

# TP POO C++ : Les relations d'association

© 2013-2017 tv <tvaira@free.fr> - v.1.1

<b>Notions de relations</b>	<b>2</b>
<b>Travail demandé</b>	<b>2</b>
Présentation des classes . . . . .	3
Itération 1 : la relation d'agrégation entre les classes Ligne et Article . . . . .	4
Itération 2 : la relation de composition entre les classes Commande et Ligne . . . . .	11
Itération 3 : la relation d'association entre les classes Client et Commande . . . . .	15
Itération finale : saisie et affichage d'une commande . . . . .	18

# TP POO C++ : Les relations d'association

Les objectifs de ce TP sont de découvrir la programmation orientée objet et les relations d'association entre classes en C++.

## Notions de relations

Étant donné qu'en POO les objets logiciels interagissent entre eux, il y a donc des **relations** entre les classes.

On distingue trois différents types de **relations** entre les classes :

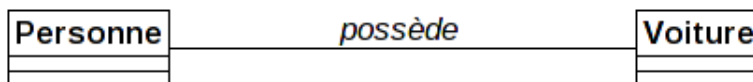
- l'**association** (trait plein avec ou sans flèche)
- la dépendance (flèche pointillée)
- la relation de généralisation ou d'héritage (flèche fermée vide)



Les relations de dépendance et de généralisation ne sont pas traitées dans ce TP.

Une association représente une relation sémantique durable entre deux classes.

*Exemple* : Une personne peut posséder des voitures. La relation possède est une association entre les classes **Personne** et **Voiture**.



Il existe aussi deux cas particuliers d'**association** que nous allons découvrir :

- l'**agrégation** (trait plein avec ou sans flèche et un losange vide)
- la **composition** (trait plein avec ou sans flèche et un losange plein)

## Travail demandé

Dans ce TP, nous allons successivement découvrir les relations d'association, d'agrégation et de composition entre classes. Ce TP présente les éléments (briques) logiciels permettant de **traiter des commandes d'articles** (des livres par exemple).

Ces objets logiciels permettront d'**éditer une commande** comme celle-ci :

-----	
Client : VAIRA	Numéro : 1
-----	
Le 10/04/2013,	Ref. : A00001
-----	
Qte	Description Prix uni  Total
-----	
2	.....Le Trone de fer, tome 14  12  24 euros
1	.....A Game of Thrones - Le Trone de fer, tome 1  0  0 euros
-----	
	24 euros
-----	



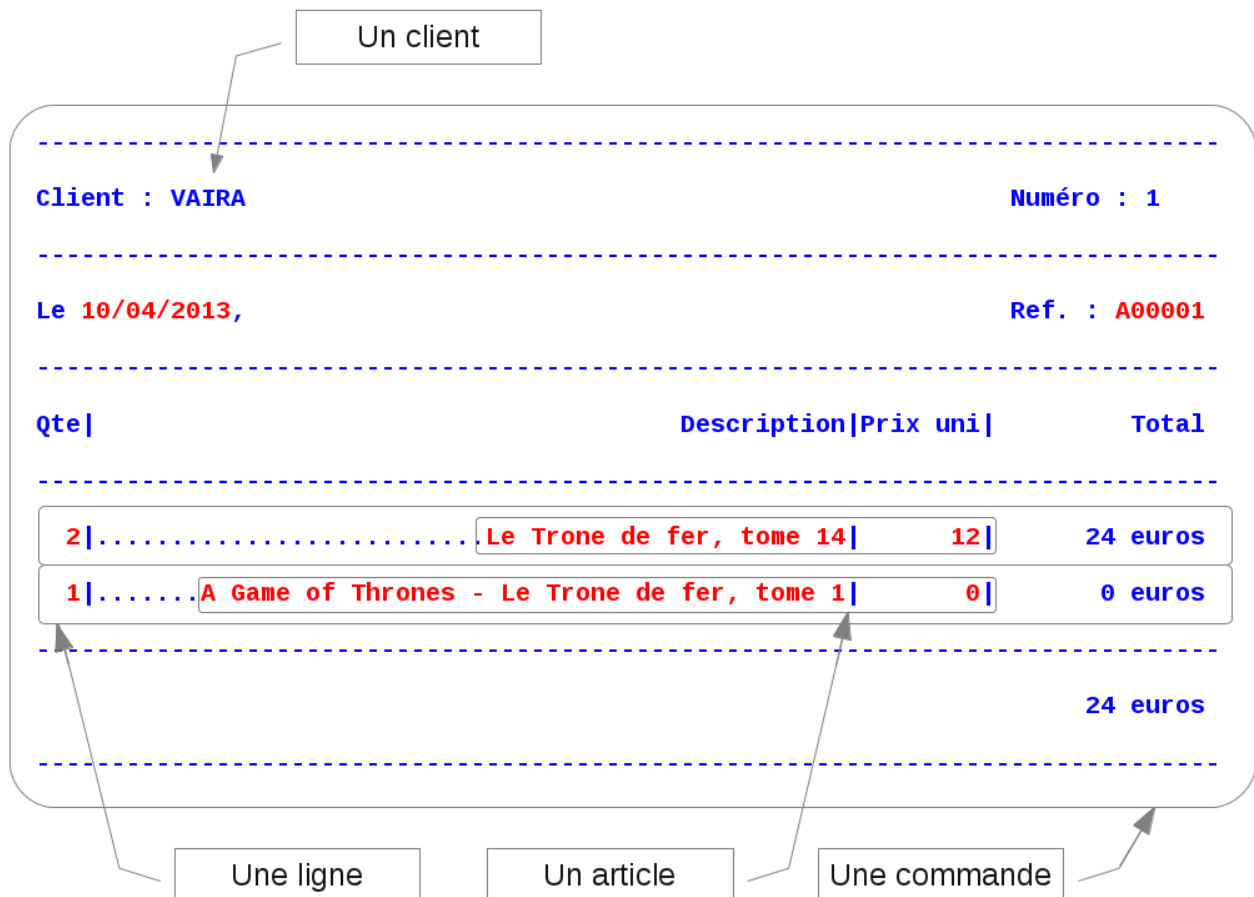
Nous ne gérons pas de base de données pour les articles car ce n'est pas l'objet de ce TP.

