# UML et C++ : Composition et Agrégation

## La Composition (Rappel):

Dans la vie réelle, des objets complexes sont souvent construits à partir de petits objets simples. Par exemple, une voiture est construite en utilisant une structure métallique, un moteur, des pneus, une transmission, un volant de direction, et un grand nombre d'autres parties.

Un ordinateur personnel est construit à partir d'un processeur, d'une carte mère, de la mémoire, etc ...

**Ce processus de construction d'objets complexes à partir de plus simples est appelé composition.**

Plus précisément, la composition est utilisée pour les objets qui ont une relation de type **un à l'autre**. Une voiture a **un** châssis métallique, a **un** moteur, et a **une** transmission.

Un ordinateur personnel a **un** processeur, **une** carte mère et d'autres composants.

Jusqu'à présent, toutes les classes que nous avons utilisées dans nos exemples possédaient des variables membres qui appartenaient à un type de données simple (par exemple int, double).
Afin de faciliter la construction de classes complexes, C + + nous permet d'utiliser des classes en tant que variables membres dans d'autres classes.

## Exemple :

Si nous concevons une classe **PersonnalComputer** , nous pourrions la créer de cette manière (en supposant que nous avons déjà écrit une classe **CPU**, une classe **Motherboard** et la classe **RAM**):

```
#include "CPU.h"
#include "Motherboard.h"
#include "RAM.h"

class PersonalComputer
{
private:
   CPU m_cCPU;
   Motherboard m_cMotherboard;
   RAM m_cRAM;
};
```

## Initialisation des variables membres de classe

Nous allons écrire un constructeur de notre classe **PersonalComputer** qui utilise une liste d'initialisation pour initialiser les variables membres.

Ce constructeur prend 3 paramètres:

- une vitesse de processeur,
- un modèle de la carte mère,
- et une taille de RAM.

```
PersonalComputer::PersonalComputer(int nCPUSpeed, char *strMotherboardModel, int nRAMSize)
    : m_cCPU(nCPUSpeed), m_cMotherboard(strMotherboardModel), m_cRAM(nRAMSize)
{

}
```

Maintenant, lorsqu'un objet **PersonalComputer** est instancié à l'aide de ce constructeur, cet objet **Personalcomputer** contiendra un objet **CPU** initialisé avec nCPUSpeed, un objet **Motherboard** initialisé avec strMotherboardModel, et un objet **RAM** initialisé avec nRAMSize.

## Important :

Il est important de noter explicitement que la composition implique la propriété entre la classe complexe et les sous-classes.

- **Lorsque la classe complexe est créée, les sous-classes sont créées**.
- **Lorsque la classe complexe est détruite, les sous-classes sont détruites**.

## Autre exemple :

- **La classe point2D**

```
#pragma once
#include<iostream>
#include<string>

using namespace std;

//une classe pour pouvoir afficher un Point2D
class point2D
{
    //un point a deux coordonnées :
    int x;
    int y;
public:
    //Constructeur qui prend deux valeurs :
    point2D(int,int);
    //destructeur :
    ~point2D();
    //récuperer l'ordonnée :
    int getX();
```

```
        //modifier l'ordonnée :
        void setX(int);
        //récuperer l'abscisse :
        int getY();
        //modifier l'abscisse :
        void setY(int);
        //afficher les coordonnées du point :
        void affichepoint();
};
```

- **La classe Robot.h**

```
#pragma once
#include<iostream>
#include<string>
using namespace std;
#include"point2D.h"

//Classe pour gérer le robot :
class Robot
{
    //un nom, une position et un niveau d'essance.
    string nom;
    point2D position;
    int niveau;
public:
    Robot(string,point2D,int);
    ~Robot();
    void allerAGauche();
    void allerADroite();
    void allerEnHaut();
    void allerEnBas();
    point2D getPosition();
    void sePresenter();
    int getNiveau();
};
```

- **Le constructeur de la classe Robot**

```
Robot::Robot(string nom,point2D p,int niveau):position(p.getX(),p.getY())
{
    this->nom=nom;
    this->niveau=niveau;
};
```

