GL / C++
Chapitre 11

Standard Template Library (STL)
Présentation rapide

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

La librairie C++ standard

- C'est une collection de plusieurs éléments (fonctions, constantes, classes, objets et patrons (*templates*)) qui étendent le langage C++
- Cette librairie fournit des fonctionnalités de base :
  - Chaînes de caractères : **string Library**
  - Entrées/Sorties via des flux : **Input** / **Output Streams Library**
  - Compatibilité avec le C : **C Library**
  - Conteneurs de données : **Standard Template Library (STL)**
- Les déclarations des différents éléments de la librairie C++ standard sont réparties dans plusieurs fichiers qui doivent être inclus dans votre code pour avoir accès aux fonctionnalités dont vous avez besoin :

```
#include <string>
#include <vector>
…
```

- Les éléments proposés par la librairie standard C++ sont définis dans l'espace de nom standard (**`std`**). Pour les utiliser il faut donc :
  - soit faire précéder chaque identificateur par **`std::`**
    `std::string, std::vector, std::list, std::sort, …`
  - soit ajouter une instruction **`using namespace std`**

Standard Template Library (STL)

- Cette librairie définit plusieurs sortes de **conteneurs** de données
- Un **conteneur** est un objet permettant de stocker/manipuler une collection de données
- Le type des données stockées dans un conteneur peut être quelconque : les conteneurs sont donc proposés par la STL sous forme de **templates** (patrons)
- La STL offre un mécanisme unifié pour accéder aux éléments d'un conteneur, quel que soit le type de ce conteneur : c'est la notion **d'itérateur** (très proche du pointeur)
- Grâce à cette notion d'itérateur, la STL propose également, sous forme de fonctions, une collection **d'algorithmes** standards qui peuvent s'appliquer au contenu (ou à une partie du contenu) de n'importe quel conteneur.

Les patrons "conteneurs" de la STL (1)

- **1. Les conteneurs "séquentiels" :**
  Pour stocker une séquence d'éléments
  - **vector** : le vecteur traditionnel (éléments indicés)
  - **dequeue** (**d**ouble-**e**nded **queue**) : proche du vecteur (éléments indicés), permet facilement l'insertion/suppression en début ou en fin et le parcours dans n'importe quel ordre
  - **list** : la liste (doublement) chaînée traditionnelle

Les patrons "conteneurs" de la STL (2)

- **2. Les conteneurs "adaptateurs" :**
  Adaptateurs car ces conteneurs encapsulent un autre conteneur dont ils sont, en quelque sorte, une adaptation
  - **stack** : pile LIFO (*Last In First Out*)
  - **queue** : queue FIFO (*First In First Out*)
  - **priority_queue** : le premier élément est toujours le plus grand de la collection, selon un critère à définir

Les patrons "conteneurs" de la STL (3)

- **3. Les conteneurs associatifs :**
  Pour stocker des éléments de la forme (clé, valeur associée), ou simplement (clé), et accéder facilement à un élément grâce à la valeur de sa clé.
  - **set** : pour stocker une collection d'éléments uniques (deux éléments ne peuvent pas avoir la même valeur)
  - **multiset** : identique à set mais permet aussi de stocker des éléments ayant la même valeur
  - **map** : mémoire associative : collection d'éléments qui sont des couples (clé, valeur associée). Chaque élément à une clé unique. On accède peut accéder à un élément en donnant sa clé.
  - **multimap** : identique à **map** mais permet que plusieurs éléments aient la même clé
  - **bitset** : conteneur spécial permettant de stocker une collection de bits (optimise l'espace de stockage)

Les algorithmes de la STL

- C'est un ensemble de fonctions qui implémentent des algorithmes "standards" :
  - recherche, copie, tri, suppression, fusion, permutations, partitionnement,… etc.
- Ces fonctions opèrent en général sur une (ou des) partie(s) quelconque(s) de conteneur(s) définie(s) grâce à deux itérateurs :
  - un itérateur (first) pointant sur le premier élément de la partie à traiter
  - un autre itérateur (last) pointant sur le dernier élément de la partie à traiter

Les itérateurs

- Iterateur : objet associé à un conteneur, qui permet de "balayer" l'ensemble des objets du conteneur, sans connaître la structure de données sous-jacente
- Unifie la syntaxe et les algorithmes (tous les conteneurs peuvent se parcourir de la même façon)
- Un itérateur s'utilise comme un pointeur :
  - L'itérateur "pointe" sur un élément du conteneur
  - L'opérateur de déréférencement **`*`** s'applique à un itérateur pour obtenir l'élément pointé par cet itérateur
  - L'opérateur **`++`** permet de faire pointer un itérateur sur l'élément suivant dans le conteneur
- On peut distinguer différents types d'itérateurs :
  - RandomAccessIterator, BidirectionalIterator, ForwardIterator, InputIterator, OutputIterator
  - Selon le type de l'itérateur, d'autres opérateurs que **`*`** et **`++`** peuvent être appliqués (**`--`**, **`+`**, **`-`**, **`==`**, **`!=`**, **`<`**, **`<=`**, **`>`**, **`>=`**, **`operator []`**, **`+=`**, **`-=`**)

Exemple : le template Vector (1)

