

# La programmation orientée objet (POO) en C++

## Troisième partie : Les relations entre classes

**Thierry Vaira**

BTS SN Option IR

v.1.2 - 7 décembre 2020

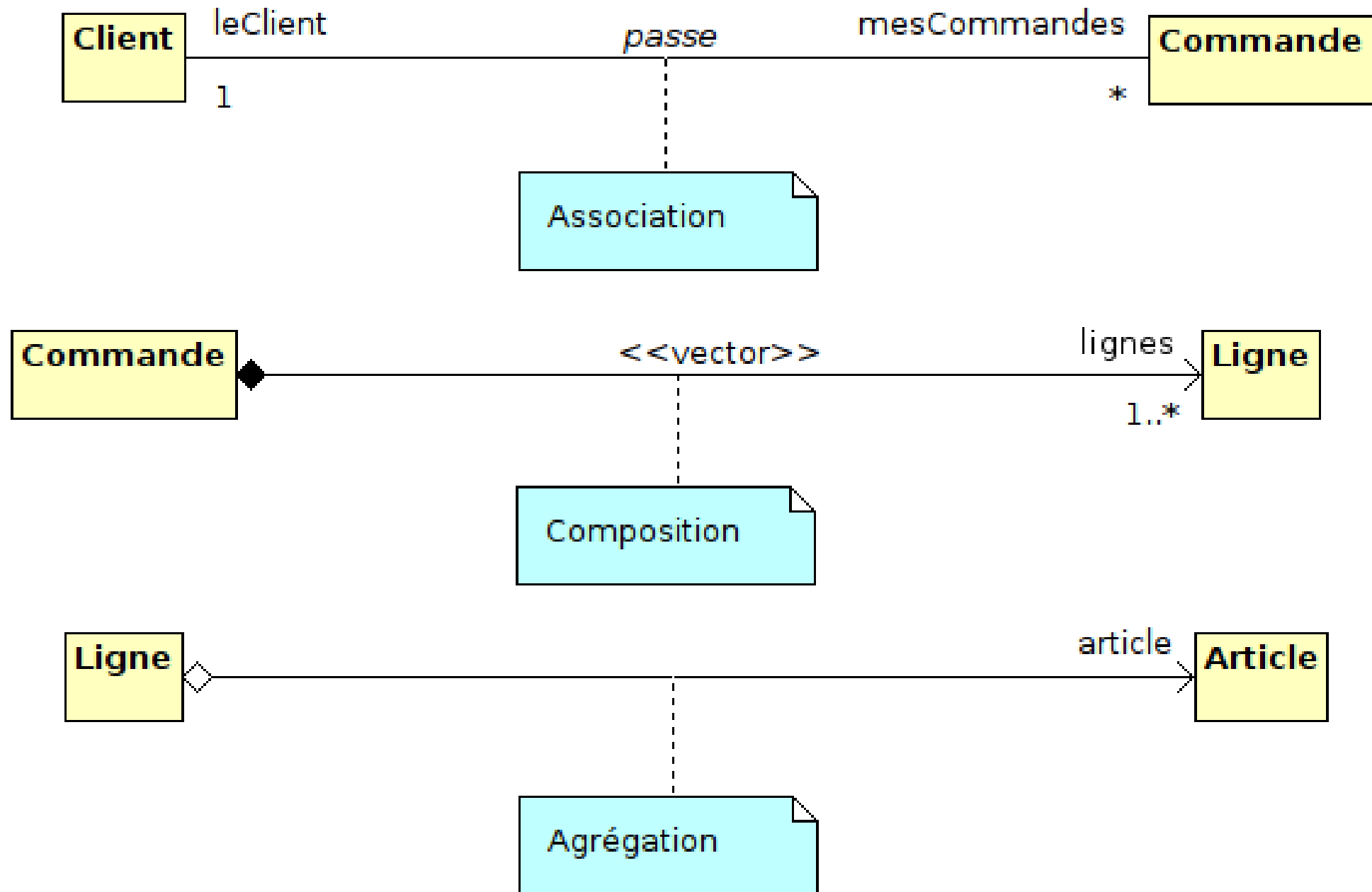


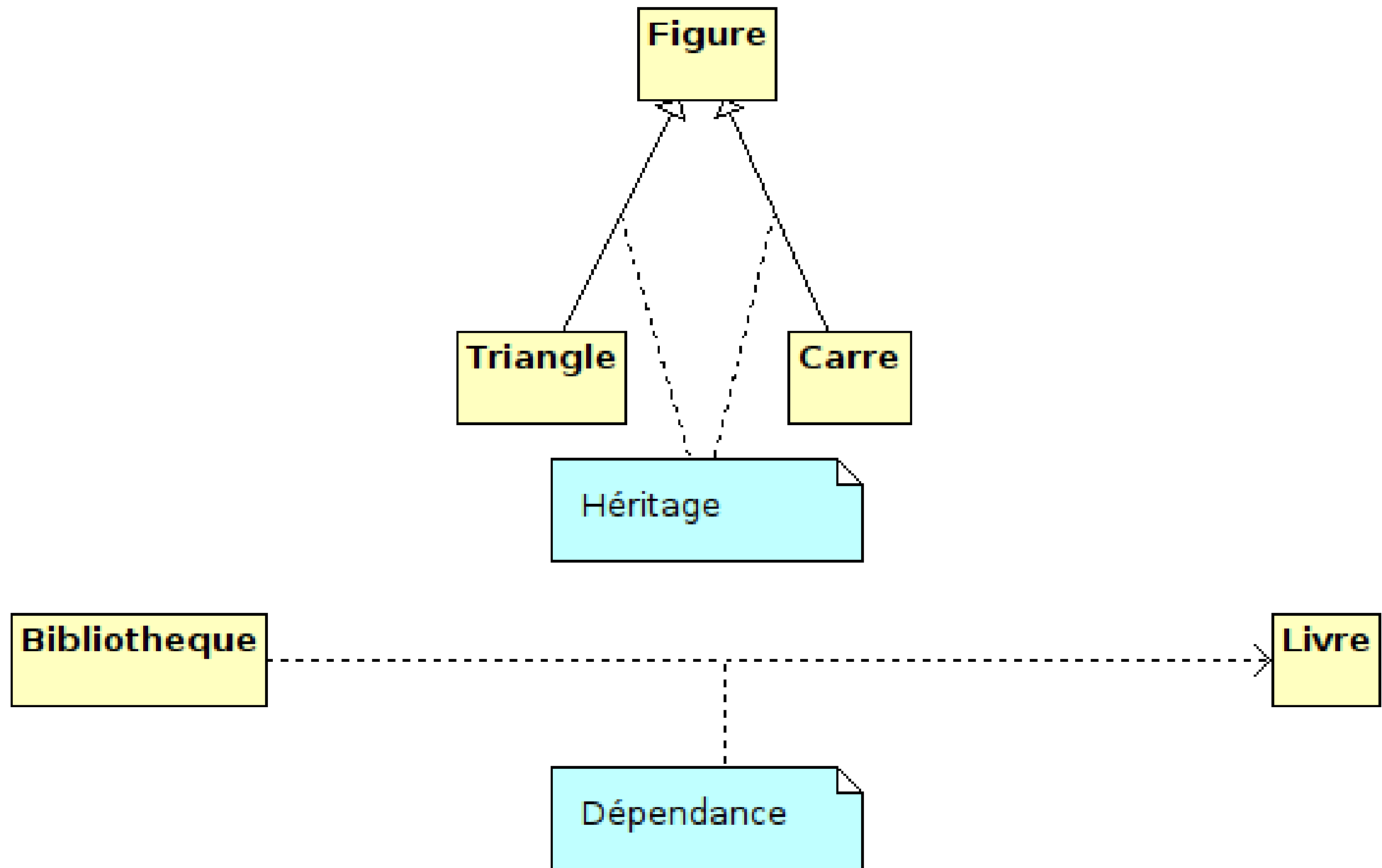
# Sommaire

- 1 Les relations
- 2 L'association
- 3 L'agrégation
- 4 La composition
- 5 Les détails d'une relation d'association
- 6 Les multiplicités
- 7 Classe association
- 8 L'héritage
- 9 La dépendance
- 10 Conclusion

