
Module R3.04

Chapitre 9

Design Patterns

Composite, Singleton

9.1. Design Pattern(1)

- un **Design Pattern** décrit un problème générique, récurrent, 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 réalise plus rapidement et plus « sainement » une application en utilisant des solutions qui ont déjà fait leurs preuves (réutilisation...)

9.1. Design Pattern(2)

Un Design Pattern permet :

- De trouver la bonne représentation « objet » et en particulier de bien choisir la granularité des objets
- De spécifier l'interface de ces objets
- De spécifier l'implémentation ces objets
- Et donc de réutiliser une (bonne) solution générique en l'adaptant à un besoin spécifique

La référence à connaître :

Design Patterns : Elements of reusable Object-Oriented Software.

**Erich Gamma, Richard Helm, Ralph Jonson, John Vlissides
(Gang OF 4)**

9.1. Eléments Principaux d'un « Pattern » (1)

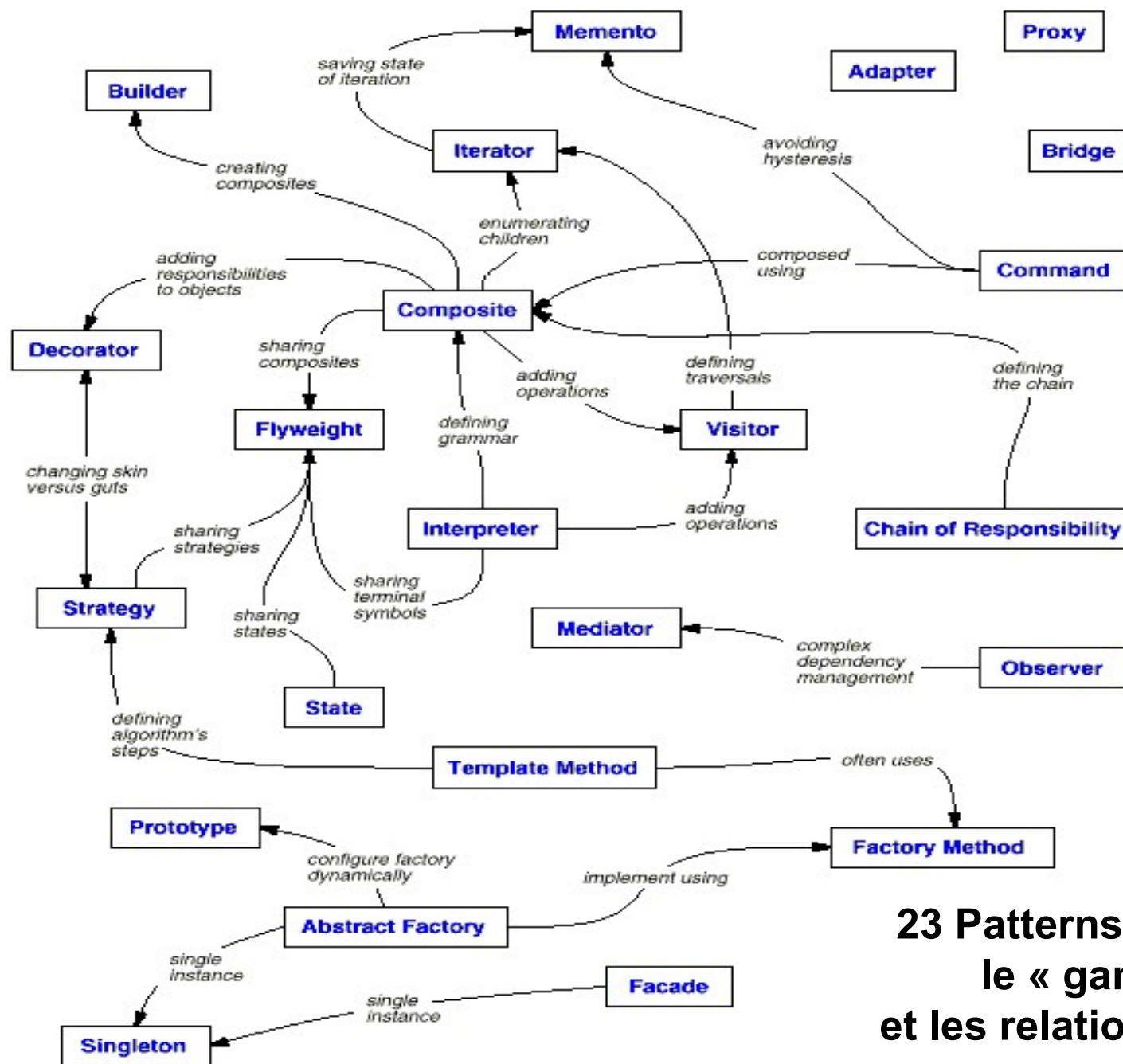
- Le **nom** du *pattern* permet de désigner en un ou deux mots un **problème** de conception, ses **solutions** et ses **conséquences**. Ce nom permet d'élargir le vocabulaire de conception et de réfléchir à un plus haut niveau.
- Le **problème** décrit dans quelles circonstances le pattern peut être 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*

