Cours UML/C++ : les relations entre classes

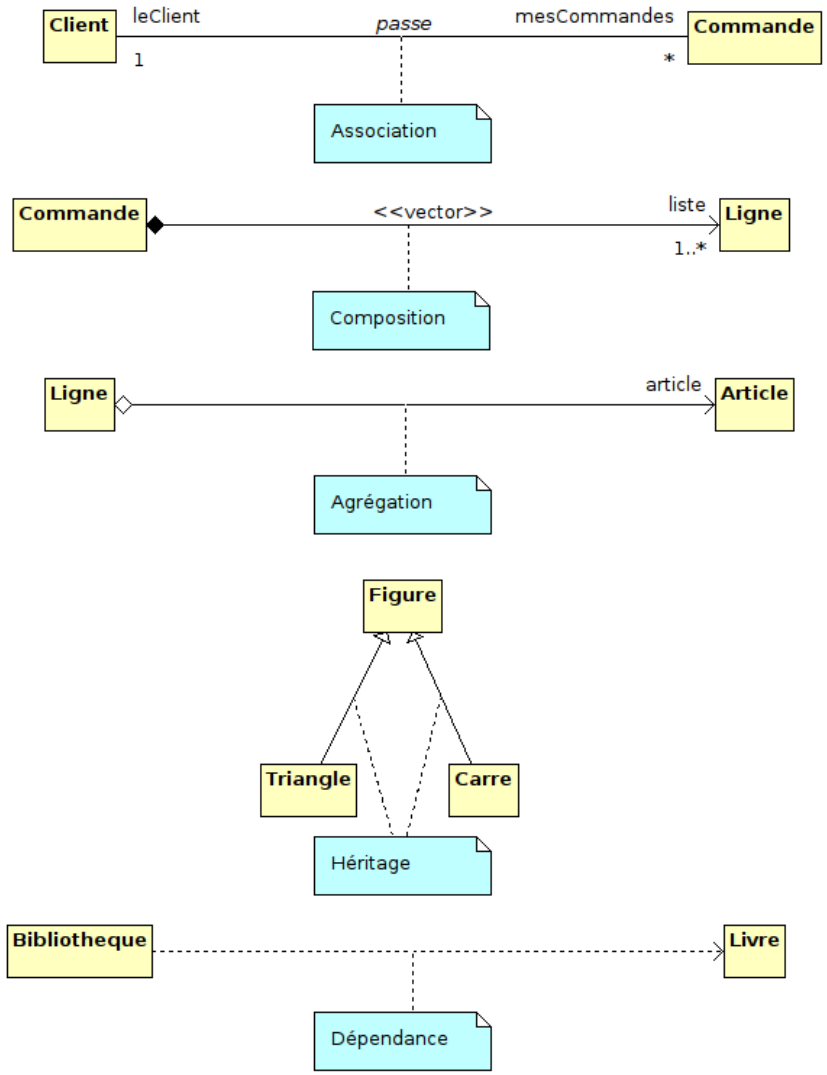
Les différents types de relations entre classes (UML/C++)

Étant donné qu'en POO (Programmation Orientée Objet), les objets logiciels interagissent entre eux, il y a donc des relations entre les classes.

ATTENTION : Il s'agit de décrire les relations entre classes pas entre objets (pour ça, il existe le diagramme d'objets)

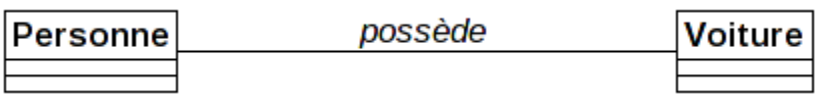
On distingue cinq différents types de relations de base entre les classes (nous ne verrons dans ce cours que les trois premières) :

- l'**association** (trait plein avec ou sans flèche)
- la **composition** (trait plein avec ou sans flèche et un losange plein)
- l'**agrégation** (trait plein avec ou sans flèche et un losange vide)
- la relation de généralisation ou d'**héritage** (flèche fermée vide)
- la **dépendance** (flèche pointillée)



Il existe des relations durables (association, composition, agrégation, héritage) et des relations temporaires (dépendance).

Exemple d'association entre classes : Une *Personne* possède une *Voiture*. La relation *possède* est une association entre la classe *Personne* et la classe *Voiture*.