# UML/C++

- Étant donné qu'en POO les **objets logiciels interagissent entre eux**, il y a donc des **relations** entre les classes.
  - On distingue quatre types de relations **durables** :
    - 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 **généralisation** ou l'**héritage** (flèche fermée vide)
  - Il existe une relation **temporaire** :
    - la **dépendance** (flèche pointillée)
- ➡ Remarque : L'**agrégation** et la **composition** sont deux cas particuliers d'**association**.



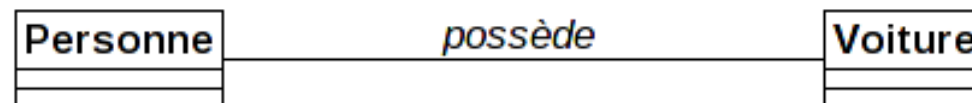


# L'association

- Une **association** représente une **relation durable** entre deux classes.

👉 *Exemple* : Une personne possède une voiture.

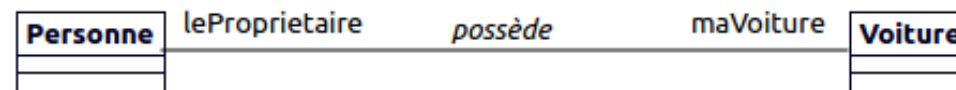
- La relation **possède** est une **association** entre les classes Personne et Voiture.
- Les associations peuvent donc être nommées pour donner un sens précis à la relation.



⇒ Ici, la relation est bidirectionnelle, on a une **navigabilité** dans les deux sens. Ici, Personne « **possède** » Voiture et Voiture « **est possédée** » par Personne.

# L'association en C++

⇒ En C++, l'association s'implémente par un **pointeur**. Les accesseurs *get/set* permettent de mettre en oeuvre la relation.



```
class Personne
{
    private:
        Voiture *maVoiture;

    public:
        Voiture* getVoiture();
        void setVoiture(Voiture* v);
};
```

```
class Voiture
{
    private:
        Personne *leProprietaire;

    public:
        Personne* getPersonne();
        void setPersonne(Personne* p);
};
```

# L'agrégation

- L'**agrégation** est un cas particulier d'association non symétrique exprimant **une relation de contenance**.
- ☞ *Exemple* : Une ligne d'une commande contient l'achat d'un article.
- Les agrégations n'ont pas besoin d'être nommées : implicitement elles signifient « **contient** » ou « **est composé de** ».
- Ici, Ligne est le **composite** et Article le **composant**.



⇒ Dans une agrégation, le composant peut être partagé entre plusieurs composites ce qui entraîne que, lorsque le composite Ligne sera détruit, le composant Article ne le sera pas forcément.



# L'agrégation en C++

⇒ En C++, l'agrégation s'implémente par un **pointeur** (ou par référence).



```
class Ligne
{
    private:
        Article *article;
        long quantite;
    public:
        Article* getArticle();
        void setArticle(Article* a);
        ...
};
```

```
class Article
{
    private:
        string libelle;
        double prix;
    public:
        ...
};
```

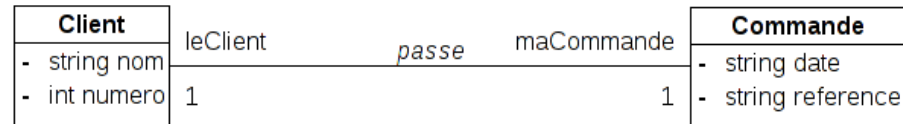
⇒ L'adresse de l'objet Article peut aussi être un argument passé à un constructeur de Ligne : `Ligne(Article* article, long quantite);`

# La liaison en C++ pour les associations et agrégations (1/3)

- ➡ Une association (et une agrégation) représente la capacité d'un objet à envoyer un message (appeler une méthode) à un autre. L'objet émetteur doit donc connaître l'identité du destinataire.
- ➡ L'identité d'un objet est son adresse. Il faut donc conserver **durablement** (en attribut) l'adresse du destinataire. Une association entre deux classes est donc implémentée par un pointeur vers ce type.
- ➡ Il est aussi nécessaire de fournir un moyen d'associer (*bind*) les deux objets pour encapsuler la liaison (c'est-à-dire conserver les adresses des objets impliqués dans la relation).