9.1. Éléments Principaux d'un « Pattern » (2)

- 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 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
- Les **conséquences** 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

9.1. Chaque pattern 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**

9.2. Le pattern **Composite**

Objectif :

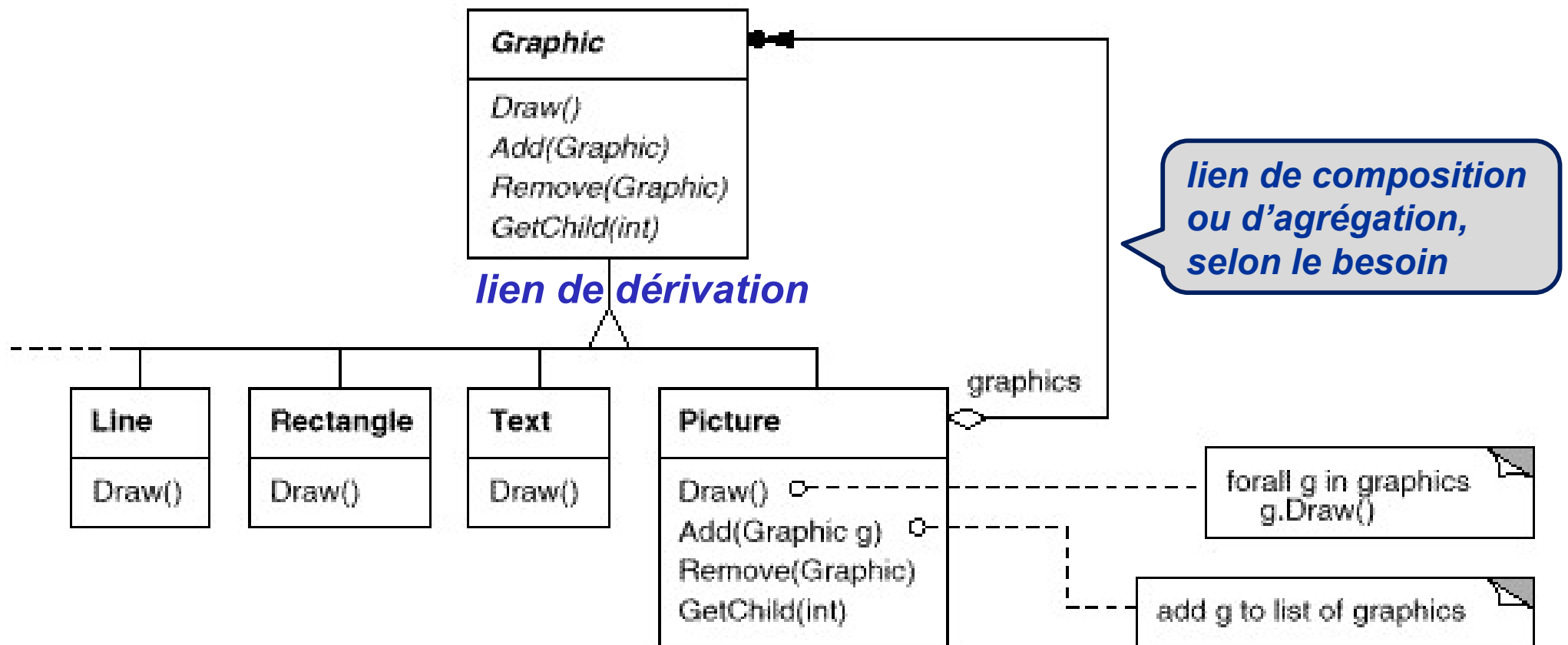
- Assembler (composer) des objets selon une structure d'arbre pour représenter des hiérarchies
- Manipuler de façon transparente, par une interface uniforme :
 - les objets **atomiques** (ou objets **simples**, ou **primitives**, ou **feuilles**)
 - les objets **composés** (ou **conteneurs**, ou **nœud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