- Implémente un tableau traditionnel (éléments indicés, stockés consécutivement en mémoire)
  La taille du vecteur peut varier dynamiquement
- Méthodes de base :
  - **`v.size()`** : Retourne le nombre d'éléments contenus dans **`v`**
  - **`v[i]`** : Retourne l'élément d'indice **`i`**
  - **`v.push_back(e)`** : Ajoute un élément de valeur **`e`** à la fin du vecteur
  - **`v.pop_back()`** : Supprime le dernier élément
  - **`v.begin()`** : Retourne un itérateur sur le premier élément
  - **`v.end()`** : Retourne un itérateur sur le dernier élément
  - **`v.insert(it,e)`** : Insère un élément de valeur **`e`** DEVANT l'élément pointé par l'itérateur **`it`**.
    Après l'insertion, **`it`** pointe sur l'élément inséré
  - **`… etc`**

Exemple : le template vector (2)

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
  // Déclaration d'un vecteur v de réels
  vector<float> v; // v est vide, v.size()=0
  // Ajout d'éléments à la fin du vecteur
  v.push_back(10.5); // V[0]=10.5, v.size()=1
  v.push_back(20.3); // V[0]=10.5, V[1]=20.3, v.size()=2
  // Parcours de V en utilisant des indices
  for (unsigned int i=0; i<v.size(); i++)
    cout << v[i] << endl;
  // Parcours de V en utilisant des itérateurs
  vector<float>::iterator it; // déclaration d'un itérateur sur un vecteur
  for (it=v.begin(); it!=v.end(); it++)
    cout << *it << endl;
  // Insertion d'un élément, en position 1 par exemple
  it = v.insert(v.begin()+1, 8.2); // V[0]=10.5, V[1]=8.2, V[2]=20.3, v.size()=3
                                   // it pointe sur v[1]
  // Suppression du dernier élément
  v.pop_back(); // V[0]=10.5, V[1]=8.2, v.size()=2
  return 0;
}
```

Exemple : le template list (1)

- Implémente un liste doublement chaînée (chaque élément peut être stocké n'importe où en mémoire, il possède un pointeur vers l'élément précédent dans la liste et un autre pointeur vers l'élément suivant).

- Méthodes de base :
  - `l.push_front(e)` : Ajoute un élément e en début de liste `l`
  - `l.push_back(e)` : Ajoute un élément e en fin de liste `l`
  - `l.pop_front()` : Supprime le premier élément de la liste `l`
  - `l.pop_back()` : Supprime le dernier élément de la liste `l`
  - `l.sort()` : Trie la liste `l`
  - `l1.merge(l2)` : fusion des listes `l1` et `l2` qui doivent être triées.
    Tous les éléments de `l2` sont insérés dans `l1`
    A la fin, `l1` est triée et `l2` est vide
  - `l.unique()` : Supprime, dans chaque groupe d'éléments égaux consécutifs, tous les éléments, sauf le premier.
    Donc, si `l` est triée, supprime les doublons.
  - `l1.swap(l2)` : Permute le contenu des listes `l1` et `l2`
  - `l.reverse()` : Inverse l'ordre des éléments de la liste `l`
  - `l.remove(e)` : Supprime de la liste `l` tous les éléments égaux à `e`
  - *…etc*

Exemple : le template list (2)