## Présentation des classes

On a besoin de modéliser quatre classes :

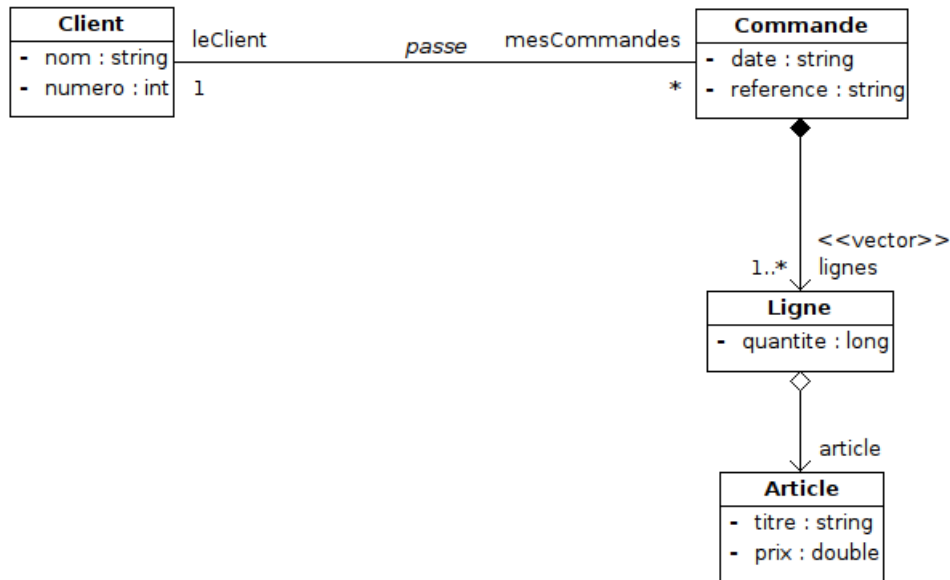
- une classe **Client** qui caractérise une personne qui passe une commande. Un **client** est caractérisé par **son nom** (une chaîne de caractères de type **string**) et **son numéro de client** (un entier de type **int**).
- une classe **Article** décrivant les articles que l'on peut commander. Un **article** est caractérisé par **son titre** (une chaîne de caractères de type **string**) et **son prix** (un réel de type **double**).
- une classe **Commande** qui contient l'ensemble des articles commandés par un client. Une **commande** est caractérisée par **sa référence** (une chaîne de caractères de type **string**) et **sa date** (une chaîne de caractères de type **string**).
- une classe **Ligne** qui correspond la commande d'un article. Une **ligne** d'une commande est caractérisée par **son article** (un objet de type **Article**) et **sa quantité** (un entier de type **long**).

On décompose la commande désirée pour faire apparaître ses composants :



On constate qu'il existe trois relations :

- entre les classes **Client** et **Commande** : Le *Client* passe une *Commande* (**Association**)
- entre les classes **Commande** et **Ligne** : Une *Commande* est composée de *Ligne* (**Composition**)
- entre les classes **Ligne** et **Article** : Une *Ligne* contient un *Article* (**Agrégation**)



Nous allons construire l'application demandée en **trois itérations**.

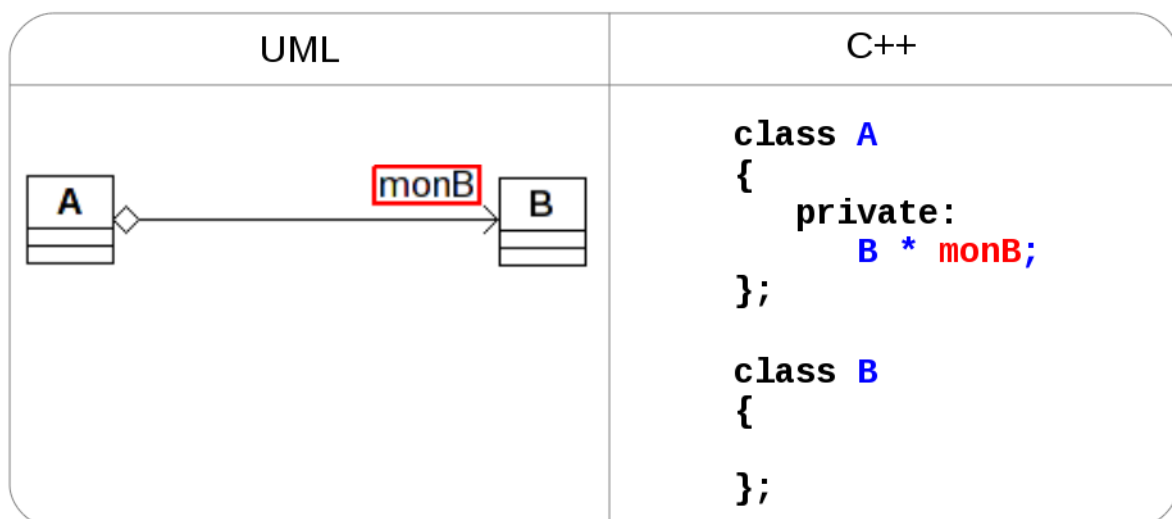


Un développement itératif s'organise en une série de développement très courts de durée fixe nommée itérations. Le résultat de chaque itération est un système partiel exécutable, testé et intégré (mais évidemment incomplet). Chaque itération s'ajoute et enrichit l'existant. Un incrément est donc une avancée dans le développement. On parle de développement itératif et incrémental.