9.2. 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

9.2. Exemple de Besoin (Motivation)

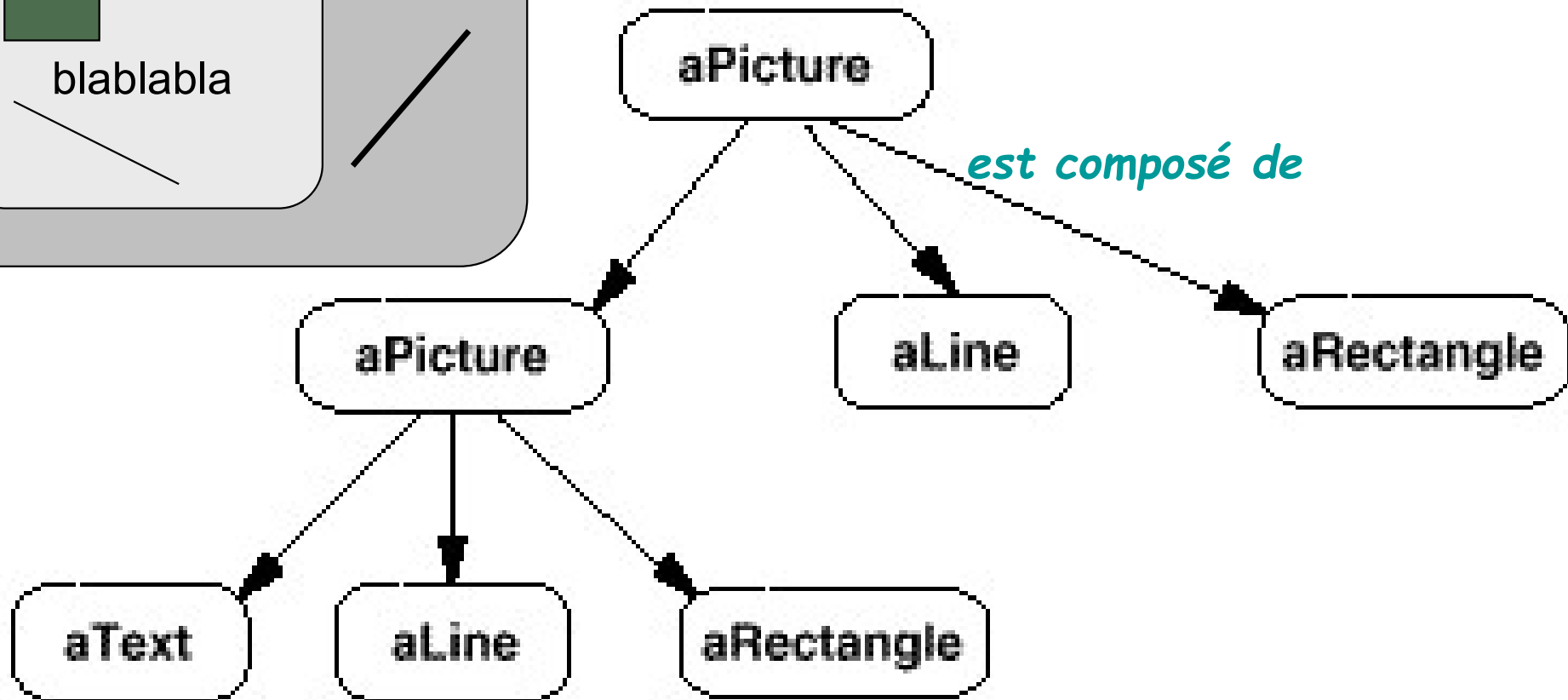
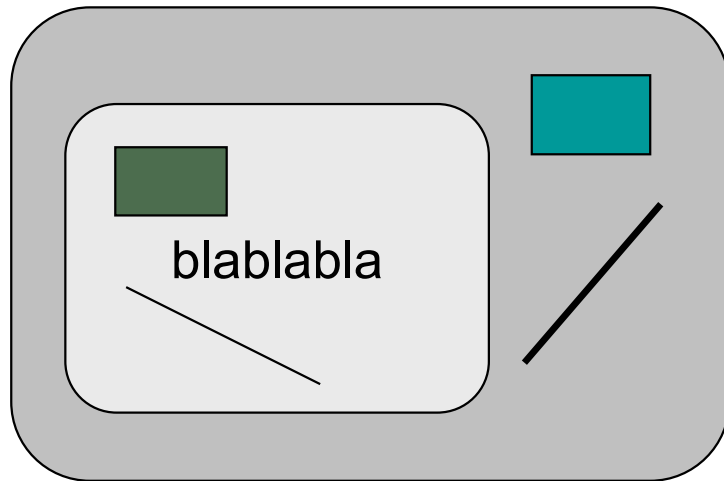
- Objets graphiques composés récursivement d'objets graphiques