```
#include <iostream>
#include <string>
#include <list>
using namespace std;
int main() {
  // Déclaration d'une liste de chaînes
  list<string> l;          // l est vide (l.size()=0)
  // Insertion d'éléments en queue de liste
  l.push_back("zebre");  // l = "zebre"
  l.push_back("auroch"); // l = "zebre", "auroch"
  // Insertion d'éléments en tête de liste
  l.push_front("gnu");   // l = "gnu", "zebre", "auroch"
  // tri de la liste
  l.sort();              // l = "auroch", "gnu", "zebre"
  // Parcours de la liste en utilisant un itérateur sur la liste
  for (list<string>::iterator it=l.begin(); it!=l.end(); it++)
    cout << *it << endl;
  return 0;
}
```

Avantages de la STL

- Ce sont les avantages de la **RE-U-TI-LI-SA-TION** :
  - On ne réinvente pas inutilement la roue
  - On gagne du temps de développement
  - On évite les bugs (la STL est bien programmée !)
  - On produit un code plus lisible (tous les développeurs C++ connaissent la STL)
  - On utilise des bibliothèques bien documentées
    `http://www.cplusplus.com/reference/stl/`
  - On travaille à un niveau d'abstraction supérieur (le concept de vecteur ou de liste chaînée devient une *brique de base* du travail de développement)
  - *…etc*
- **Conclusion** :
  - En C++, dès que possible, il faut utiliser les outils proposés par la librairie standard en général, et par la STL en particulier !
  - Développer dans un langage de haut niveau (C++, Java, C#, PHP5, …) qui propose de telles bibliothèques et ne pas les utiliser relève de la **FAUTE PROFESSIONNELLE** !

GL / C++
Chapitre 12

Design Patterns
Pattern "composite"

IUT 2 Grenoble
Université Pierre-Mendès-France
Informatique

Design Pattern

- un ***Design Pattern*** décrit un problème générique répétitif et sa solution : ce sont des ***modèles de conception réutilisables***
- On parle aussi de : ***forme de conception, pattern, modèle, patron de conception***
- Il s'agit de solutions génériques pouvant être adaptées de nombreuses fois sans que deux adaptations soient jamais identiques.