## Itération 1 : la relation d'agrégation entre les classes Ligne et Article

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** » et impliquant :

- qu'une partie peut être partagée avec un autre composite
- que la destruction du composite n'entraînera pas forcément la destruction de toutes ses parties




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** et ce sera un **attribut** de la classe A.

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

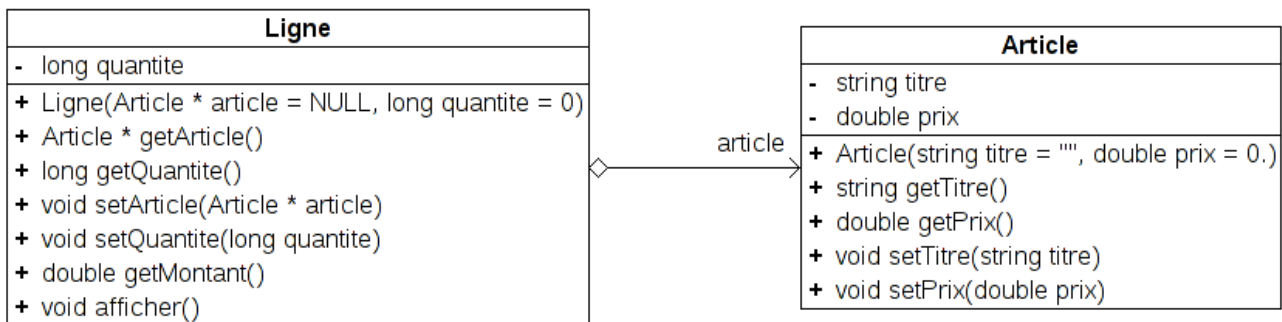
### Quelle relation choisir pour les classes Article et Ligne ?

La relation d'agrégation correspond bien à notre besoin car :

- un article d'une commande peut être partagé avec une autre commande
- quand on devra supprimer une ligne, on ne supprimera pas l'article !

 En effet, lorsqu'on supprime une ligne d'une commande, on ne doit pas supprimer l'article correspondant qui reste commandable par d'autres clients. D'autre part, un même article peut se retrouver dans plusieurs commandes (heureusement pour les ventes!).

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



On va tout d'abord écrire la classe Article :

```

#ifndef ARTICLE_H
#define ARTICLE_H

class Article
{
private:
    string titre;
    double prix;

public:
    Article(string titre="", double prix=0.);
    string getTitre() const;
    double getPrix() const;
    void setTitre(string titre);
    void setPrix(double prix);
};

#endif //ARTICLE_H
    
```

Article.h

**Étape 1.** On doit tout d'abord pouvoir instancier des objets `Article`.

**Question 1.** Compléter la classe `Article` fournie (fichiers `Article.cpp` et `Article.h`). Tester en décommentant les parties de code correspondantes à la question dans le programme fourni `iteration1.cpp`.

```
/* Question 1 */
Article a1; // un article
a1.setTitre("Le Trone de fer, tome 14");
a1.setPrix(12.);
cout << "Titre de l'article : " << a1.getTitre() << endl;
cout << "Prix de l'article : " << a1.getPrix() << endl;

// un autre article
Article a2("A Game of Thrones - Le Trône de fer, tome 1", 13.29);
cout << "Titre de l'article : " << a2.getTitre() << endl;
cout << "Prix de l'article : " << a2.getPrix() << endl;
cout << endl;
```

*Extrait de iteration1.cpp*

Faire :

```
$ make iteration1
```

```
$ ./iteration1
```

Ce qui doit donner :

```
Titre de l'article : Le Trone de fer, tome 14
Prix de l'article : 12
Titre de l'article : A Game of Thrones - Le Trône de fer, tome 1
Prix de l'article : 13.29
```

**Étape 2.** De la même façon, on doit pouvoir instancier des objets `Ligne`.

La déclaration à compléter de la classe `Ligne` est la suivante :

```
#ifndef LIGNE_H
#define LIGNE_H

class Ligne
{
private:
    long quantite;

public:
    Ligne(long quantite=0);

    long getQuantite() const;
    void setQuantite(long quantite);
};

#endif //LIGNE_H
```

*Ligne.h*

La définition à compléter de la classe `Ligne` est la suivante :

```
#include <iostream>
#include "Ligne.h"

using namespace std;

Ligne::Ligne(long quantite/*=0*/) : quantite(quantite)
{
}

// etc ...
```

*Ligne.cpp*

**Question 2.** Compléter la classe `Ligne` (fichiers `Ligne.cpp` et `Ligne.h`). Tester en décommentant les parties de code correspondantes à la question dans le programme fourni `iteration1.cpp`.

```
/* Question 2 */
Ligne l1; // une ligne vide

cout << "Quantité commandée pour cette ligne de commande : " << l1.getQuantite() << endl;
cout << endl;
```