9.2. La Solution du Pattern Composite

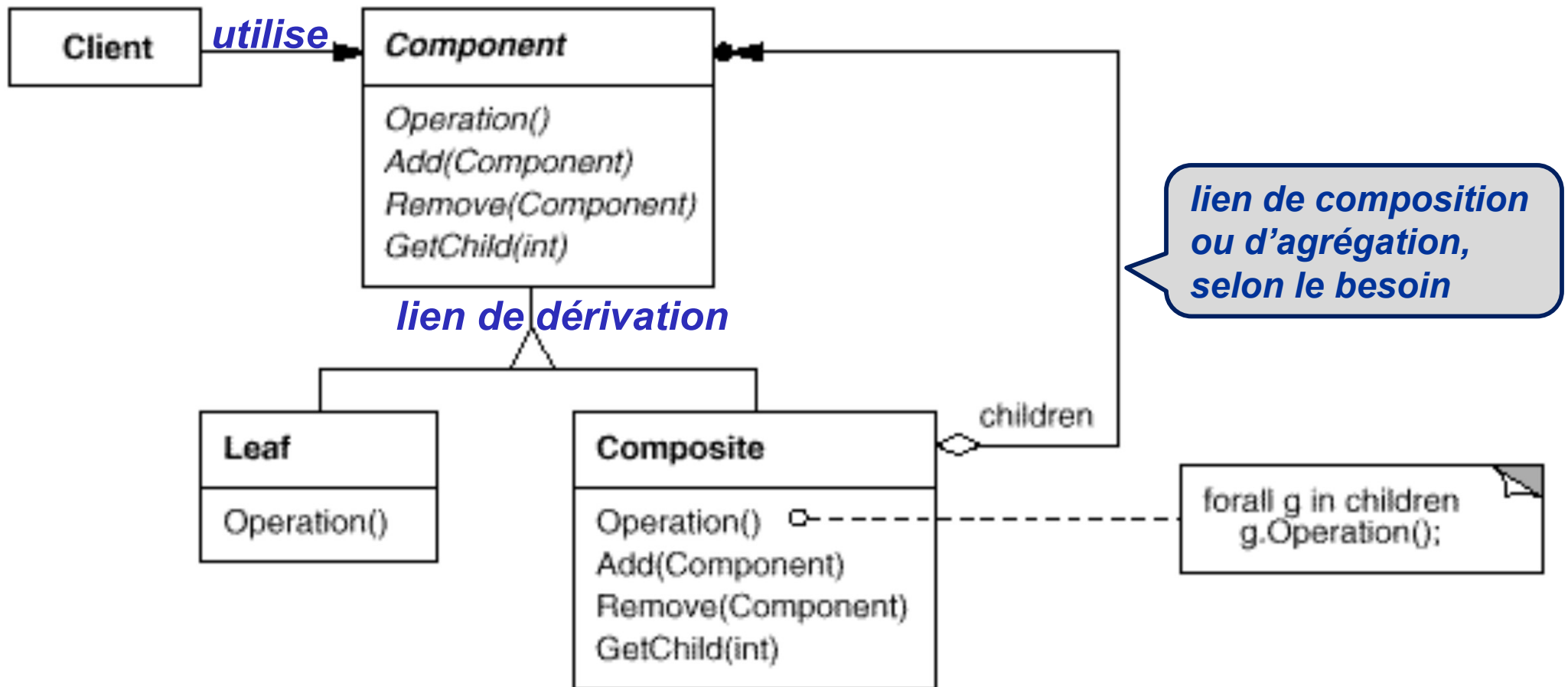
- Le pattern composite suggère de définir une classe abstraite pour représenter les **primitives** et les **conteneurs**.
- Dans le cas de cette application graphique, il s'agirait d'une classe **Graphic** qui déclare les **opérations** propres à tous les objets graphiques (exemple : **Draw**).
- Elle déclare aussi toutes les **opérations** (méthodes) que partagent les objets composés : accès aux composants et traitement 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 relatives 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 composées « récursivement » d'autres objets de classe **Picture**

