TP POO C++ : les relations d'association

Sommaire

Les objectifs de ce tp sont de découvrir la programmation orientée objet en C++ et les relations entre classes.

Rappels

La **programmation orientée objet** consiste à **définir des objets logiciels et à les faire interagir entre eux**.

Une **classe** représente la **description abstraite d'un ensemble d'objets possédant les mêmes caractéristiques**. On peut parler également de **type**.

Exemples : la classe Voiture, la classe Personne.

Un **objet** est une entité concrète, possédant **une identité et encapsulant un état et un comportement**. Un **objet** est **une instance d'une classe**.

Exemples : Thierry Vaira est un objet instancié de la classe Personne. La voiture de Thierry Vaira est une instance de la classe Voiture.

Un **attribut** représente un **type d'information (une variable)** contenu dans une classe.

Exemples : vitesse courante, cylindrée, numéro d'immatriculation, etc. sont des attributs potentiels d'une classe Voiture.

Une **opération (une méthode)** représente un **élément de comportement (un service)** contenu dans une classe.

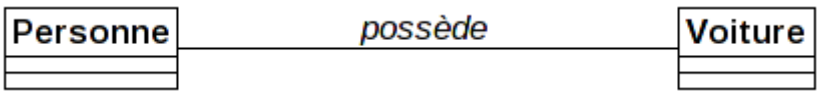
Exemples : demarrer et rouler sont des opérations possibles d'une classe Voiture.

Notion 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)

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`.



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

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 notions suivantes :
– les relations d'association, d'agrégation et de composition entre classes
– les conteneurs