*Extrait de iteration1.cpp*

Faire :

```
$ make iteration1
```

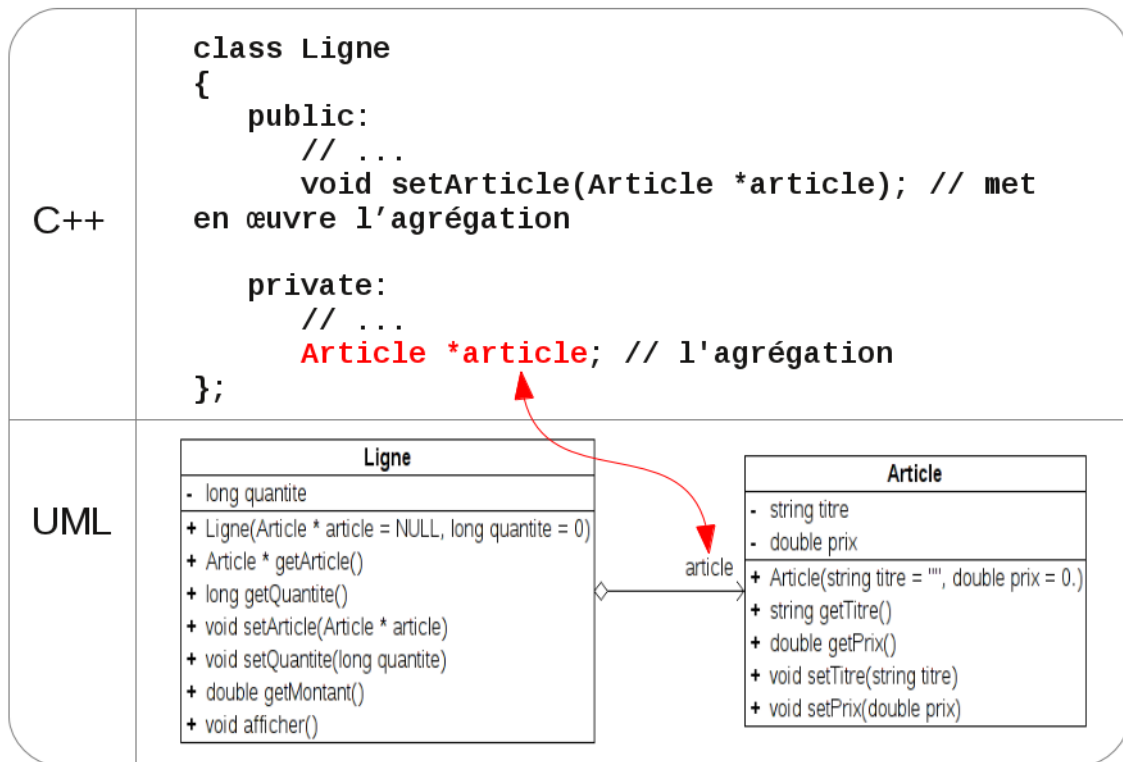
```
$ ./iteration1
```

Ce qui doit donner :

```
Quantité commandée pour cette ligne de commande : 0
```

**Étape 3.** On va maintenant mettre en œuvre la relation d'agrégation entre la classe `Ligne` et la classe `Article`.

Une relation d'agrégation s'implémente généralement par un **pointeur** (pour une relation 1 vers 1) :



Les accesseurs `getArticle()` et `setArticle()` permettent de gérer la relation `article`. Ici, il est aussi possible d'initialiser la relation au moment de l'instanciation de l'objet de type `Ligne` (cf. son constructeur).

On va aussi ajouter une méthode `getMontant()` qui permettra de calculer, à partir de la quantité de l'article commandé, le montant total d'une ligne.

La déclaration 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
    long quantite;

public:
    Ligne(Article *article=NULL, long quantite=0);
    long getQuantite() const;
    void setQuantite(long quantite);
    Article * getArticle() const;
    void setArticle(Article *article);
    double getMontant() const;
};

#endif //LIGNE_H

```

*Ligne.h*





(1) Cette ligne est obligatoire pour indiquer au compilateur que `Article` est de type `class` et permet d'éviter le message d'erreur suivant à la compilation : erreur: 'Article' has not been declared. La classe `Ligne` n'en demande pas plus car elle ne manipule que des pointeurs sur des objets `Article`. Il n'est donc pas nécessaire d'inclure le fichier d'en-tête `<Article.h>`.

La définition à compléter de la classe `Ligne` est la suivante :

```
#include <iostream>

#include "Ligne.h"
#include "Article.h" // accès à la déclaration complète de la classe Article (2)

using namespace std;

Ligne::Ligne(Article *article/*=NULL*/, long quantite/*=0*/) : article(article), quantite(
    quantite)
{
}

// etc ...
```

*Ligne.cpp*



(2) Sans cette ligne, on va obtenir des erreurs à la compilation car celui-ci ne connaît pas "suffisamment" le type `Article` : erreur: invalid use of incomplete type 'struct Article'. Pour corriger ces erreurs, il suffit d'**inclure la déclaration (complète) de la classe Article** qui est contenue dans le **fichier d'en-tête (header) Article.h**

**Question 3.** Compléter la classe `Ligne` (fichiers `Ligne.cpp` et `Ligne.h`) afin d'implémenter l'agrégation `article` et le calcul du montant de la ligne. Tester en décommentant les parties de code correspondantes à la question dans le programme fourni `iteration1.cpp`.