9.2. Instanciations des objets



9.2. La structure du pattern "*composite*"

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



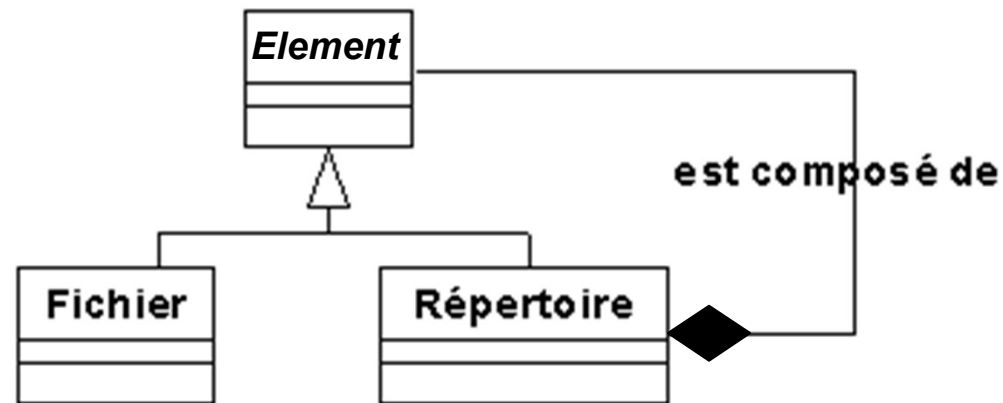
9.2. Les participants du pattern "*composite*"

- **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.

9.2. 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 initialise les attributs de l'élément
 - ❑ L'**opération** afficher
 - ❑ L'**opération** ajouter

9.2. Exemple : Représenter un Système de Fichiers



- 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**

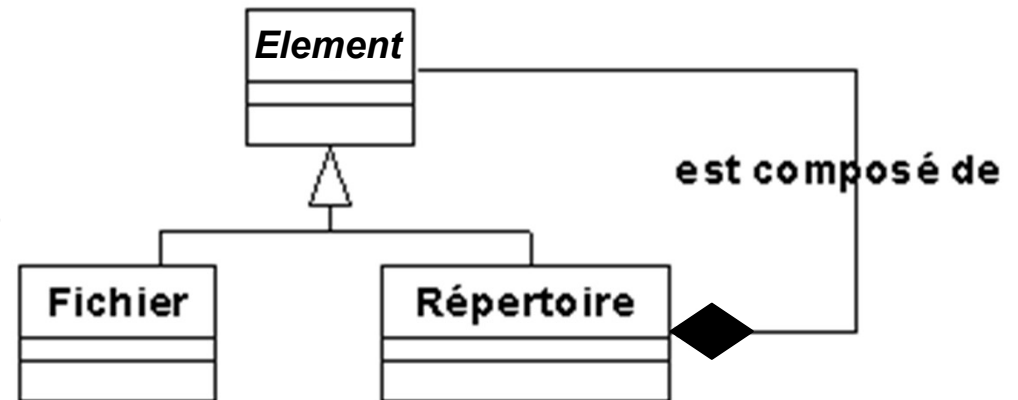
9.2. Classes **Fichier** et **Repertoire**