Techniquement, une référence pourrait être utilisée en remplacement d'un pointeur. Il est préférable d'utiliser un pointeur qui notamment le remplacement au moment de l'exécution. D'autre part, il est déconseillé d'utiliser des pointeurs "intelligents" pour une association (car il est impossible de savoir si l'objet associé a été alloué dynamiquement).

# La liaison en C++ pour les associations et agrégations (2/3)



➡ Il est aussi possible de passer une référence de l'objet à associer.

```

class Client {
private:
    Commande* maCommande; // association
                           vers Commande
public:
    void passeUneCommande(Commande&
                           uneCommande) // par référence
    { this->maCommande = &uneCommande; }
    void passeUneCommande(Commande*
                           uneCommande) // par adresse
    { this->maCommande = uneCommande; }
};

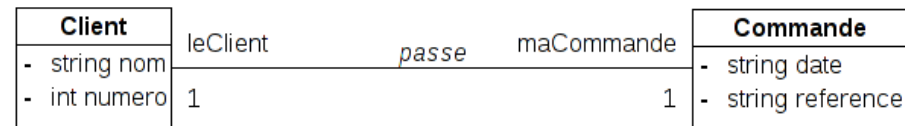
```

```

class Commande {
private:
    Client* leClient; // association
                      vers Client
public:
    void estPasseeparUnClient(Client&
                              unClient) // par référence
    { this->leClient = &unClient; }
    void estPasseeparUnClient(Client*
                              unClient) // par adresse
    { this->leClient = unClient; }
};

```

# La liaison en C++ pour les associations et agrégations (3/3)



➡ Il est possible de confier la responsabilité d'établir la liaison (*bind*) à une fonction (éventuellement une fonction amie si besoin d'accéder aux attributs privés des deux classes) :

```

void etablirRelation(Client &unClient, Commande &uneCommande)
{
    unClient.passeUneCommande(uneCommande); // Client ---> Commande
    uneCommande.estPasseeparUnClient(unClient); // Client <--- Commande
}

// Exemple :
Commande laCommande;
Client leClient;

etablirRelation(leClient, laCommande); // ou : passerUneCommande()
  
```

# La composition

- Une **composition** est une agrégation plus forte signifiant « **est composé d'un** » et impliquant :
  - un composant ne peut appartenir qu'à un seul composite (agrégation non partagée)
  - la destruction du composite entraîne la destruction de tous ses composants (il est responsable du cycle de vie de ses parties).

👉 *Exemple* : Une commande est composée d'une ou plusieurs lignes.



➡ La relation de composition correspond à la situation : quand on devra supprimer une commande, on détruira chaque ligne de celle-ci. D'autre part, une ligne d'une commande ne peut être partagée avec une autre commande : elle lui est propre.

# La composition en C++

⇒ En C++, la composition s'implémente par **valeur**. On peut aussi utiliser une allocation dynamique et donc un **pointeur**.



```

class Commande
{
    private:
        Ligne ligne; // -> 1
        string date;
        string reference;
        ...
};
  
```

```

class Ligne
{
    private:
        Article *article;
        long quantite;
    public:
        ...
};
  
```

⇒ L'objet Ligne doit être initialisé dans la liste d'initialisation du constructeur Commande.

