans la pratique, pour forcer l'édition de lien => 2 lignes supplémentaires

Algorithme : problème mécanique statique sans contact

Initialisation des données existantes
Création de nouveaux containers



Exemple d'implantation de l'algorithme

Dans la classe est définie principalement une méthode qui exécute la plus grande partie de l'algorithme

```
// Calcul de l'équilibre de la pièce
void AlgoriNonDyna::Calcul_Equilibre(ParaGlob * paraGlob,LesMaillages * lesMail,
                LesReferences* lesRef,LesLoisDeComp* lesLoisDeComp,DiversStockage* diversStockage,
            Charge* charge,LesCondLim* lesCondLim,LesContacts* lesContacts
            ,Resultats* resultats)
    { // INITIALISATION globale
  // cas du chargement, on vérifie également la bonne adéquation des références
      charge ➔ Initialise(lesMail,lesRef,pa);
      // mise à zero de tous les ddl et création des tableaux a t+dt
  // les ddl de position ne sont pas mis à zéro ! ils sont initialisés à la position courante
      lesMail ➔ ZeroDdl(true);
     //  calcul de la largeur de bande effective
      int demi,total;
      lesMail ➔ Largeur_Bande(demi,total);
      int nbddl = lesMail->NbTotalDdlActifs(); // nb total de ddl
      total = min(total,nbddl);
      //  def de la matrice bande et d'un second membre global a priori matrice symétrique, initialisation à
   zéro
      MatBand matglob(demi,nbddl,0);
      Vecteur vglobin(nbddl); // def vecteur puissance interne
      Vecteur vglobex(nbddl); // def vecteur puissance externe
      Vecteur & vglobal = vglobin; // def vecteur puissance totale qui écrase vglobin
```

```
Assemblage Ass; // définition d'une instance de la classe assemblage
…...
//  boucle sur les incréments de charge
int icharge = 1; // initialisation du compteur de charge
// tant que la fin du chargement n'est pas atteinte
while ((charge ➔ Fin())||(icharge == 1))
{ double maxPuissExt;   // maxi de la puissance des efforts externes
  double maxPuissInt;   // maxi de la puissance des efforts internes
  double maxReaction;   // maxi des réactions
  int inReaction = 0;   // pointeur d'assemblage pour le maxi de réaction
  int inSol =0 ;       // pointeur d'assemblage du maxi de variation de ddl
  double maxDeltaDdl=0; //  maxi de variation de ddl
  charge ➔Avance(); // incrément de charge
  // affichage de l'incrément de charge
  cout << "\n============================================================"
     << "\n    INCREMENT DE CHARGE : " << icharge << "  intensite " << charge ➔ IntensiteCharge()
     << "\n============================================================";
  // initialisation des coordonnées et des ddl à tdt en fonction des
// ddl imposés et de l'incrément du chargement
  lesCondLim ➔ MiseAJour_tdt(lesMail,lesRef,charge  ➔ MultCharge());
  // mise en place du chargement imposé sur le second membre
// et éventuellement sur la raideur en fonction de sur_raideur
  vglobex.Zero();
  bool sur_raideur = false; // pour l'instant pas de prise en compte sur la raideur
  charge ➔ ChargeSMembreRaideur_Im
          (Ass,lesMail,lesRef,vglobex,sur_raideur,matglob,assembMat);
  maxPuissExt = vglobex.Max_val_abs();
```

```
//  boucle de convergence sur un incrément
  Vecteur * sol; // pointeur du vecteur solution
  int compteur; // déclaré à l'extérieur de la boucle car utilisé après la boucle
  for (compteur = 0; compteur<= pa.iterations;compteur++)
   {// initialisation de la matrice et du second membre
    matglob.Initialise (0.); vglobin.Zero();
    // boucle sur les éléments
    for (int nbMail =1; nbMail<= lesMail->NbMaillage(); nbMail++)
     for (int ne=1; ne<= lesMail ➔ Nombre_element(nbMail);ne++)
      { //calcul de la raideur local et du residu local
        Element & el = lesMail ➔ Element_LesMaille(nbMail,ne); // l 'élément
        Tableau<Noeud *>& taN = el.Tab_noeud(); // tableau de nœuds de l'el
        Element::ResRaid resu = el.Calcul_implicit();
        // assemblage
        Ass.AssemSM (vglobin,*(resu.res),el.TableauDdl(),taN); // du second membre
        Ass.AssembMatSym (matglob,*(resu.raid),el.TableauDdl(),taN); // de la raideur
       }
    // affichage avant mise en place des conditions limites
// calcul des maxi des puissances internes
    maxPuissInt = vglobin.Max_val_abs();
    // second membre total ( vglobal est identique a vglobin)
    vglobal += vglobex ;
    // initialisation des sauvegardes sur matrice et second membre
    lesCondLim ➔ InitSauve();
    // calcul et sauvegarde des réactions aux ddl bloque
    lesCondLim ➔ ReacAvecCoLineaire(vglobin,lesMail,lesRef);
```

```
// mise en place des conditions limites
    lesCondLim ➔ ImposeConLimtdt(lesMail,lesRef,matglob,vglobal);
    // calcul du maxi des réactions
    maxReaction = lesCondLim ➔ MaxEffort(inReaction);
    // résolution
// sortie d'info sur l'incrément concernant les réactions
    if (compteur != 0)
     InfoIncrementReac(lesMail,compteur,inReaction,maxReaction);
    // test de convergence sur un incrément
    if (Convergence(vglobal,maxPuissExt,maxPuissInt,maxReaction))
     { // sortie des itérations
       break;
      }
    // ici sol en fait = vecglob qui est écrase par la résolution
    sol = &(matglob.Resol_systID(vglobal));
    // calcul du maxi de variation de ddl
    maxDeltaDdl = sol ➔ Max_val_abs(inSol);
    // sortie d'info sur l'incrément concernant les variations de ddl
    InfoIncrementDdl(lesMail,inSol,maxDeltaDdl);
    // actualisation des ddl actifs a t+dt
    lesMail ➔ PlusDelta_tdt(*sol);
   }
```

```
   // sortie d'un message d'erreur si l'on a atteint le maximum d'itération
   if (compteur > pa.iterations)
      {  cout << "\n=========================================================="
             << "\n   ******* NON convergence des itérations d 'équilibre ********* "
             << "\n==========================================================";
       };
   // actualisation des ddl actifs de t+dt vers t
   lesMail->TdtversT();
   icharge++;
  }
 // fin des calculs
};
```