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
--------------------------------------------------------------------------------
```

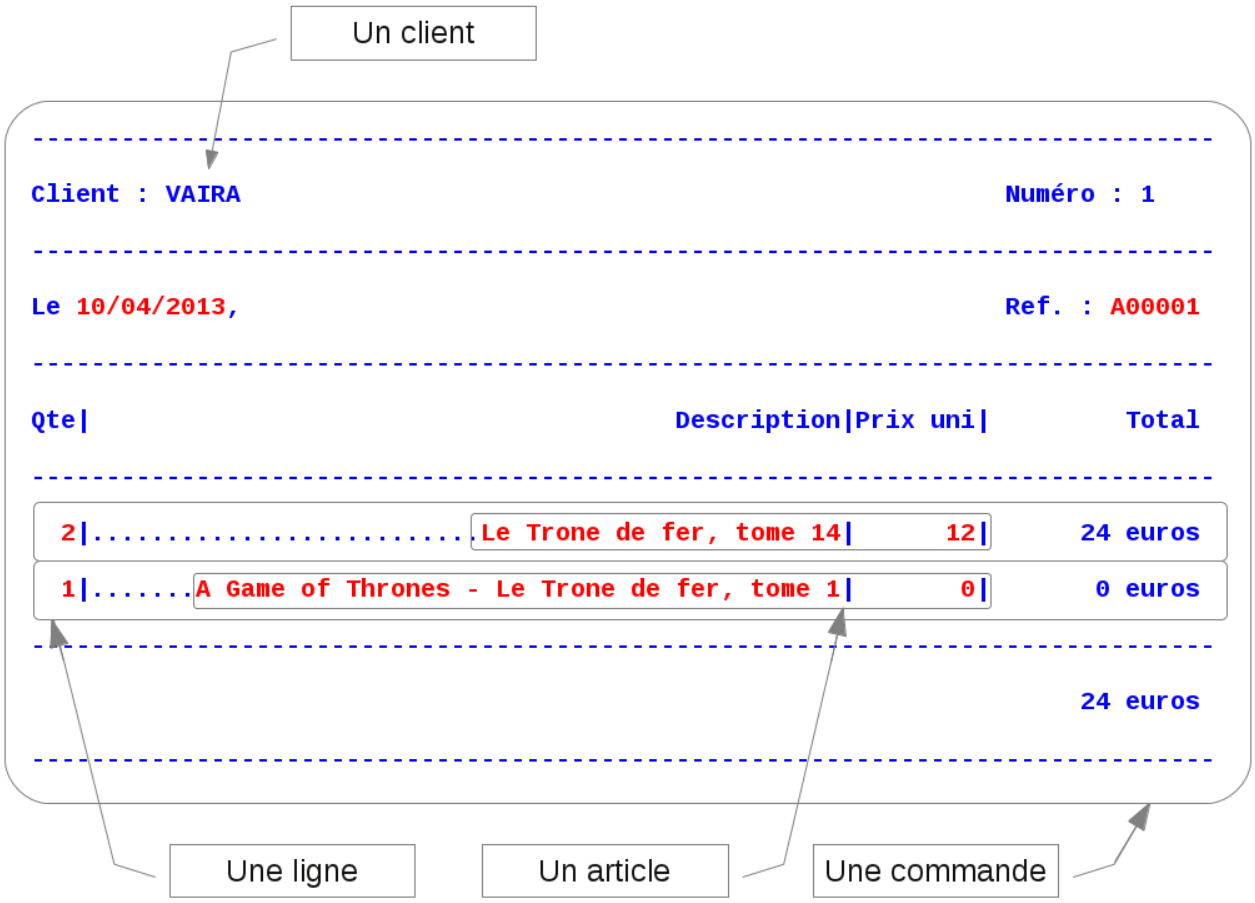
Nous ne gérerons pas de base de données pour les articles car ce n'est pas l'objet de ce TP.

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. 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 à un élément d'un type d'article appartenant à une commande. 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 :



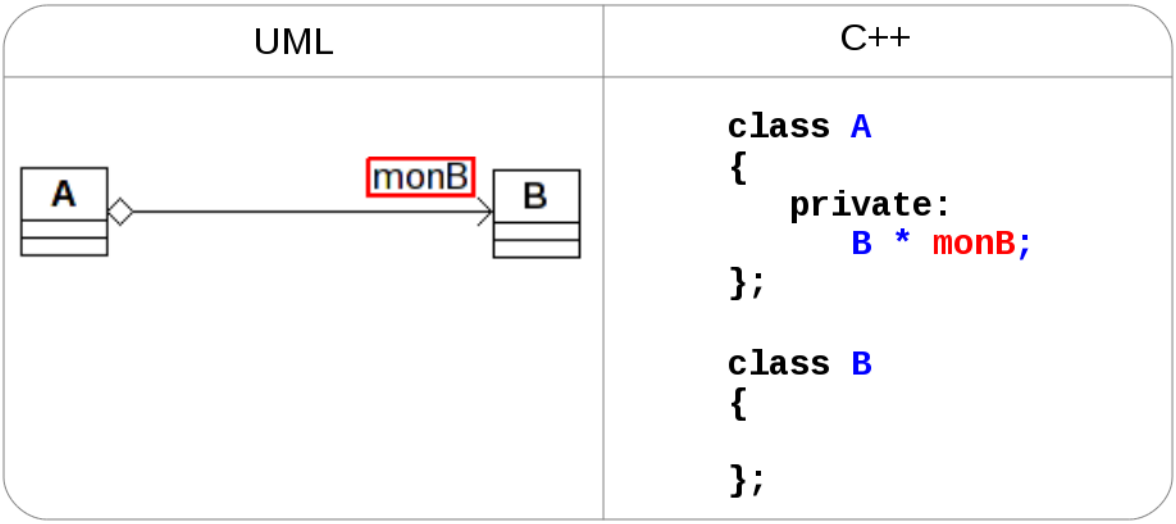
On constate qu'il existe trois relations :
– entre les classes `Client` et `Commande` : Le *Client* passe une *Commande* (**Association**)
– entre les classes `Commande` et `Ligne` : Une *Commande* est composée de *Ligne* (**Composition**)
– entre les classes `Ligne` et `Article` : Une *Ligne* contient un *Article* (**Agrégation**)

Nous allons construire l'application demandée en **trois itérations** (3 séquences).

Développement itératif et incrémental : 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 incomplet). Chaque itération s'ajoute et enrichit l'existant. Un incrément est donc une avancée dans le développement.

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

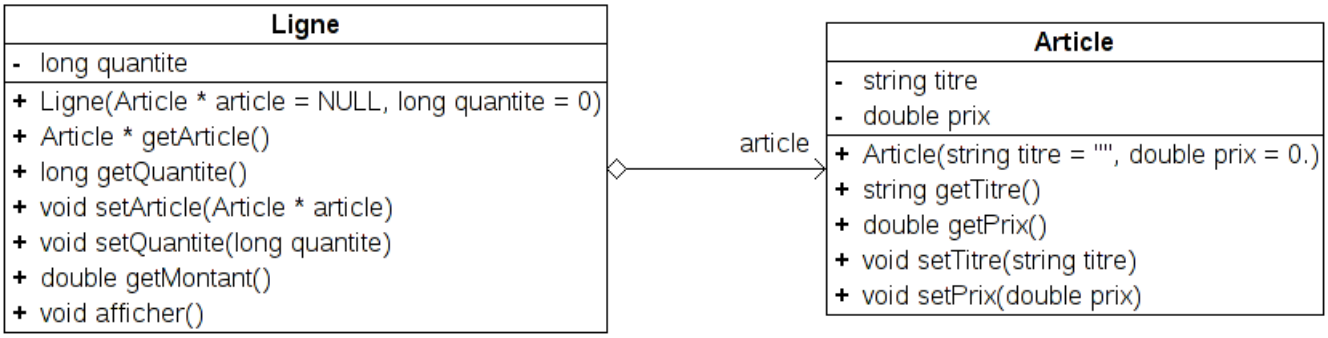
Une **agrégation** est un cas particulier d'association non symétrique exprimant **une relation de contenance**. Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « **contient** » ou « **est composé de** ».



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 est 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 bidirectionnel (pas de flèche) ou unidirectionnel (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` :



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.); //constructeur
     //Accesseurs
     string getTitre() const;
     double getPrix() const;
     void setTitre(string titre);
     void setPrix(double prix);
};

#endif //ARTICLE_H
```