```
/* Question 3 */
Ligne l2; // une autre ligne
l2.setArticle(&a2);
l2.setQuantite(3);
cout << "Quantité commandée pour cette ligne de commande : " << l2.getQuantite() << endl;
cout << "Titre de l'article : " << l2.getArticle()->getTitre() << endl;
cout << "Prix de l'article : " << l2.getArticle()->getPrix() << endl;
cout << "Total pour cette ligne : " << l2.getMontant() << endl;

Article a3("Le Trône de fer, Tome 13 : Le Bûcher d'un roi", 17.96);
Ligne l3(&a3, 0);
cout << "Quantité commandée pour cette ligne de commande : " << l3.getQuantite() << endl;
cout << "Titre de l'article : " << l3.getArticle()->getTitre() << endl;
cout << "Prix de l'article : " << l3.getArticle()->getPrix() << endl;
cout << "Total pour cette ligne : " << l3.getMontant() << endl;
cout << endl;
```

*Extrait de iteration1.cpp*

Faire :

```
$ make iteration1
```

```
$ ./iteration1
```

Vous devez obtenir à l'exécution :

```
Quantité commandée pour cette ligne de commande : 3
Titre de l'article : A Game of Thrones - Le Trone de fer, tome 1
Prix de l'article : 13.29
Total pour cette ligne : 39.87
Quantité commandée pour cette ligne de commande : 0
Titre de l'article : Le Trone de fer, Tome 13 : Le Bucher d'un roi
Prix de l'article : 17.96
Total pour cette ligne : 0
```

**Étape 4.** Pour terminer, on désire qu'une *Ligne* sache s'afficher de manière formatée afin d'établir au final une commande.

Pour cela, on va ajouter une méthode `afficher()`. L'affichage formaté sera réalisé à partir des fonctions `setfill()` et `setw()` de `iomanip`. Ces fonctions réalisent un alignement par défaut à droite (`right`). Il est possible d'obtenir un alignement à gauche en utilisant `left` dans le flux de `cout`. Attention toutefois, l'alignement choisi est conservé pour les affichages suivants.

**Question 4.** Compléter la classe *Ligne* (fichiers `Ligne.cpp` et `Ligne.h`) afin d'assurer un affichage formaté de chaque ligne de la commande (méthode `afficher()`). Tester en décommentant les parties de code correspondantes à la question dans le programme fourni `iteration1.cpp`.

```
/* Question 4 */
// on termine la commande
l1.setArticle(&a1);
l1.setQuantite(2);

// le masque d'affichage d'une commande
cout << setfill(' ') << setw(3) << "Qte";
cout << "|" << setfill(' ') << setw(50) << "Description";
cout << "|" << setfill(' ') << setw(8) << "Prix uni";
cout << "|" << setfill(' ') << setw(15) << "Total\n";
cout << setfill('-') << setw(80) << "\n";

// on affiche les 3 lignes de la commande
l1.afficher(); cout << endl;
l2.afficher(); cout << endl;
l3.afficher(); cout << endl;
```

*Extrait de iteration1.cpp*

Faire :

```
$ make iteration1
```

```
$ ./iteration1
```

Vous devez obtenir cet affichage formaté (que l'on réutilisera pour éditer la commande finale) :

Qté	Description Prix uni	Total
2 .....	Le Trone de fer, tome 14	12  24 euros
3 .....	A Game of Thrones - Le Trone de fer, tome 1	13.29  39.87 euros
0 .....	Le Trone de fer, Tome 13 : Le Bucher d'un roi	17.96  0 euros

**L'itération 1 est terminée** : on est capable d'instancier des objets `Article` et `Ligne` et de les associer. On rappelle qu'une agrégation est une relation d'association particulière. Nous possédons maintenant deux « briques » logicielles : les classes `Article` et `Ligne`.

## Itération 2 : la relation de composition entre les classes Commande et Ligne

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

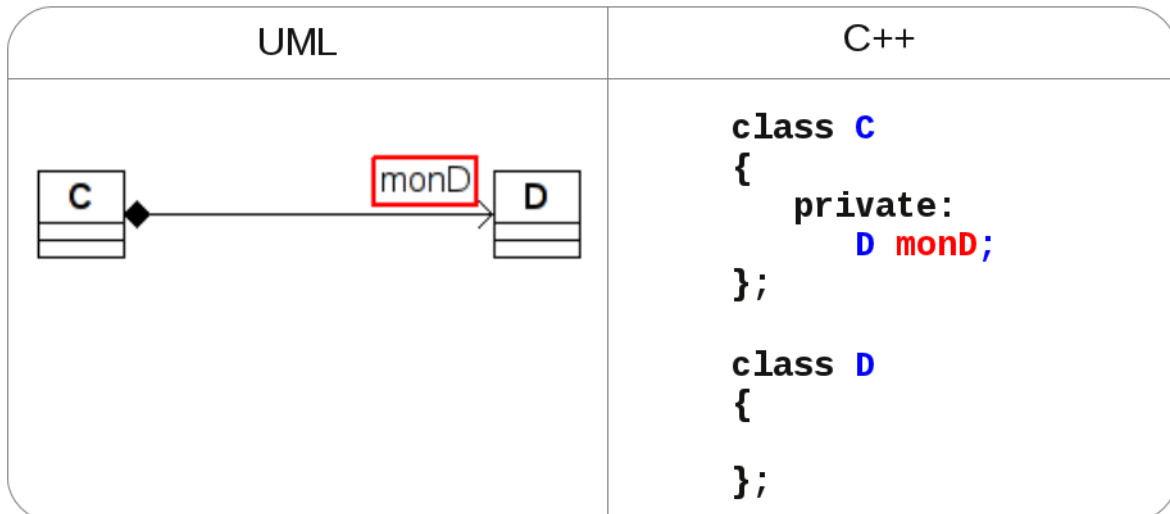
- qu'une partie ne peut appartenir qu'à un seul composite (agrégation non partagée)
- que la destruction du composite entraîne la destruction de toutes ses parties (il est responsable du cycle de vie de ses parties).

### Quelle relation choisir pour les classes `Commande` et `Ligne` ?

La relation de composition correspond bien à notre besoin car :

- une ligne d'une commande ne peut être partagée avec une autre commande : elle est lui est propre
- quand on devra supprimer une commande, on supprimera chaque ligne de celle-ci

La **composition** se représente de la manière suivante en UML :

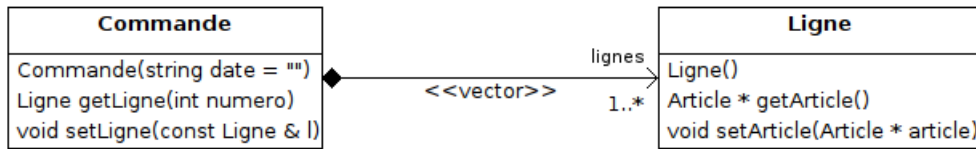


Aux extrémités d'une association, agrégation ou composition, il est possible d'y indiquer une **multiplicité** (ou **cardinalité**) : c'est pour préciser le nombre d'instances (objets) qui participent à la relation. Ici la composition est implicitement de 1 vers 1.



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

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



Dans notre cas, une commande peut contenir une (1) ou plusieurs (\*) lignes. Pour pouvoir conserver plusieurs lignes, on va utiliser un **conteneur** de type **vector** (indiqué dans le diagramme UML ci-dessus par un stéréotype).



Rappel sur la notion de **vector** (cf. [www.cplusplus.com/reference/vector/vector/](http://www.cplusplus.com/reference/vector/vector/)) : Un *vector* est un **tableau dynamique** où il est particulièrement aisé d'accéder directement aux divers éléments par un **index**, et d'en ajouter ou en retirer à la fin. A la manière des tableaux de type C, l'espace mémoire alloué pour un objet de type *vector* est toujours continu, ce qui permet des algorithmes rapides d'accès aux divers éléments.

On n'apporte aucune modification à la classe *Ligne* existante.

**Étape 5.** Au départ, on doit déjà pouvoir instancier des objets *Commande*.

On va donc déclarer la classe *Commande* :