- Utiliser cette approche apprend à bien concevoir une application et à standardiser les moyens d'effectuer la conception
- On résout plus rapidement et plus sainement son application en utilisant des solutions qui ont déjà fait leurs preuves.

Un *design pattern* permet de :

- Trouver la bonne représentation « objet » et en particulier de bien choisir la granularité des objets
- Spécifier l'interface de ces objets
- Spécifier l'implémentation ces objets
- Mieux réutiliser

2 références importantes :

- *Design Patterns : Elements of reusable Object-Oriented Software.*
  Erich Gamma, Richard Helm, Ralph Jonson, John Vlissides (Gang OF 4)
  Addison-Wesley Professional computing series

- *Design Patterns for Object-Oriented Software Development.*
  Wolgang Pree
  Addison -Wesley

Eléments Principaux d'un « Pattern » (1)

1. Le **nom du pattern** permet de désigner en un ou deux mots un **problème** de conception, ses **solutions** et ses **conséquences**. Ce nom permet d'élargir le vocabulaire de conception et de réfléchir à un plus haut niveau.
2. Le **problème** décrit dans quelles circonstances le pattern peut être appliqué. C'est une description de la problématique et de son contexte. Ce peut être par exemple :
   - Comment représenter des algorithmes *via* des objets (ex : parcours d'arbre)
   - Une description de classes ou de structures d'objets « symptomatiques »
   - Ce peut être une liste de pré-conditions qui doivent être remplies pour pouvoir appliquer le pattern
   - etc…

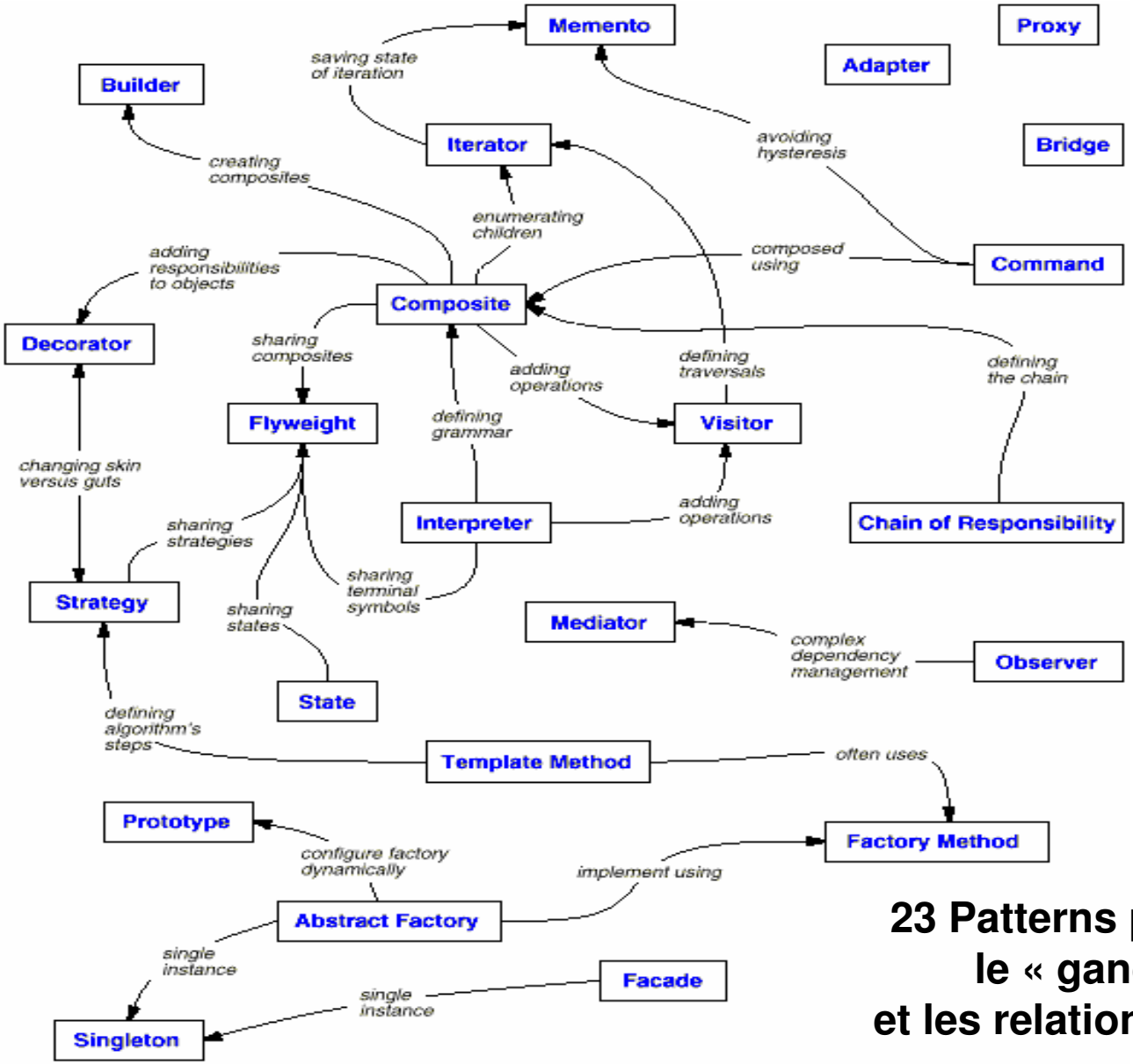
Eléments Principaux d'un « Pattern » (2)

3. La **solution** est une description des éléments de la conception, de leurs relations, de leurs rôles et de la façon dont ils collaborent.
   - La solution ne décrit pas une implémentation particulière.
   - C'est plutôt une description abstraite d'un problème de conception et de la façon dont une organisation générale de classes et d'objets permet de le résoudre