Article.h

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 `testArticle.cpp`.

```
/* Question 1 */
Article a1, a2("A Game of Thrones - Le Trône de fer, tome 1", 13.29), a3("Le Trône de fer,
   Tome 13 : Le Bûcher d'un roi", 17.96);

cout << "Question 1 : " << endl;
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;
cout << "Titre de l'article : " << a2.getTitre() << endl;
cout << "Prix de l'article : " << a2.getPrix() << endl;
cout << endl;
```

Extrait de testArticle.cpp

Ce qui donnera :

```
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
```

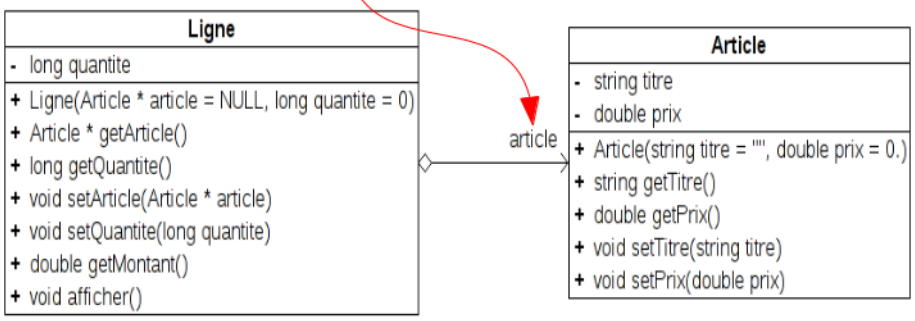

Une relation d'agrégation s'implémente généralement par un **pointeur** (pour une relation 1 vers 1) :

C++

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

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

UML



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).