1. La relation d'association

Une association représente une relation sémantique durable entre deux classes. Les associations peuvent donc être nommées pour donner un sens précis à la relation.
L'association se représente de la manière suivante en UML et son équivalent en C++ :

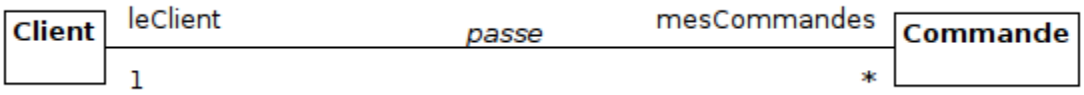
UML	C++

	`class E`
	`{`
	`  private:`
	`  F * monF;`
	`};`
	`class F`
	`{`
	`  private:`
	`  E * monE;`
	`};`

La relation est bidirectionnelle (pas de flèche), on a une navigabilité dans les deux sens. Donc « A » connait « B » et « B » connait « A ».
En C++, les objets de type E ont un pointeur sur un objet de type F. Les objets de type F ont un pointeur sur un objet de type E.
Attention : ça n'est pas forcément le même objet !
Exemple : imaginons qu'on veuille représenter l'association entre la classe « Pilote » et la classe « Voiture ». Un pilote peut conduire une voiture. L'objet P1 « Pilote » possède un pointeur vers un objet V1 « Voiture ». Le pointeur de l'objet V1 « Voiture », lui, n'est pas « obligé » de pointer vers l'objet P1 « Pilote ». Cela permet à une voiture d'être pilotée par d'autres pilotes.
Aux extrémités d'une association, agrégation ou composition, il est possible d'y indiquer une multiplicité (ou cardinalité) : c'est pour préciser le nombre d'instances (objets) qui participent à la relation. Dans l'exemple ci-dessus, l'association est implicitement de 1 vers 1.

- Une multiplicité peut s'écrire : n (exactement n, un entier positif), n..m (n à m), n..* (n ou plus), * (plusieurs)

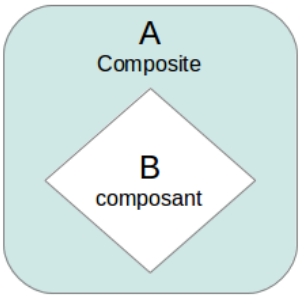
Ce diagramme de classe ci-dessous illustre une relation d'association 1 vers plusieurs (*) entre Client et Commande :



- On utilisera les accesseurs *get* et *set* pour mettre en place la relation.

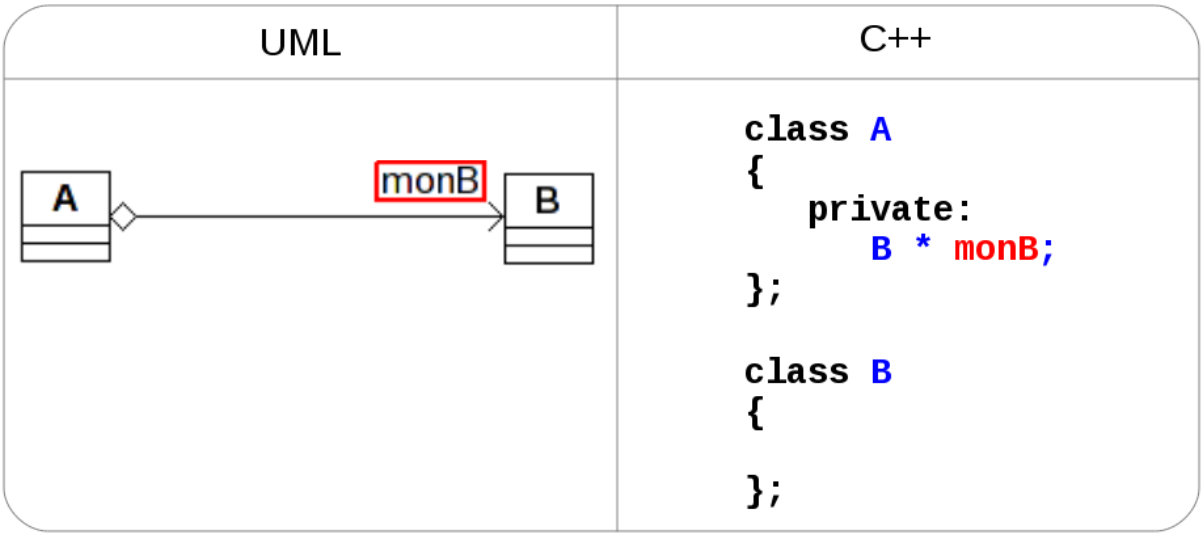
2. La relation d'agrégation

Une agrégation est un cas particulier d'association non symétrique exprimant une relation de contenance. Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « contient » ou « est composé de ».