⇒ Pour pouvoir conserver **plusieurs lignes (\*)**, on pourrait utiliser un vector de Ligne : `vector<Ligne> lignes;`

# La composition en C++ (fin)

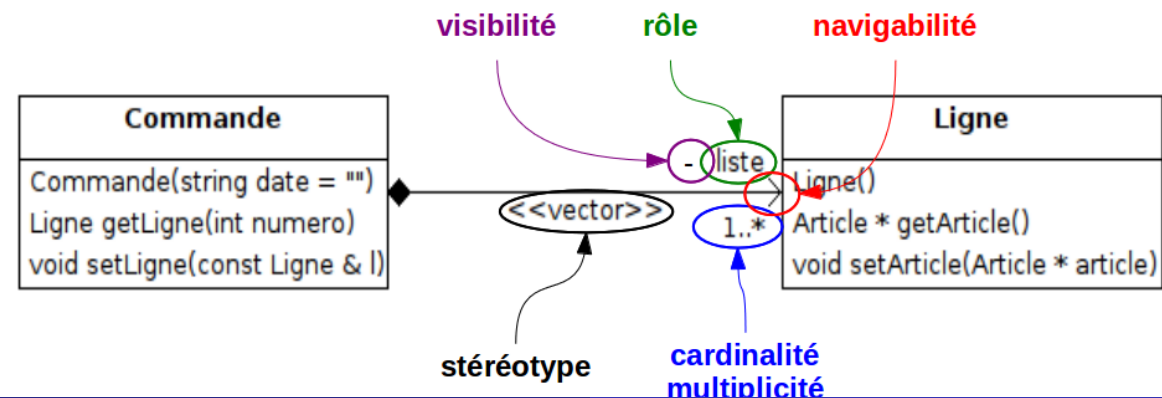
⇒ En C++, la composition pourrait s'implémenter par **pointeur** en s'assurant d'allouer et de libérer l'instance souhaitée.

```
class Commande
{
    private:
        Ligne *ligne; // composition par pointeur
    public:
        Commande() : ligne(NULL) {
            ligne = new Ligne;
        }
        ~Commande() {
            if(ligne != NULL)
                delete ligne;
        }
        Ligne getLigne() const {
            return *ligne; // retourne une copie
        }
};
```

# Détails d'une relation

➡ Aux extrémités d'une relation, il est possible de préciser :

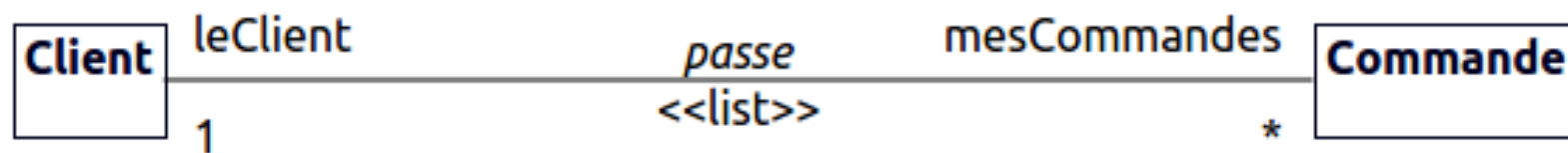
- Un **rôle** : c'est la manière dont les instances d'une classe voient les instances d'une autre classe. Le rôle se traduit par un **attribut** liste dans la classe Commande.
- Une **multiplicité** (ou **cardinalité**) : c'est pour préciser le **nombre d'instances (objets)** qui participent à la relation. Une multiplicité peut s'écrire : n (exactement n, un entier positif), n..m (n à m), n..\* (n ou plus) ou \* (plusieurs).
- Une **navigabilité** : c'est précisé par la présence ou non d'une flèche sur la relation. Ici, Commande « connaît » Ligne mais pas l'inverse. Les relations peuvent être bidirectionnelles (pas de flèche) ou unidirectionnelles (avec une flèche qui précise le sens).
- La **visibilité** du rôle : - pour private, # pour protected et + pour public.





# Les multiplicités plusieurs (\*)

⇒ Pour pouvoir conserver **plusieurs instances** (c'est-à-dire plusieurs objets), on doit utiliser un **conteneur** de type vector, list ou map par exemple. Cela pourra être précisé dans le diagramme UML par un **stéréotype**.