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;
      long quantite;

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

#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 définition (incomplète) de la classe `Ligne` est la suivante :

```
#include <iostream>
#include <iomanip>

using namespace std;

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

Ligne::Ligne(Article *article/*=NULL*/, long quantite/*=0*/)
{
   this->article = article; // initialise la relation d'agrégation
   this->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 2. Compléter la classe `Ligne` (fichiers `Ligne.cpp` et `Ligne.h`) afin d'implémenter ses accesseurs. Tester en décommentant les parties de code correspondantes à la question dans le programme fourni `testLigne.cpp`.

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

cout << "Question 2 : " << endl;
cout << "Quantité commandée pour cette ligne de commande : " << l1.getQuantite() << endl;
```

Extrait de testLigne.cpp

Ce qui donnera :

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

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

```
/* Question 3 */
Article a1, a2("A Game of Thrones - Le Trone de fer, tome 1", 13.29), a3("Le Trone de fer,
    Tome 13 : Le Bucher d'un roi", 17.96);
a1.setTitre("Le Trone de fer, tome 14");
a1.setPrix(12.);
```



```
Ligne l2, l3(&a3, 0);

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;
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;
```

Extrait de testLigne.cpp

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
```

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 `testLigne.cpp`.

```
/* Question 4 */
l1.setArticle(&a1);
l1.setQuantite(2);

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";

l1.afficher(); cout << endl;
l2.afficher(); cout << endl;
l3.afficher(); cout << endl;
```

Extrait de testLigne.cpp

Vous obtiendrez un affichage formaté réutilisable pour éditer la commande finale :

```
Qte|                                              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'affichage formaté est réalisé à partir des fonctions `setfill()` et `setw()` de `iomanip`. Ces fonctions réalisent un alignement par défaut à droite (`right`), utilisé dans l'exemple ci-dessus. 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.

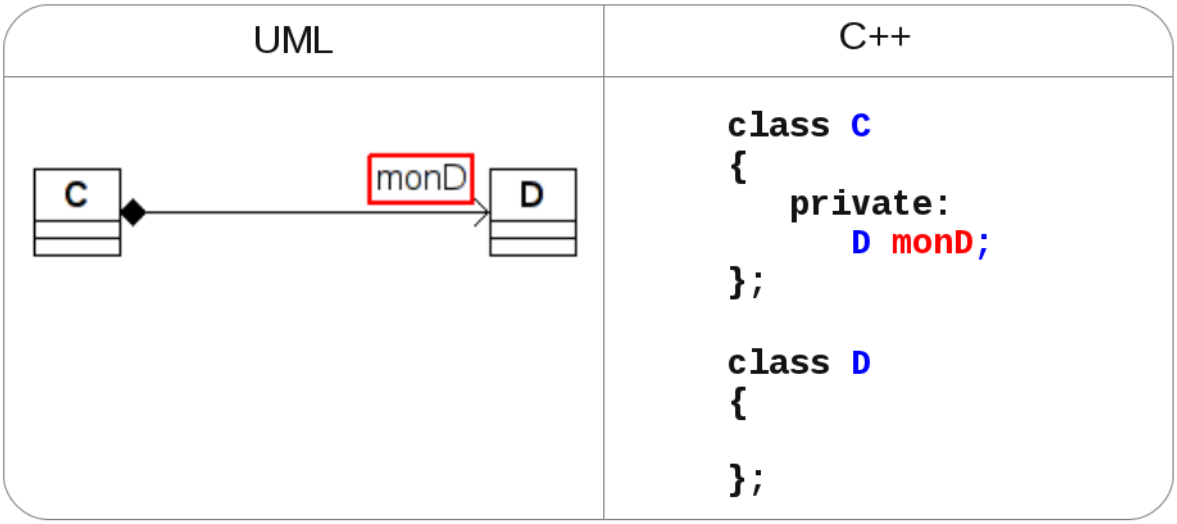
Séquence 2 : la relation de composition entre les classes Commande et Ligne

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

La relation de composition correspond bien à notre besoin car, quand on devra supprimer une commande, on supprimera chaque ligne de celle-ci. D'autre part, une ligne d'une commande ne peut être partagée avec une autre commande : elle est lui est propre.