■ La classe **Fichier**

- Dérive de **Element**
- Ajoute l'attribut **dateCreation**
- Redéfinit l'opération **afficher**
- Redéfinit l'opération **ajouter**

■ La classe **Repertoire** :

- Dérive de **Element**
- Ajoute un attribut tableau (**m_contenu**) de pointeurs sur les éléments qui composent le contenu du répertoire.
- Définit un constructeur qui initialise le nom du répertoire et le tableau à « vide »
- Rédefinit l'opération **afficher** (affiche les caractéristiques du répertoire et son contenu)
- Redéfinit l'opération **ajouter** pour permettre l'ajout d'un **Element** au répertoire



9.2. Classe **Element** (composant)

```
class Element { // C'est la classe abstraite "composant" (component)
private:
    std::string m_nom; // Chaque élément du système de fichiers a un nom
public:
    Element(const std::string & nom : m_nom(nom){}

    const std::string & getNom const {return this->m_nom;} // virtual inutile

    virtual void afficher() const {
        // Les opérations communes sont visibles dans le composant
        // Si possible, on factorise déjà ici le code commun à tous les éléments
        std::cout << "Nom : " << this->getNom() << std::endl;
    }

    virtual void ajouter (Element * element) = 0 ;
        // Les opérations propres aux non-terminaux doivent aussi être visibles ici
        // Mais on ne les implémente réellement que dans les non-terminaux

    virtual ~Element() {}
};
```

= 0 : **méthode pure**
sans implémentation

En cas de polymorphisme, fournir
un destructeur virtuel, même vide

- En C++, une classe est réellement **abstraite** si elle comporte au moins une méthode pure
- Aucune instanciation d'un objet de classe abstraite n'est autorisée par le compilateur
- Une classe C++ ne contenant que des méthodes pures est équivalente à une interface en Java

9.2. Classe **Fichier** (feuille)

```
#include "Element.h"
class Fichier : public Element { // C'est une classe « feuille »

private:
    std::string m_dateCreation;

public:
    Fichier(const std::string & nom,
            const std::string & dateCreation)
        : Element(nom),
          m_dateCreation(dateCreation)
    {}

    void afficher() const override {
        std::cout << "Fichier : ";
        this->Element::afficher(); // réutilise l'op. afficher d'Element
        std::cout << "Date creation : "
                  << m_dateCreation << std::endl;
    }

    void ajouter (Element * element) override {
        throw "Opération Interdite sur un Fichier";
    }

    virtual ~Fichier() {}
};
```

- **override** signifie **redéfinition**
- Indication facultative
- Permet au compilateur de vérifier l'existence d'une méthode héritée ayant la même signature