```

#ifndef COMMANDE_H
#define COMMANDE_H

#include <iostream>

using namespace std;

class Commande
{
private:
    string reference;
    string date;

public:
    Commande(string reference="", string date="");

    string getReference() const;
    void setReference(string reference);
    string getDate() const;
    void setDate(string date);
};

#endif //COMMANDE_H
  
```

*Commande.h*

La définition à compléter de la classe *Commande* est la suivante :

```

#include "Commande.h"

Commande::Commande(string reference/*=""*/, string date/*=""*/) : reference(reference), date(
    date)
  
```

```
{  
}  
  
// etc ...
```

*Commande.cpp*

**Question 5.** Compléter la classe `Commande` (fichiers `Commande.cpp` et `Commande.h`) permettant d'instancier des objets `Commande`. Tester en décommentant les parties de code correspondantes à la question dans le programme fourni.

```
/* Question 5 */  
cout << "Question 5 : " << endl;  
Commande c1; // une commande  
c1.setReference("A00001");  
c1.setDate("10/04/2013");  
cout << "Référence de la commande : " << c1.getReference() << endl;  
cout << "Date de la commande : " << c1.getDate() << endl;  
  
Commande c2("A00002", "11/04/2013"); // une autre commande  
cout << "Référence de la commande : " << c2.getReference() << endl;  
cout << "Date de la commande : " << c2.getDate() << endl;  
cout << endl;
```

*Extrait de iteration2.cpp*

Faire :

```
$ make iteration2
```

```
$ ./iteration2
```

Ce qui doit donner :

```
Référence de la commande : A00001  
Date de la commande : 10/04/2013  
Référence de la commande : A00002  
Date de la commande : 11/04/2013
```

**Étape 6.** On va mettre en œuvre la relation de composition entre les classes `Ligne` et `Commande`.

Il faut ici ajouter un conteneur `vector` de `Ligne` pour conserver les lignes de la commande. La composition sera réalisée par la méthode `ajouterLigneArticle` qui créera une nouvelle ligne à la commande. Cette méthode recevra en argument l'`Article` à commander et sa quantité.

La classe `Commande` devra aussi offrir les services suivants :

- une méthode `getTotal()` qui retournera sous la forme d'un `double` le montant total de la commande
- une méthode `getNbLignes()` qui retournera le nombre de lignes de la commande
- une méthode `getNbArticles()` qui retournera la quantité totale d'articles commandés

On va donc déclarer la classe `Commande` :

```
#ifndef COMMANDE_H  
#define COMMANDE_H
```

```

#include <iostream>
#include <vector>
#include "Ligne.h" // ici il faut un accès à la déclaration complète de la classe Ligne (3)

using namespace std;

class Commande
{
private:
    string reference;
    string date;
    vector<Ligne> lignes; // composition

public:
    Commande(string reference="", string date="");
    string getReference() const;
    void setReference(string reference);
    string getDate() const;
    void setDate(string date);
    void ajouterLigneArticle(Article *article, long quantite=1);
    double getTotal() const;
    int getNbLignes() const;
    long getNbArticles() const;
};

#endif //COMMANDE_H

```

*Commande.h*



(3) Cette ligne est obligatoire ici car le compilateur a besoin de "connaître complètement" le type Ligne car des objets de ce type vont devoir être construits.

**Question 6.** Compléter la classe `Commande` (fichiers `Commande.cpp` et `Commande.h`) afin d'éditer une commande de plusieurs articles. Tester en décommentant les parties de code correspondantes à la question dans le programme fourni.