La relation de composition ne serait pas un bon choix pour la relation entre `Ligne` et `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 !). Donc, l'agrégation choisie à la séquence 1 était bien le bon choix.

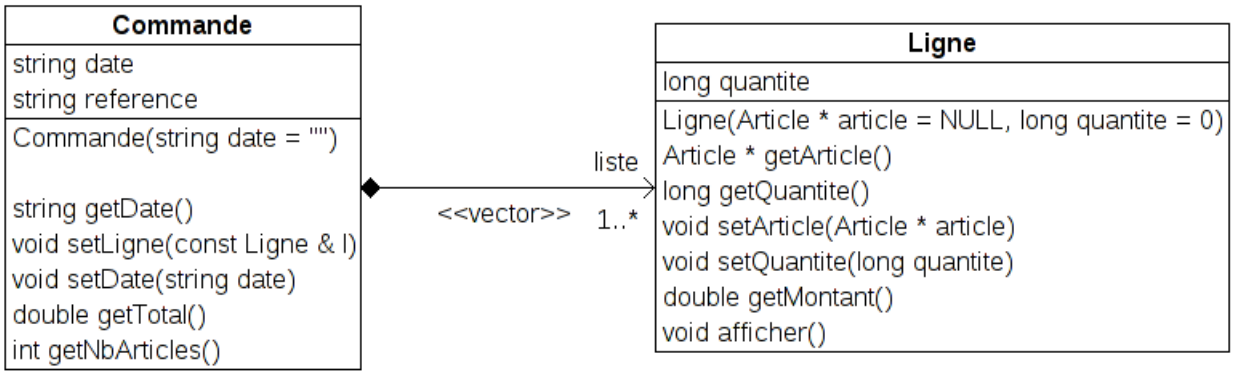
La **composition** se représente de la manière suivante en UML :



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). On aurait pu aussi choisir un conteneur de type `list` ou `map`.

Rappel sur la notion de **vector** (cf. 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. On va donc maintenant déclarer 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:
      string reference;
      string date;
      vector<Ligne> liste; // la composition 1..*

   public:
      Commande(string reference="", string date="");
      string getReference() const;
      void setReference(string reference);
      string getDate() const;
      void setDate(string date);
      void setLigne(const Ligne &l);
      double getTotal();
      int getNbArticles() const;
```

```
    void afficher();
};

#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.

La définition (incomplète) de la classe `Commande` est la suivante :

```
#include <iostream>
#include <iomanip>

using namespace std;

#include "Commande.h"

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

// etc ...
```

Commande.cpp

Question 5. Compléter la classe `Commande` (fichiers `Commande.cpp` et `Commande.h`) en codant l'ensemble des accesseurs `get()` et `set()`. Tester en décommentant les parties de code correspondantes à la question dans le programme fourni.

```
/* Question 5 */
Commande c1, c2("A00002", "11/04/2013");

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

Extrait de testCommande.cpp

Ce qui permettra de valider certains accesseurs :

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


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 */
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), a3("Le Trone de fer, Tome
    13 : Le Bucher d'un roi", 17.96);
Ligne cadeau(&gratuit, 1);
Ligne l2, l3(&a3, 2);
Commande c("A00003", "10/04/2013");

l2.setArticle(&a2);
l2.setQuantite(3);

cout << "Question 6 : " << endl;

c.setLigne(l2);
c.setLigne(l3);
c.setLigne(cadeau);

c.afficher();
```