4. Les **consequences** sont les résultats et les « compromis » qui découlent de l'application du pattern. Ces conséquences peuvent concerner :
   - Le temps d'exécution, l'espace mémoire requis
   - Le type de langage à utiliser
   - L'impact sur la flexibilité, l'extensibilité et la portabilité du système que l'on développe

   Enumérer ces éléments permet de mieux les comprendre et les évaluer

Chaque pattern particulier est décrit par :

1. Son But
2. Ce qui le motive
3. Son domaine d'application
4. Sa structure
5. Le rôle de chacun des « participants »
6. Les collaborations entre les participants
7. Les conséquences
8. Des indications d'implémentation
9. Un exemple d'utilisation (avec son code)
10. Des utilisations connues du Pattern
11. Les liens avec d'autres Patterns



23 Patterns proposés par le « gang des 4 » et les relations qui les lient

Le pattern "*composite*"

Objectif :

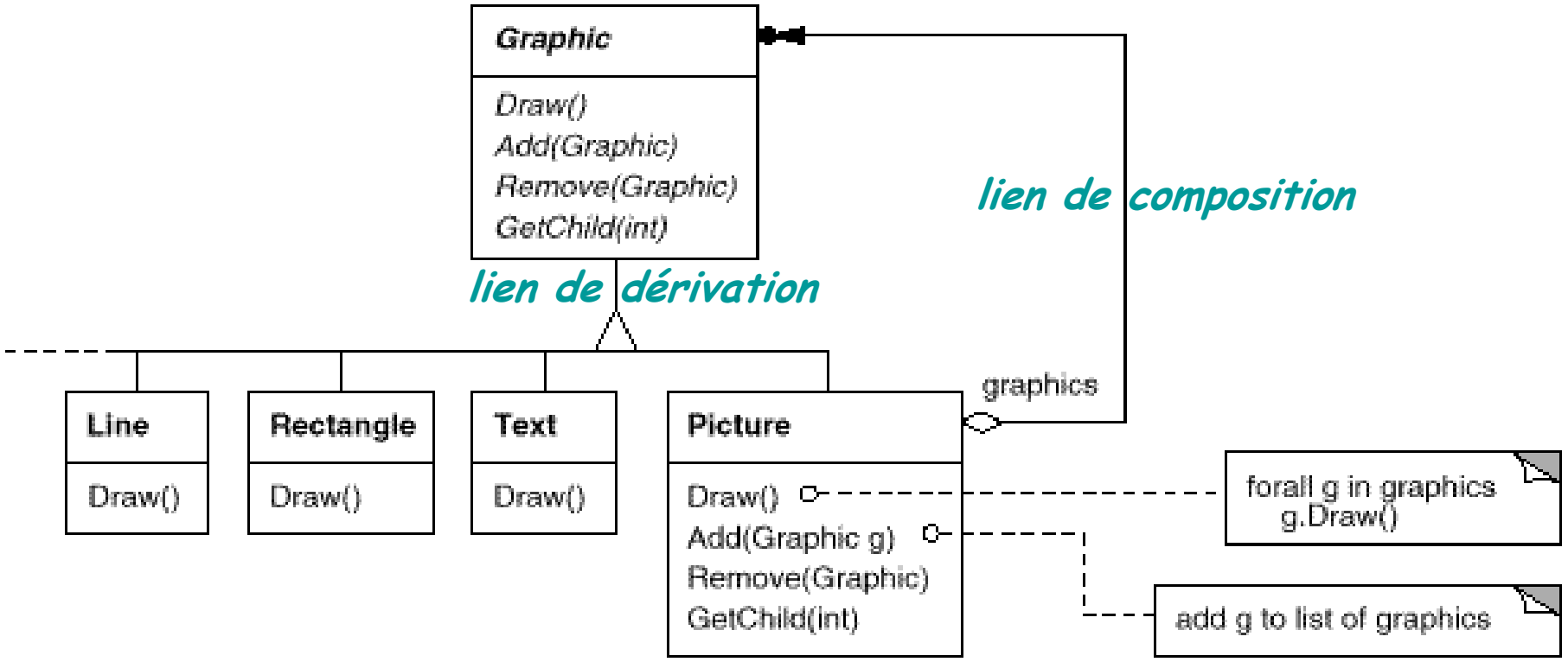
- Assembler (composer) des objets selon une structure d'arbre pour représenter des hiérarchies
- Manipuler de façon transparente, par une interface uniforme :
  - les objets **atomiques** (ou objets **simples**, ou **primitives**, ou **feuilles**)
  - les objets **composés** (ou **conteneurs**, ou **nœud non-terminal,** ou **non-terminal**)

Exemple de Besoin (Motivation)

- Un éditeur graphique permet à l'utilisateur de construire des diagrammes complexes en partant de figures simples
- L'utilisateur peut « grouper » des figures simples pour construire une figure plus complexe qui peut à son tour être regroupée avec d'autres figures pour former une figure encore plus complexe, …
- Une approche simple consisterait à définir des classes pour les primitives graphiques simples (texte, ligne, …) et d'autres classes qui seraient des « conteneurs ».
- **Problème** :
  - le code qui doit manipuler ces classes devra traiter de façon différente les « primitives » et les « conteneurs »
  - Cela rendrait donc le code plus complexe