- **Le programme principal :**

```
#include <iostream>
#include "Jeu.h"
#include "point2D.h"

using namespace std;

int  main()
{
    point2D position(0,0);
    Robot *rob;

    rob = new Robot("Goldorak", position,100);
    rob->sePresenter();
    rob->allerADroite();
    rob->allerAGauche();
    rob->allerEnBas();
    rob->allerEnHaut();

    return 0;
}
```

## Pourquoi utiliser la composition?

Au lieu d'utiliser la classe Point2D pour connaitre l'emplacement du robot, nous aurions pu simplement ajouter deux entiers à la classe Robot.

L'utilisation de la composition nous fournit un certain nombre d'avantages:

- Chaque classe peut être maintenue simplement, chaque classe réalise une tâche. Cela rend les classes plus simple à écrire et beaucoup plus facile à comprendre
- Chaque classe est réutilisable. Par exemple, nous pourrions réutiliser notre classe Point2D dans une application complètement différente.
- La classe complexe peut avoir des sous-classes simples qui réalisent le travail, la classe complexe se concentre sur la coordination du flux de données entre les sous-classes.

# L'agrégation :

Une **agrégation** est un type spécifique de composition où il n'existe aucun droit de propriété entre l'objet complexe et les sous-objets.

Quand un **agrégat est détruit**, les **sous-objets ne sont pas détruits**.

Par exemple, considérons le département de mathématiques d'une école où travaillent plusieurs enseignants. L'école ne possède pas les enseignants (ils travaillent simplement là), l'école devrait être un agrégat. Lorsque l'école est détruite, les enseignants existent toujours (ils peuvent aller travailler dans une autre école).

Parce que les agrégations sont un type spécial de compositions, ils sont mis en œuvre presque à l'identique, la différence entre eux est surtout sémantique.

Dans une composition, on ajoute généralement des sous-classes en utilisant soit des variables normales ou des pointeurs où le processus d'allocation /libération de la mémoire est réalisé par la classe de composition.

Dans une agrégation, nous ajoutons également d'autres sous-classes dans notre classe complexe comme variables membres. Cependant, ces variables membres sont généralement soit des références ou des pointeurs qui sont utilisés pour pointer vers des objets qui ont été créés à l'extérieur du champ d'application de la classe complexe.

Parce que les objets de la classe complexe vivent en dehors de la portée de la classe complexe, lorsque la classe complexe est détruite, la variable pointeur ou référence sur une classe membre sera détruite, mais les objets continueront d'exister.

## Exemple :

```
class Enseignant
{
    private:
        string strNom;
    public:
        Enseignant(string Name):strNom(Name)
        {

        }
        // ou
         Enseignant(string Name)
        {
            this->strNom = Name;
        }

    string GetName() { return strNom; }
};
```

```
class Ecole
{
    private:
        Enseignant *ptrEnseignant; //

    public:
        Ecole(Enseignant *pcTeacher=NULL)
            : ptrEnseignant(pcTeacher)
        {
        }
};

int main()
{
    // Creation d'un enseignant hors de la portée de l'objet uneEcole
    Enseignant *pProf = new Enseignant("Bob");
    {
        // Création d'une école et utilisation du constructeur
        // paramétré. Passage de Bob comme argument.
        Ecole uneEcole(pProf);

    } // uneEcole est hors de portée ici et est donc supprimée

    // pProf existe ici car uneEcole ne l'a pas detruit.
    delete pProf;
}
```

# Pour résumer:

## Composition:

- Utilise généralement des variables membres normaux.
- Peut utiliser des pointeurs si la classe de composition gère automatiquement l'allocation/ désallocation de l'objet composite.
- Responsable de la création / destruction des objets composites.

## Agrégation:

- Utilise généralement des variables de type pointeur qui pointent vers un objet qui vit en dehors de la portée de la classe globale
- Peut utiliser des valeurs de type référence d' objet qui vivent en dehors de la portée de la classe complexe
- Pas responsable de la création des sous-classes / détruire