Extrait de testCommande.cpp

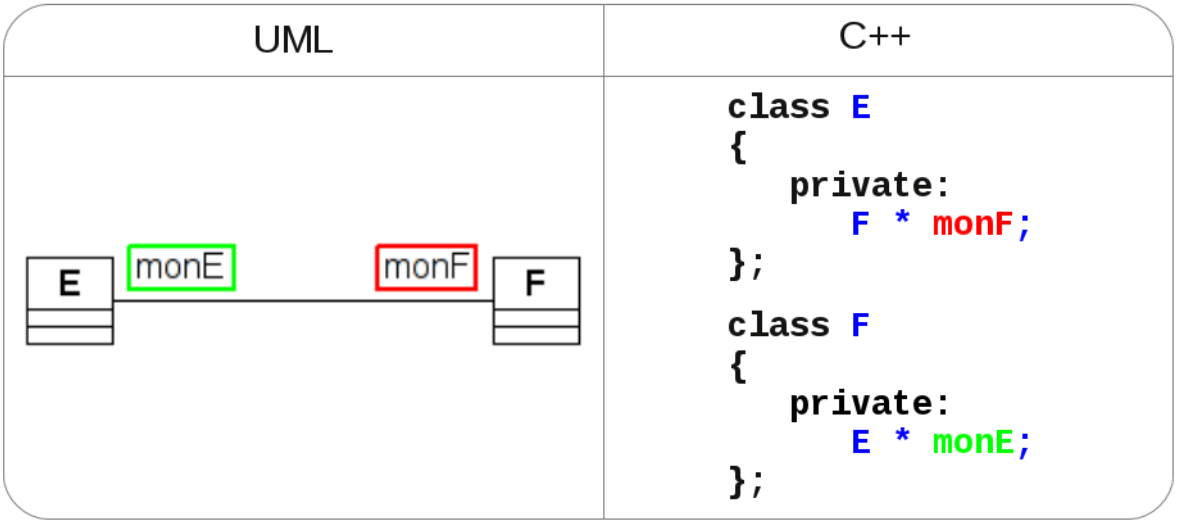
Vous devez obtenir à l'éxécution :

```
--------------------------------------------------------------------------------
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
--------------------------------------------------------------------------------
```


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

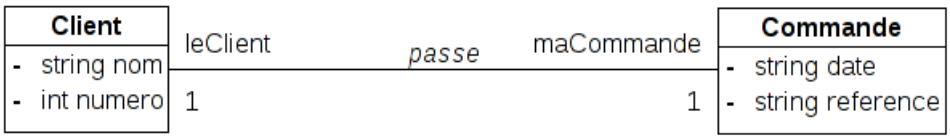
Une **association** représente une **relation sémantique durable entre deux classes**. Les associations peuvent donc être nommées pour donner un sens précis à la relation.

L'**association** se représente de la manière suivante en UML :

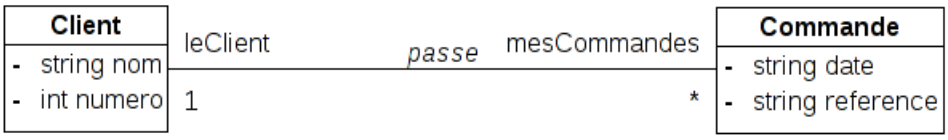


Ici, la relation est bidirectionnelle (pas de flèche), on a une navigabilité dans les deux sens. Ici, A « **connaît** » B et B « **connaît** » A. On peut remarquer que l'association se code de la même manière qu'une agrégation.

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

Question 7. Modifier la classe `Commande` (fichiers `Commande.cpp` et `Commande.h`) pour y intégrer la relation d'association avec la classe `Client`.

Question 8. Écrire la classe `Client` (fichiers `Client.cpp` et `Client.h`) en y intégrant la relation d'association avec la classe `Commande`.

Question 9. Écrire un programme `testClient.cpp` afin de valider les classes `Client` et `Commande`. Tester.

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 10. *Bonus :* Écrire un programme `testSaisie.cpp` afin de valider un programme final (voir ci-dessous) intégrant la saisie et l'affichage d'une commande. Tester.

```
cout << "Question 10 : (bonus) " << endl;

Commande uneCommande;
Client leClient;

leClient.passeUneCommande(&uneCommande);

uneCommande.saisir();

uneCommande.afficher();
```

Extrait de testCommande.cpp

Vous devez obtenir à l'éxécution :

```
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 ? A Game of Thrones - Le Trone de fer, tome 1
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
--------------------------------------------------------------------------------
```