- Le pattern **"composite"** décrit comment utiliser une composition récursive pour que le code qui utilise les différentes classes n'ait pas à distinguer ces classes

Exemple de Besoin (Motivation)

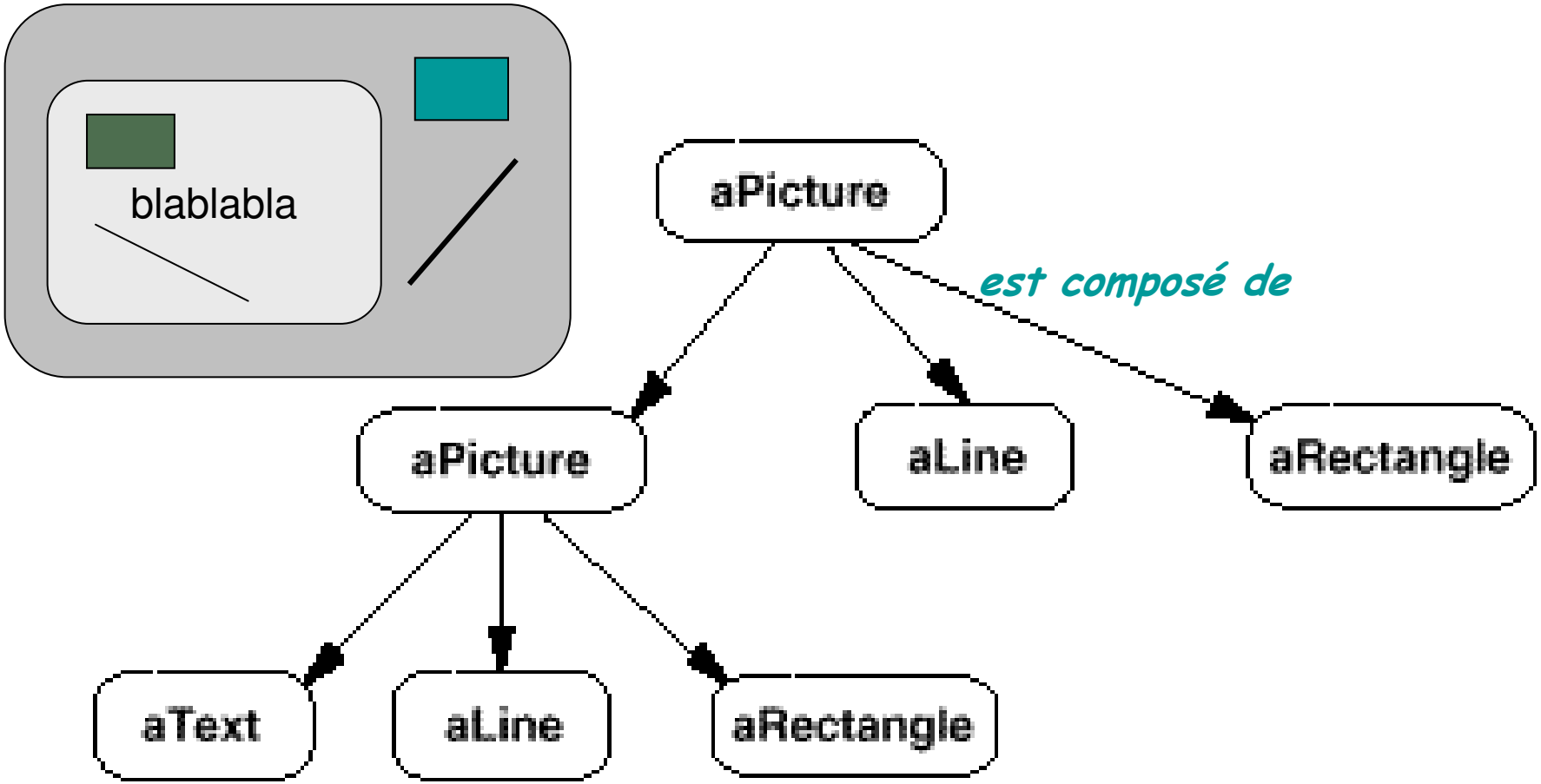
- objets graphiques composés récursivement d'objets graphiques



La Solution du Pattern Composite

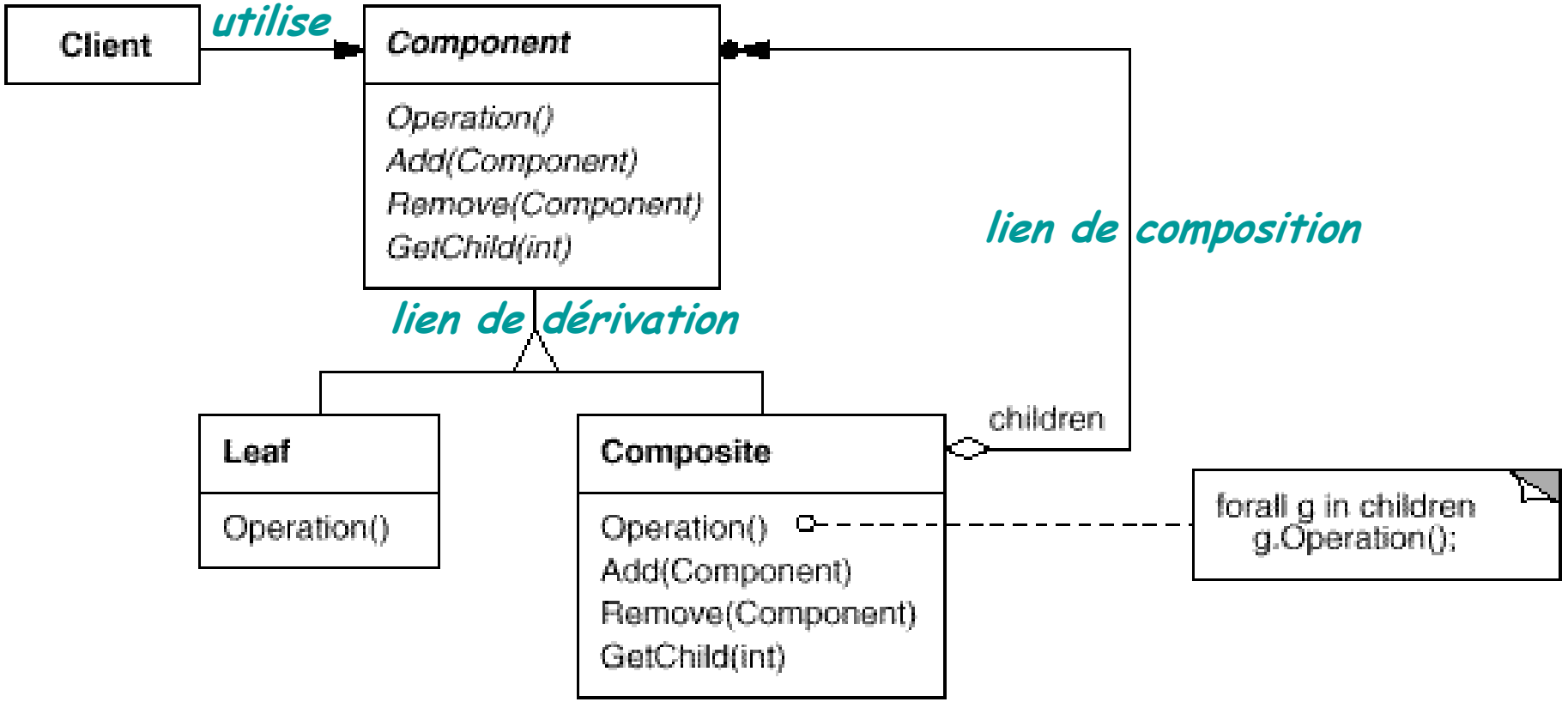
- Le pattern composite suggère de définir une classe abstraite pour représenter les **primitives** et les **conteneurs**.
- Dans le cas de cette application graphique, il s'agirait d'une classe **Graphic** qui déclare les opérations propres à tous les objets graphiques (exemple : Draw).
- Elle déclare aussi toutes les opérations que partagent les objets composés : accès aux composants et gestion de ceux-ci
- Les sous-classes (**Line**, **Rectangle**, et **Text**) représentent les primitives.
  - Ces classes implémentent la méthode Draw selon leur besoin.
  - Comme il s'agit de primitives, ces classes n'implémentent pas les méthodes relative à la gestion des composants.
- La classe **Picture** permet de représenter une composition d'objets graphiques.
  - **Picture** implémente Draw en appelant la méthode **Draw** de chacun de ses composants.
  - **Picture** implémente aussi les méthodes de gestion de ses composants
  - Puisque l'interface de Picture est conforme à l'interface de Graphic, ses instances peuvent être composé « récursivement » d'autres objets de classe **Picture**

Instanciations des objets pour cet exemple



La structure du pattern "*composite*"

- diagramme de classes selon la notation OMT (voir cours UML)



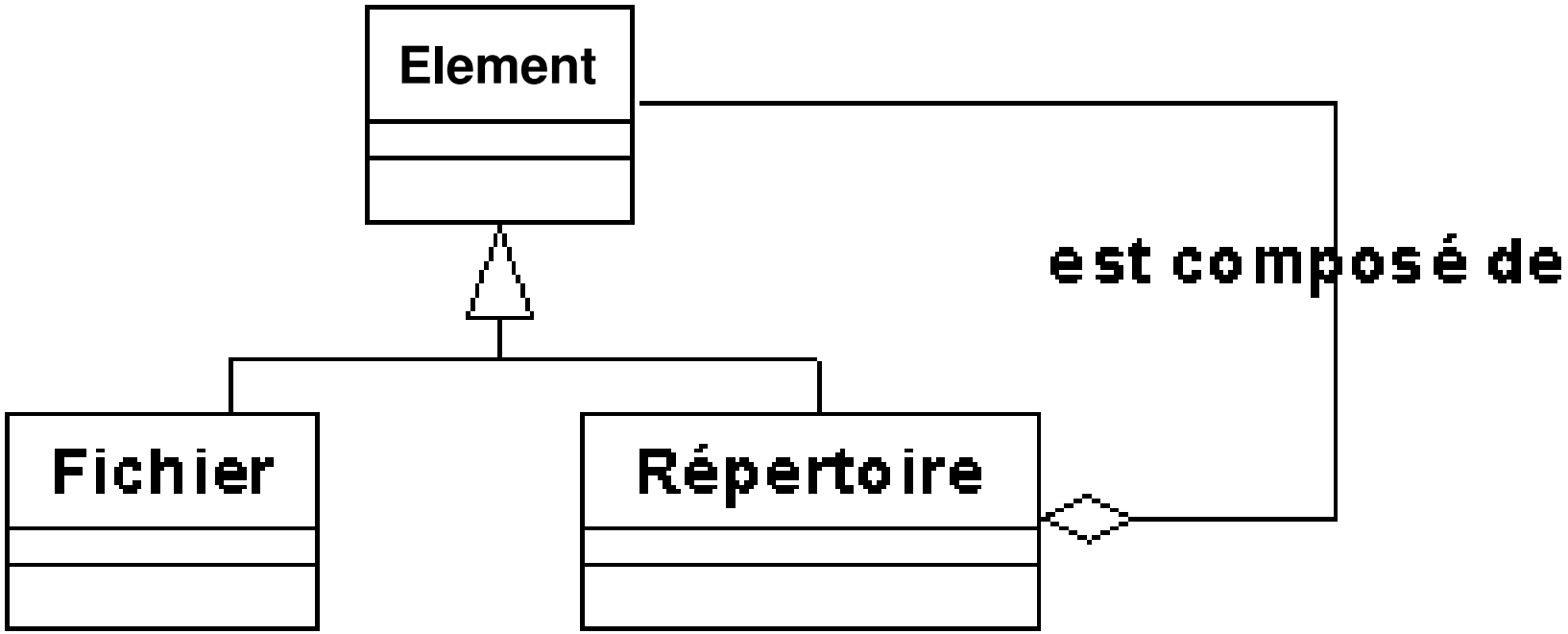
Les participants du pattern "*composite*"

- **Component (Composant)** (sur l'exemple : Graphic)
  - déclare l'interface des objets de la composition.
  - Implémente le comportement par défaut des méthodes de l'interface communes à toutes les classes.
- **Leaf (feuille)** (sur l'exemple : Rectangle, Line, Text, ...)
  - Représente les objets feuilles de la composition.