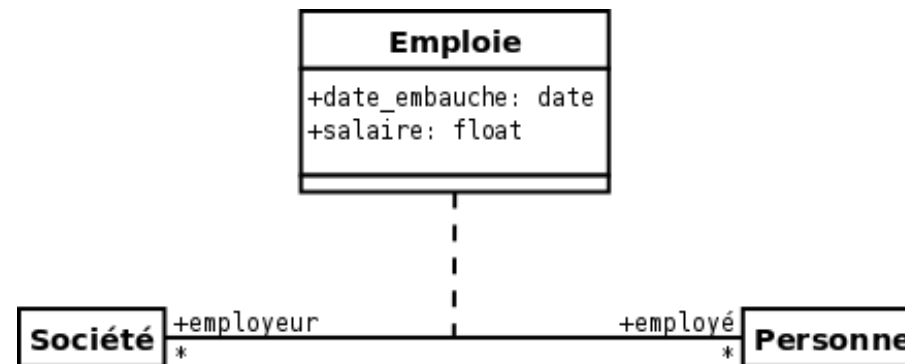


```
class Client
{
    private:
        list<Commande*> mesCommandes; // -> *
        ...
};
```

```
class Commande
{
    private:
        Client *leClient; // -> 1
        ...
};
```

# Classe association

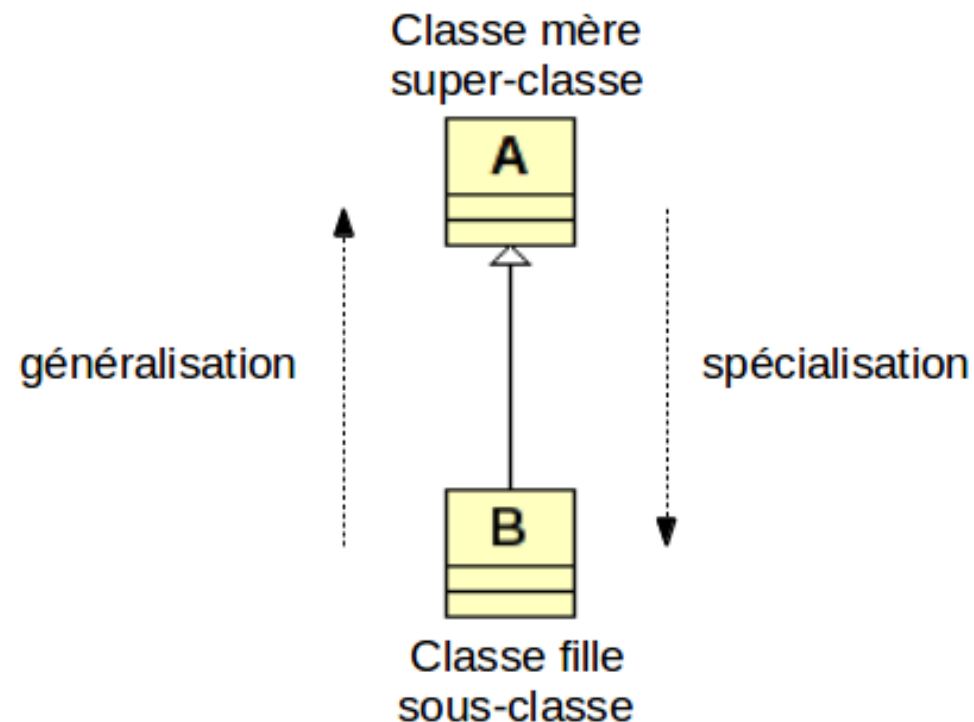
- Parfois, une association doit posséder des propriétés.
- ☞ *Exemple* : l'association **Emploie** entre une société et une personne possède comme propriétés le salaire et la date d'embauche.
- Les associations ne pouvant posséder de propriété, il faut introduire un nouveau concept pour modéliser cette situation : celui de **classe-association**.
- Une classe-association est caractérisée par un trait discontinu entre la classe et l'association qu'elle représente.