Agrégation

A est le **composite** et B le **composant**. Dans une agrégation, le composant peut être partagé entre plusieurs composites, ce qui entraîne que, lorsque le composite A sera détruit, le composant B ne le sera pas forcément.



À l'extrémité d'une association, agrégation ou composition, on donne un nom : c'est le **rôle** de la relation. Par extension, c'est la manière dont les instances d'une classe voient les instances d'une autre classe au travers de la relation. Ici **l'agrégation** est nommée monB qui se traduit par un attribut de type pointeur sur un B dans la classe A.

- La flèche sur la relation indique la navigabilité. A « connaît » B mais pas l'inverse. Les relations peuvent être bidirectionnelles (pas de flèche) ou unidirectionnelles (avec une flèche qui précise le sens).

Le diagramme de classe ci-dessous illustre la relation d'agrégation entre la classe Ligne et la classe Article :

- Les accesseurs *getArticle()* et *setArticle()* permettre de gérer la relation *article*

Il s'agit bien d'une relation de type *agrégation* et non *compostion* (voir plus loin). En effet, si on supprime une ligne d'une commande, il n'est pas concevable de supprimer l'article associé ! De plus

l'article doit pouvoir se retrouver dans plusieurs lignes de commandes de plusieurs commandes (heureusement pour le vendeur).

C++

```
class Ligne
{
   public:
      // ...
      void setArticle(Article *article); // met
en œuvre l'agrégation

   private:
      // ...
      Article *article; // l'agrégation
};
```

UML

La déclaration (incomplète) de la classe Ligne intégrant la relation d'agrégation sera :

```
#ifndef LIGNE_H
#define LIGNE_H
class Article;    //je ''déclare'':Article est une classe!(1)

class Ligne
{
    private:
        Article *article;    //l'agrégation
    public:
        Ligne();
        Article *getArticle() const;
        void setArticle(Article *article);
};

#endif // LIGNE_H
```

(1) est obligatoire pour indiquer que *Article* est un type *class* et éviter un message d'erreur à la compilation. Cela ne remet pas en question la déclaration de *Article* ailleurs dans un autre .h

```
#include <iostream>
#include <iomanip>
#include "ligne.h"
#include "article.h" //accès à la déclaration complète de la classe
Article (2)

using    namespace    std;

Ligne::Ligne()
{
    this->article = NULL;    //initialise la relation d'agrégation
}

Article *Ligne::getArticle()    const
{
    return article;
}

void Ligne::setArticle(Article *article)
{
    this->article = article;
```

```
}
```

(2) sans cette ligne, on aura une erreur de compilation. il faut inclure le fichier « article.h » pour que le compilateur connaisse le type *Article* complétement.

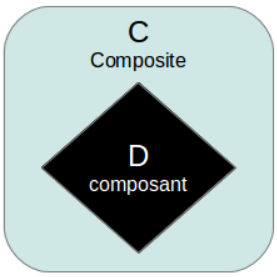
Voici un exemple d'utilisation de ces deux classes :

```
#include <iostream>
#include "ligne.h"
#include "article.h"

using namespace std;

int main()
{
    Article    a1;    //    un    objet    Article
    Ligne    l1;    //    un    objet    Ligne
    l1.setArticle(&a1); // met en place l'agrégation entre l1 et a1

    return 0;
}
```