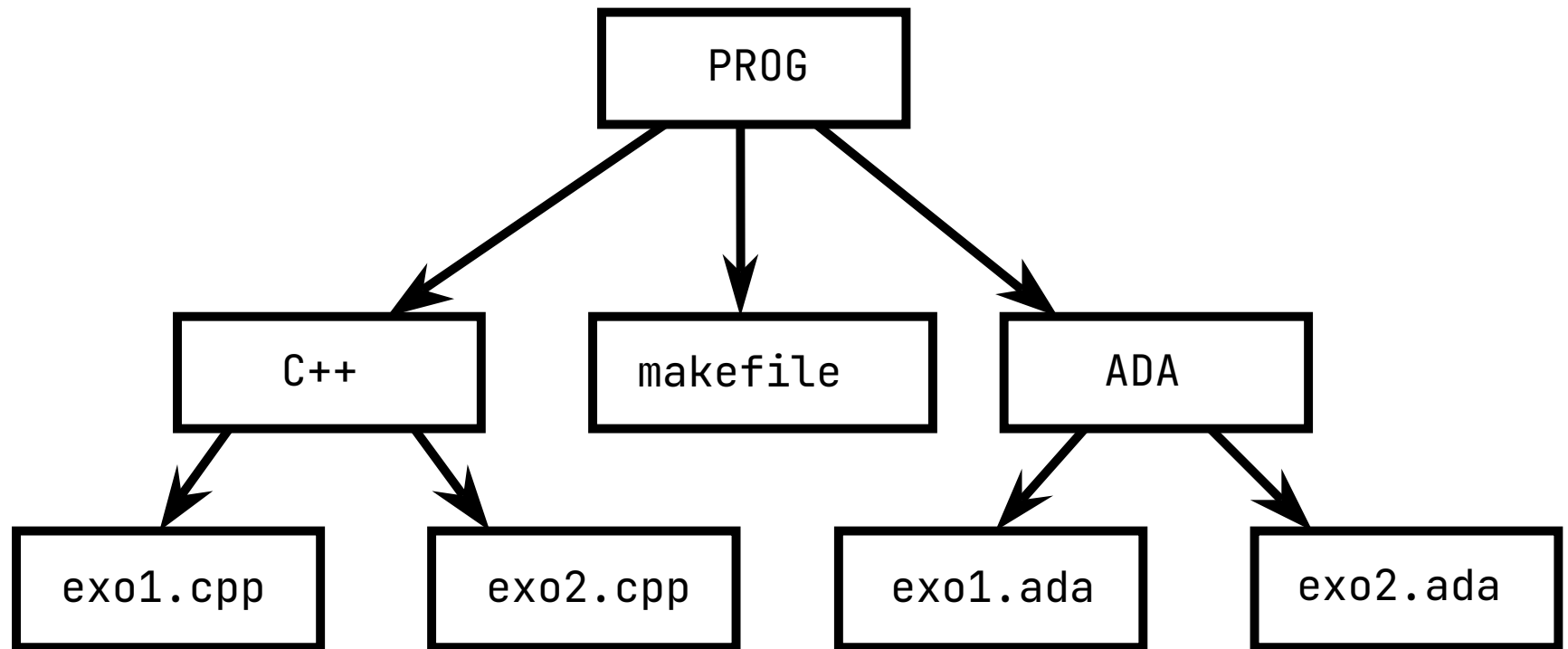
Une classe concrète qui dérive d'une classe abstraite doit redéfinir les méthodes pures dont elle hérite, sinon elle sera abstraite elle aussi...

9.2. Classe **Repertoire** (composite)

```
class Repertoire : public Element {  
    // C'est une classe "composée" (composite)  
private:  
    std::vector<Element *> m_contenu; // le conteneur d'éléments : un vecteur  
  
public:  
    Repertoire(const std::string & nom) : Element(nom) {}  
  
    void ajouter(Element * e) override {  
        // On implémente ici les opération de gestion des fils, comme l'ajout  
        this->m_contenu.push_back(e);  
    }  
  
    void afficher() const override {  
        // On redéfinit le comportement de afficher pour un répertoire  
        std::cout << "Répertoire :" << std::endl;  
        this->Element::afficher(); // réutilise l'op. afficher héritée d'Element  
        // afficher un "composé" consiste à afficher chacun de ses "composants"  
        for (Element* ptrElement : m_contenu) ptrElement->afficher();  
    }  
  
    virtual ~Repertoire() {  
        for (Element* ptrElement : m_contenu) delete ptrElement; }  
};
```

Ici le polymorphisme doit fonctionner
C'est pour cela que les opérations
doivent être déclarées « **virtual** »

9.2. Liens entre les objets



9.2. Exemple d'utilisation

```
int main() {  
    Repertoire * monRep = new Repertoire("PROG");  
    Repertoire * sousRep1 = new Repertoire("C++");  
    Repertoire * sousRep2 = new Repertoire("ADA");  
    sousRep1->ajouter(new Fichier("exo1.cpp", "01/10/2020"));  
    sousRep1->ajouter(new Fichier("exo2.cpp", "08/10/2020"));  
    sousRep2->ajouter(new Fichier("exo1.ad", "15/09/2019"));  
    sousRep2->ajouter(new Fichier("exo2.ad", "22/09/2019"));  
    monRep->ajouter(new Fichier("makefile", "12/10/2020"));  
    monRep->ajouter(sousRep1);  
    monRep->ajouter(sousRep2);
```