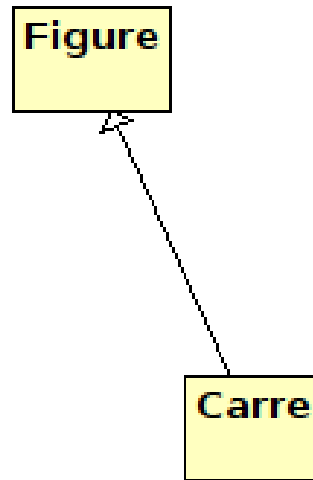
➡ Source : <https://laurent-audibert.developpez.com/Cours-UML/>

# L'héritage

- L'**héritage** permet **d'ajouter des éléments (propriétés et/ou comportement)** à une classe existante pour en obtenir une **nouvelle plus précise**.
- ☞ *Exemple* : Un wagon-bar **est** un wagon avec quelque chose en plus (un bar).



# L'héritage en C++

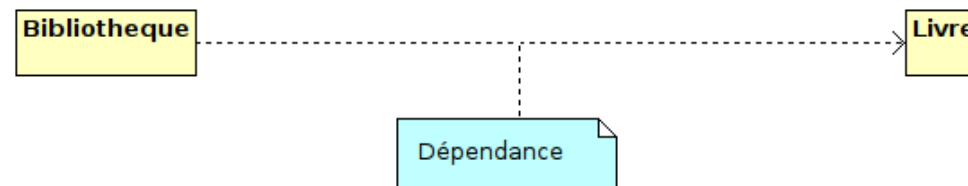


```
class Figure
{
    private:
        double x, y, z;
    public:
        Figure(double x=0, double y=0,
               double z=0) : x(x), y(y), z(
                   z) {}
};
```

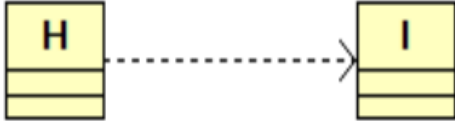
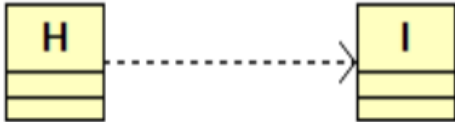
```
class Carre : public Figure
{
    private:
        double largeur;
    public:
        Carre(double largeur, double x=0,
              double y=0) : Figure(x, y,
                  0), largeur(largeur) {}
};
```

# La dépendance

- Lorsqu'un objet « **utilise** » **temporairement** les services d'un autre objet, cette relation d'utilisation est une **dépendance** entre classes.
- Une dépendance s'illustre par une **flèche en pointillée** dans un diagramme de classes en UML.
- 👉 *Exemple* : un objet livre passé en paramètre de la méthode emprunter() pour une bibliothèque. Cela peut être aussi un objet instancié localement dans une méthode.
- Généralement, les dépendances ne sont pas montrées dans un diagramme de classes car elles ne sont qu'une utilisation temporaire donc un détail de l'implémentation que l'on ne considère pas judicieux de mettre en avant.



# La dépendance en C++

UML	C++
 <p>Généralement, les dépendances ne sont pas montrées dans un diagramme de classes.</p>	<pre> class H {     void faireQuelqueChose()     {         <b>I</b> i;         // ...     } };  class I { }; </pre> <p><i>Objet temporaire car il n'existe que pendant la durée de l'exécution de la méthode.</i></p>
 <p>Généralement, les dépendances ne sont pas montrées dans un diagramme de classes.</p>	<pre> class H {     void faire(<b>I</b> i)     {         // ...     } };  class I { }; </pre> <p><i>Objet temporaire car il n'existe que pendant la durée de l'exécution de la méthode.</i></p>

# Quelle relation choisir ?

⇒ On tient compte que :

- les relations d'**association**, d'**agrégation** et de **composition** illustrent une relation de type « **avoir** » : X « a » Y
- la relation d'**héritage** illustre une relation de type « **être** » : X « est » Y

⇒ En effet :

- une personne a une voiture et non ~~une personne est une voiture~~ donc l'**association**
- un chat est un animal et non ~~un chat a un animal~~ donc l'**héritage**.