3. La relation de composition

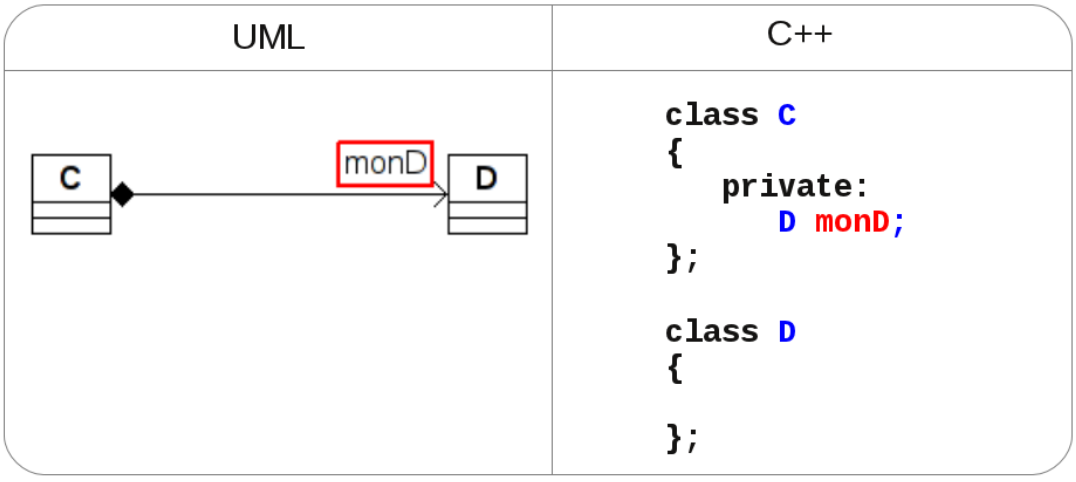
Une composition est une agrégation plus forte signifiant « est composée d'un » et impliquant :

- une partie ne peut appartenir qu'à un seul composite (agrégation non partagée)
- la destruction du composite entraîne la destruction de toutes ses parties (il est responsable du cycle de vie de ses parties).

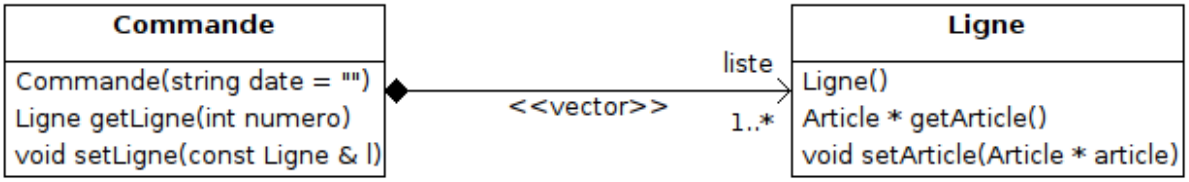


Composition

La composition se représente de la manière suivante en UML et C++ :



Le diagramme de classe ci-dessous illustre la relation de composition entre la classe *Commande* et la classe *Ligne* :



- La relation de composition correspond assez bien à la situation ici pusiqu'une commande est composée de plusieurs lignes et ces lignes ne peuvent pas être partagées avec d'autres commandes. De plus, si on **détruit** une commande alors on **détruit** toutes les lignes de cette commande

Dans notre cas, une commande peut contenir une (1) ou plusieurs (*) lignes. Pour pouvoir conserver plusieurs lignes (c'est-à-dire plusieurs objets Ligne), on va utiliser un conteneur de type vector (indiqué dans le diagramme UML ci-dessus par un stéréotype). On aurait pu aussi choisir un conteneur de type list ou map.

On n'apporte aucune modification à la classe Ligne existante. On va donc maintenant déclarer (partiellement) la classe *Commande* :

```
#ifndef COMMANDE_H
#define COMMANDE_H
#include <vector>
using namespace std;

#include "ligne.h"  //ici il faut un accès à la déclaration complète de la
classe Ligne (3)
class Commande
{
    private:
        vector<Ligne> liste;    //la composition 1..*
    public:
        Commande();
        Ligne getLigne(int numero) const;
        void setLigne(const Ligne &l);
};
#endif    //COMMANDE_H
```

(3) est obligatoire car le compilateur a besoin de connaître parfaitement la classe *Ligne*

```
#include <iostream>
#include <iomanip>
#include "commande.h"
using namespace std;

Commande::Commande()
{
}

Ligne Commande::getLigne(int numero) const
{
    return liste[numero]; // une vérification serait nécessaire    !
}

void Commande::setLigne(const Ligne &l)
{
    liste.push_back(l);
}
```

Voici un exemple d'utilisation de ces deux classes :

```
#include <iostream>
#include "ligne.h"
#include "article.h"
#include "commande.h"

using namespace std;

int main()
{
    Article a1;
    Article gratuit;
    Ligne l1;
    Ligne cadeau;
    Commande c;

    l1.setArticle(&a1);
    cadeau.setArticle(&gratuit);
    c.setLigne(l1);
    c.setLigne(cadeau);

    return 0;
}
```