Le Pattern Composite, qui utilise le polymorphisme, permet de réaliser une opération récursivement sur toute l'arborescence « sans s'en apercevoir »

```
monRep->afficher(); // Affichera récursivement tout le contenu du répertoire
```

```
try { // Ici on essaye bêtement d'ajouter un répertoire à un fichier..  
    Fichier * monFichier = new Fichier("bug.txt", "01/05/2020");  
    monFichier->ajouter(new Repertoire("exception")); // ... exception levée !  
}  
catch ( const char * exception) {  
    cout << "Exception : " << exception << endl;  
}  
delete monRep; // Supprimera tout le système de fichier  
return 0;  
}
```

9.2. Réutilisation du code & des idées

■ En réutilisant...

- Un modèle de conception (le **Pattern Composite**)
 - Un composant générique (le **template vector**)
- ... il nous a suffi d'écrire quelques lignes de code !

■ Le traitement ainsi réalisé est pourtant assez complexe :

- un répertoire peut contenir d'autres répertoires
- on manipule donc implicitement des arbres n-aires

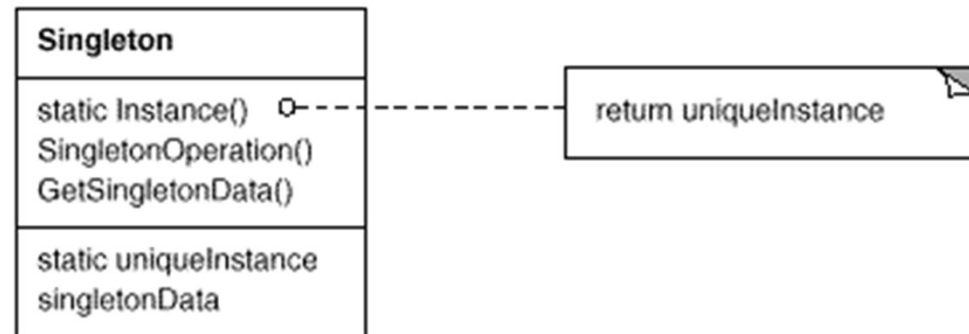
■ Le travail de programmation avec un langage impératif aurait été bien plus conséquent !

9.3. Le Pattern **Singleton** (1)

- **Objectif**
 - S'assurer qu'une classe ne peut avoir qu'une seule instance et fournir un point d'accès à cette instance
- **Motivation**
 - Dans certain cas il est indispensable d'être sûr qu'il n'existe qu'une seule instance d'une classe donnée, par exemple si cet classe modélise un objet « système » dans une application : un gestionnaire de fenêtres, un gestionnaire de son, un spooler d'imprimante, ...
 - Il serait possible d'offrir cette instance unique par le biais d'une variable globale (pas très joli...) mais rien n'empêcherait qu'une autre instance soit créée par erreur quelque part dans l'application
 - La bonne solution consiste à rendre la classe responsable de créer et fournir l'accès à son unique instance.

9.3. Le Pattern Singleton (2)

- Structure




- Les participants

- Le singleton lui-même... Il crée une opération **Instance()** qui donne à l'utilisateur de la classe un accès à l'unique instance de cette classe.
- **Instance()** sera une méthode de classe (**static**)
- La classe sera responsable de créer cette instance unique

9.3. Le Pattern Singleton (3)

- Exemple d'Implémentation

Singleton.h	Singleton.cpp
<pre>class Singleton { public: static Singleton& getInstance(); protected: Singleton(); //constr. protected private: static Singleton* m_instance; //... + membres d'instance nécessaires };</pre> 	<pre>Singleton* Singleton::m_instance = nullptr; Singleton::Singleton():... {...} //constr. Singleton& Singleton::getInstance () { if (m_instance == nullptr) { m_instance = new Singleton; } return *m_instance; } //... + méthodes d'instance nécessaires</pre>

- Le constructeur est « protected »** : un utilisateur de la classe Singleton ne pourra pas instancier lui-même un objet Singleton
- Pour accéder à l'unique instance de Singleton**, on écrira par ex. :
`Singleton & leSingleton = Singleton::getInstance();`
- Attention** : dans un programme « multi-thread » il faudrait rajouter de l'exclusion mutuelle dans le constructeur pour garantir l'unicité du singleton !