  - Définit le comportement des objets atomiques de la composition.
- **Composite (composé)** (sur l'exemple : Picture) .
  - mémorise les composants fils.
  - implémente les opérations liées au fils.
- **Client (vous, le développeur !)**
  - Utilisateur du pattern composite
  - Manipule tous les objets de la composition à travers l'interface visible dans la classe composant**.**

Exemple : un système de fichier (SF)

- On veut représenter les éléments d'un SF, c'est-à-dire une arborescence de **fichiers** et de **répertoires**.
- Pour chaque **fichier**, on souhaite mémoriser :
  - son nom
  - sa date de création.
- Pour chaque **répertoire**, on souhaite mémoriser :
  - son nom
  - la liste des fichiers et répertoires qu'il contient.
- Pour simplifier l'exemple, chaque élément possède seulement :
  - un constructeur qui donne les valeurs des attributs
  - la méthode afficher

Exemple d'utilisation du Pattern Composite
Représenter un Système de Fichiers



Classe Element (composant)

- Les **`fichiers`** et les **`répertoires`** ont en commun :
  - un attribut **`nom`**
  - une méthode pour **`afficher`** ce nom sur l'écran, **(méthode qui doit être virtuelle)**.
- D'après le pattern composite, il faut définir une classe abstraite, **`Element`**, qui définit les membres communs à tout objet **`fichier`** ou **`répertoire`**

Classe Fichier (feuille)

- La classe **`fichier`** dérivera de la classe **`Element`**
- Elle lui ajoutera l'attribut **`dateCreation`**
- Elle redéfinira la méthode **`afficher`**

Classe Repertoire (composé)

- Attributs d'un répertoire :
  - son **`nom`**
  - un tableau (tab) de pointeurs sur les éléments qui composent le contenu du répertoire.
- Méthodes d'un répertoire :
  - un constructeur :
    - qui donne le nom du répertoire
    - qui initialise le tableau à vide
  - une méthode pour **`afficher`** le contenu à l'écran
  - des méthodes pour **`ajouter`** un élément dans le tableau

Classe **Element (composant)**