```

/* Question 6 */
cout << "Question 6 : " << endl;
// des articles
Article gratuit("A Game of Thrones - Le Trone de fer, tome 1");
Article a2("A Game of Thrones - Le Trone de fer, tome 2", 13.29);
Article a3("Le Trone de fer, Tome 13 : Le Bucher d'un roi", 17.96);
Commande c3("A00003", "10/04/2013");

c3.ajouterLigneArticle(&a2, 3);
c3.ajouterLigneArticle(&a3, 2);
c3.ajouterLigneArticle(&gratuit, 1);

cout << "Nombre d'articles commandés pour cette commande : " << c3.getNbLignes() << endl;
cout << "Quantité d'articles commandés pour cette commande : " << c3.getNbArticles() << endl;
cout << "Montant de la commande : " << c3.getTotal() << endl;
cout << endl;

```

*Extrait de iteration2.cpp*

Faire :

```
$ make iteration2
```

```
$ ./iteration2
```

Vous devez obtenir à l'exécution :

Question 6 :

Nombre d'articles commandés pour cette commande : 3

Quantité d'articles commandés pour cette commande : 6

Montant de la commande : 75.79

**Étape 7.** Pour terminer, on désire qu'une `Commande` sache s'afficher de manière formatée afin d'établir au final une commande pour un client.

**Question 7.** Compléter la classe `Commande` (fichiers `Commande.cpp` et `Commande.h`) afin d'assurer un affichage formaté de la commande en ajoutant une méthode `afficher()`. Tester en décommentant les parties de code correspondantes à la question dans le programme fourni `iteration2.cpp`.

```
/* Question 7 */
cout << "Question 7 : " << endl;

c3.afficher();
cout << endl;
```

*Extrait de iteration2.cpp*

Faire :

```
$ make iteration2
```

```
$ ./iteration2
```

Vous devez obtenir cet affichage formaté (que l'on réutilisera pour éditer la commande finale) :

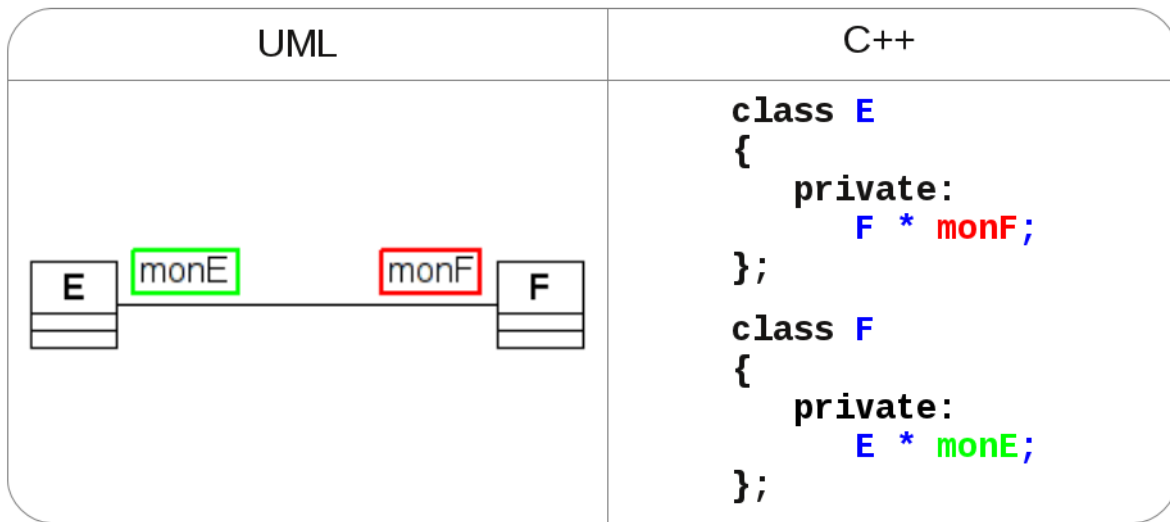
```
-----
Qtel|                Description|Prix uni|      Total
-----
 3|.....A Game of Thrones - Le Trone de fer, tome 2|   13.29|  39.87 euros
 2|.....Le Trone de fer, Tome 13 : Le Bucher d'un roi|   17.96|  35.92 euros
 1|.....A Game of Thrones - Le Trone de fer, tome 1|      0|      0 euros
-----
Le 10/04/2013,                75.79 euros
-----
```


**L'itération 2 est terminée** : on est maintenant capable d'instancier des objets `Article`, `Ligne` et `Commande` et de les faire interagir. Nous possédons maintenant une « brique » logicielle de plus : la classe `Commande`.

## Itération 3 : la relation d'association entre les classes Client et Commande

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 :

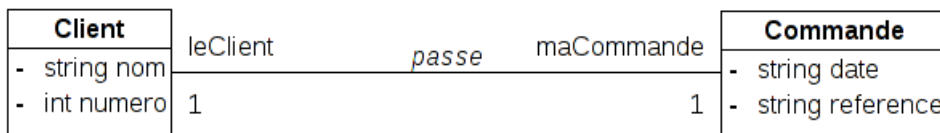


 Ici, la relation est bidirectionnelle (pas de flèche), on a une navigabilité dans les deux sens. Ici, A « connaît » B et B « connaît » A. On peut remarquer que l'association se code de la même manière qu'une agrégation.

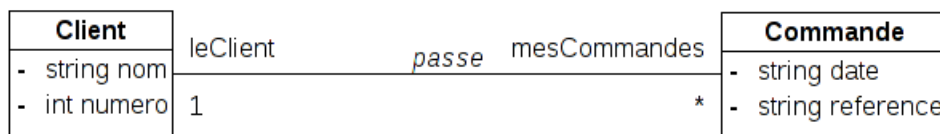
**Quelle relation choisir pour les classes Client et Commande ?**


Les relations d'agrégation et de composition signifient « contient » ou « est composé de » ce qui n'est pas le cas entre un client et une commande. On cherche à exprimer : un client « passe » une commande. C'est donc une simple relation d'association entre classes.

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



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



 Pour la suite du TP, vous pouvez implémenter simplement la relation 1 vers 1. Sinon, il vous faudra mettre en œuvre des conteneurs.

**Étape 8.** On doit tout d'abord pouvoir instancier des objets Client.

**Question 8.** Coder la classe Client (fichiers Client.cpp et Client.h). Tester en décommentant les parties de code correspondantes à la question dans le programme fourni iteration1.cpp.