```
class Element { // C'est la classe "composant" (component)
  protected:
    string nom; // Chaque élément du système de fichiers a un nom

  public:
    Element(string nom) {
      this->nom=nom;
    }

    virtual void afficher() {
      // Les opérations communes aux feuilles et aux non-terminaux
      //   sont visibles dans le composant
      // Si possible, on implémente déjà ici le comportement commun
      cout << "Nom  : " << nom << endl;
    }

    virtual void ajouter (Element *e) {
      // Les opérations propres aux non-terminaux doivent aussi être visibles ici
      // Mais on ne les implémente réellement que dans les non-teminaux
      throw new OperationInterdite; // Par défaut on lève une exception...
                                    // On pourrait aussi ne rien faire !
    }
};
```

classe OperationInterdite (exception)

```
class OperationInterdite : public exception {
  public:
    virtual const char* what() const throw() {
      return "Operation interdite sur une feuille";
    }
};
```

Classe **fichier (feuille)**

```
class Fichier : public Element {
  // C'est une classe "feuille" (leaf)
  private:
    string dateCreation;

  public:
    Fichier(string nom, string dateCreation) : Element(nom) {
      this->dateCreation=dateCreation;
    }

    void afficher() {
      // on redéfinit (si besoin) le comportement de afficher pour un fichier
      cout << "Fichier : " << endl;
      Element::afficher(); // pour afficher le nom, on réutilise...
      cout << "Date : " << dateCreation << endl << endl;
    }
};
```

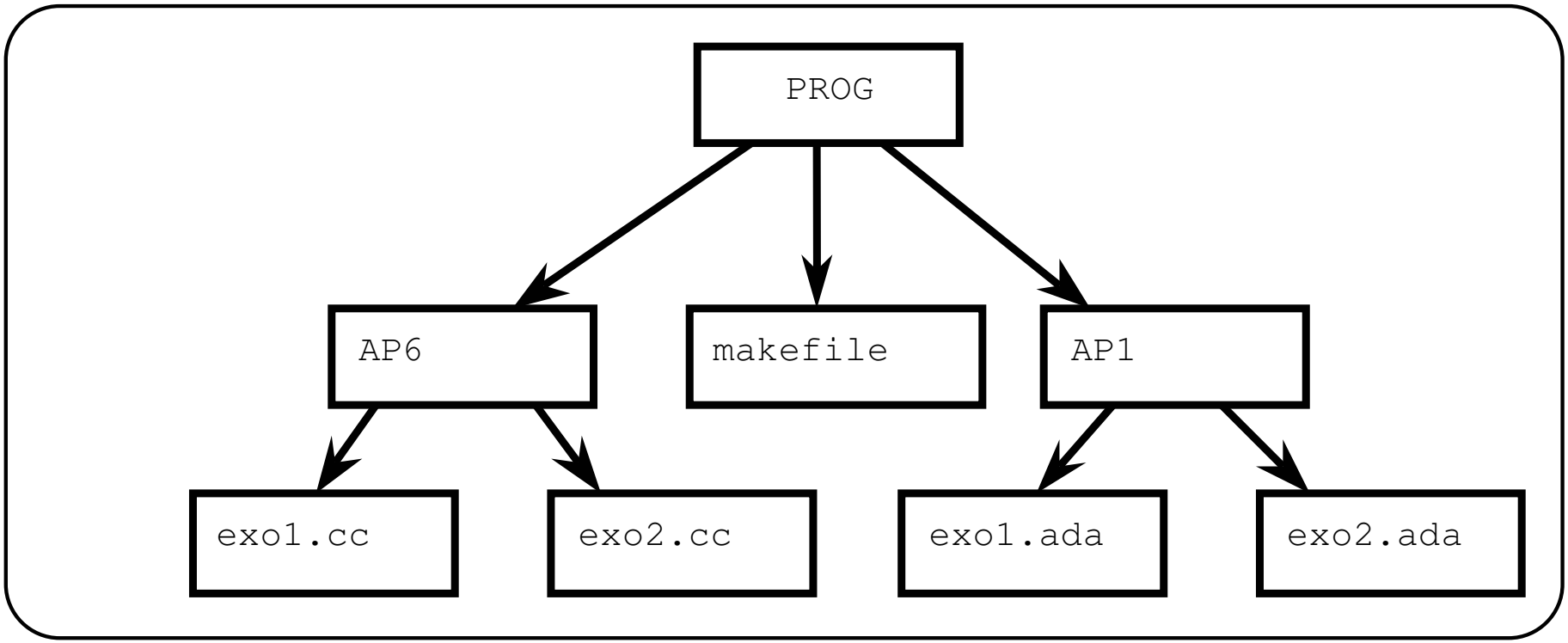
Classe **répertoire** (composé)

```
class Repertoire : public Element {
  // C'est une classe "composé" (composite)
  private:
    vector<Element *> contenu; // le conteneur d'éléments : un vecteur

  public:
    Repertoire(string nom) : Element(nom) {}

    void ajouter(Element * e) {
       // On implémente ici les opération de gestion des fils, comme l'ajout
       contenu.push_back(e);
    }

    void afficher() {
      // On redéfinit (si besoin) le comportement de afficher pour un répertoire
      cout << "Répertoire :" << endl;
      Element::afficher(); // on réutilise
      // afficher un "composé" consiste à afficher chacun de ses "composants"
      for (unsigned int i=0; i<contenu.size(); i++)
        contenu[i]->afficher();
    }
};
```

Exemple d'utilisation

```
int main() {
  Repertoire *   monRep = new Repertoire("PROG");
  Repertoire * sousRep1 = new Repertoire("AP6");
  Repertoire * sousRep2 = new Repertoire("AP1");
  sousRep1->ajouter(new Fichier("exo1.cc", "01/10/2008"));
  sousRep1->ajouter(new Fichier("exo2.cc", "08/10/2008"));
  sousRep2->ajouter(new Fichier("exo1.ada", "15/09/2007"));
  sousRep2->ajouter(new Fichier("exo2.ada", "22/09/2007"));
    monRep->ajouter(new Fichier("makefile", "11/09/2007"));
    monRep->ajouter(sousRep1);
    monRep->ajouter(sousRep2);
    monRep->afficher(); // affichera récursivement tout le contenu du répertoire

  try {
    // Le code ci-dessous est correct, mais sémantiquement stupide !
    Fichier * monFichier = new Fichier("bug.txt","04/11/2008");
    monFichier->ajouter(new Repertoire("exception")); // exception levée !!
  }
  catch ( exception * e) {
    cout << "Exception : " << e->what() << endl;
  }
}
```

Liens d'adressage (pointeurs) entre les objets créés



Puissance de ce type de programmation

- En réutilisant le *pattern composite* et le *template* **`vector`**, il nous a suffit d'écrire quelques lignes de codes...
- ... mais le programme ainsi réalisé est assez complexe :
  - un répertoire peut contenir d'autres répertoires
  - on manipule donc des arbres n-aires
- Le travail de programmation avec un langage classique aurait été beaucoup plus important