```

/* Question 8 */
cout << "Question 8 : " << endl;
Client unClient;
                    
```



```
unClient.setNom("VAIRA");
unClient.setNumero(1);
cout << "Nom du client : " << unClient.getNom() << endl;
cout << "Référence du client : " << unClient.getNumero() << endl;

Client unBonClient("DURAND", 2);
cout << "Nom du client : " << unBonClient.getNom() << endl;
cout << "Référence du client : " << unBonClient.getNumero() << endl;
cout << endl;
```

*Extrait de iteration1.cpp*

Faire :

```
$ make iteration3
```

```
$ ./iteration3
```

Ce qui doit donner :

Question 8 :

Nom du client : VAIRA

Référence du client : 1

Nom du client : DURAND

Référence du client : 2

**Étape 9.** On va maintenant mettre en œuvre la relation d'association entre la classe `Client` et la classe `Commande`.

Une relation d'association (comme l'agrégation) s'implémente généralement par un **pointeur** (pour une relation 1 vers 1) et avec un conteneur pour des relations plusieurs (\*).

Attention, la relation est **bidirectionnelle**. La classe `Commande` va donc être modifiée pour intégrer l'association vers `Client`.

Pour améliorer la lisibilité, il est possible de nommer la méthode qui met en œuvre l'association `passerUneCommande()` au lieu de `setCommande()`. De l'autre côté, on peut opter pour `estPasseeParUnClient()` au lieu de `setClient()`.

**Question 9.** Modifier la classe `Commande` (fichiers `Commande.cpp` et `Commande.h`) pour y intégrer la relation d'association avec la classe `Client`.

**Question 10.** Compléter la classe `Client` (fichiers `Client.cpp` et `Client.h`) en y intégrant la relation d'association avec la classe `Commande`.

Pour terminer, on désire qu'une commande s'affiche de la manière suivante :

*Rappel de l'objectif :*

```

Commande commande;
// ...
commande.afficher();

```

```

-----
Client : VAIRA                               Numéro : 1
-----
Le 10/04/2013,                               Ref. : A00001
-----
Qte|                Description|Prix uni|        Total
-----
 2|.....Le Trone de fer, tome 14|    12|    24 euros
 1|.....A Game of Thrones - Le Trone de fer, tome 1|    0|    0 euros
-----
                                           24 euros
-----

```

**Question 11.** Compléter le programme `iteration3.cpp` afin de valider l'association entre les classes `Client` et `Commande`. Tester et afficher une commande.

**L'itération 3 est terminée** : on est maintenant capable d'instancier des objets pour l'ensemble des classes `Article`, `Ligne`, `Commande` et `Client`. Nous possédons maintenant une collection de « briques » logicielles permettant d'écrire un programme orienté objet.

## Itération finale : saisie et affichage d'une commande

**Question 12.** Finaliser un programme `commande.cpp` afin d'intégrer la saisie et l'affichage d'une commande. Tester.

```

#include <iostream>
#include <iomanip>

using namespace std;

#include "Client.h"
#include "Commande.h"
#include "Ligne.h"
#include "Article.h"

int main()
{
    /* Question 12 */
    cout << "Question 12 : " << endl;

    Commande uneCommande;
    Client leClient;

    leClient.passeUneCommande(&uneCommande);

    uneCommande.saisir();
}

```

```

    uneCommande.afficher();

    cout << endl;

    return 0;
}

```

*Un programme qui permet d'éditer une commande*

Faire :

```
$ make commande
```

```
$ ./commande
```

Vous devez obtenir à l'exécution :

```

Question 10 : (bonus)
Nom client ? VAIRA
Numéro client ? 1
Titre article ? A Game of Thrones - Le Trone de fer, tome 2
Prix article ? 13.29
Quantité article ? 3
Un autre article à commander ? (o/n) o
Titre article ? Le Trone de fer, Tome 13 : Le Bucher d'un roi
Prix article ? 17.96
Quantité article ? 2
Un autre article à commander ? (o/n) o
Titre article ?
Prix article ? 0
Quantité article ? 1
Un autre article à commander ? (o/n) n
Date commande ? (jj/mm/aaaa) 10/04/2013

```

```
-----
Client : VAIRA
```

```
Numéro : 1
-----
```

Qte	Description	Prix uni	Total
3	A Game of Thrones - Le Trone de fer, tome 2.....	13.29	39.87 euros
2	Le Trone de fer, Tome 13 : Le Bucher d'un roi.....	17.96	35.92 euros
1	A Game of Thrones - Le Trone de fer, tome 1.....	0	0 euros

```
-----
Le 10/04/2013,
```

```
75.79 euros
-----
```