La programmation orientée objet (POO) en C++
Quatrième partie : Les relations entre classes

Thierry Vaira

BTS SN Option IR

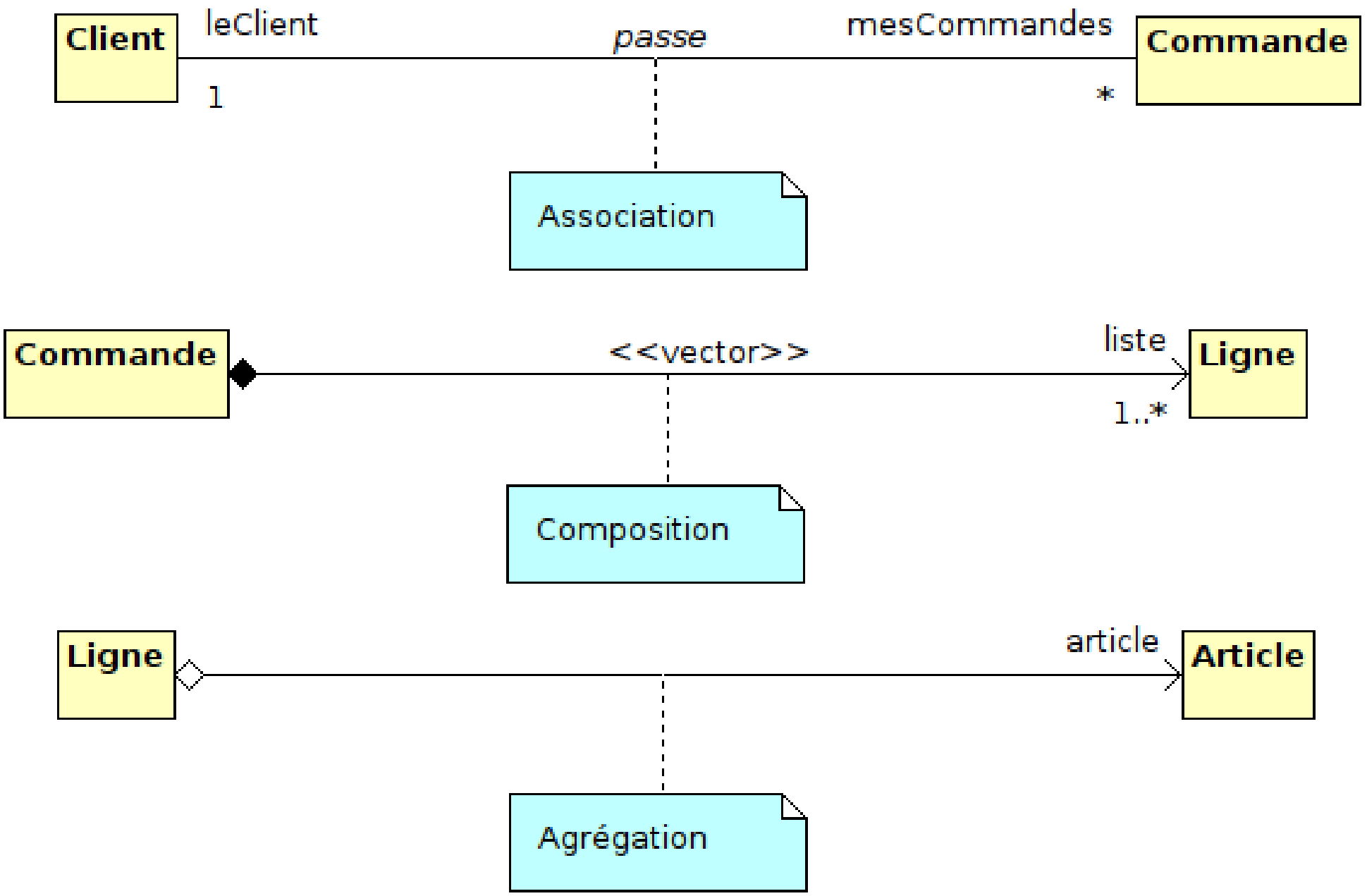
v.1.0 - 11 octobre 2017

Sommaire

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**.


Client
leClient
1
passe
mesCommandes
*
Commande
Association
Commande
<<vector>>
liste
1..*
Ligne
Composition
Ligne
article
Article
Agrégation

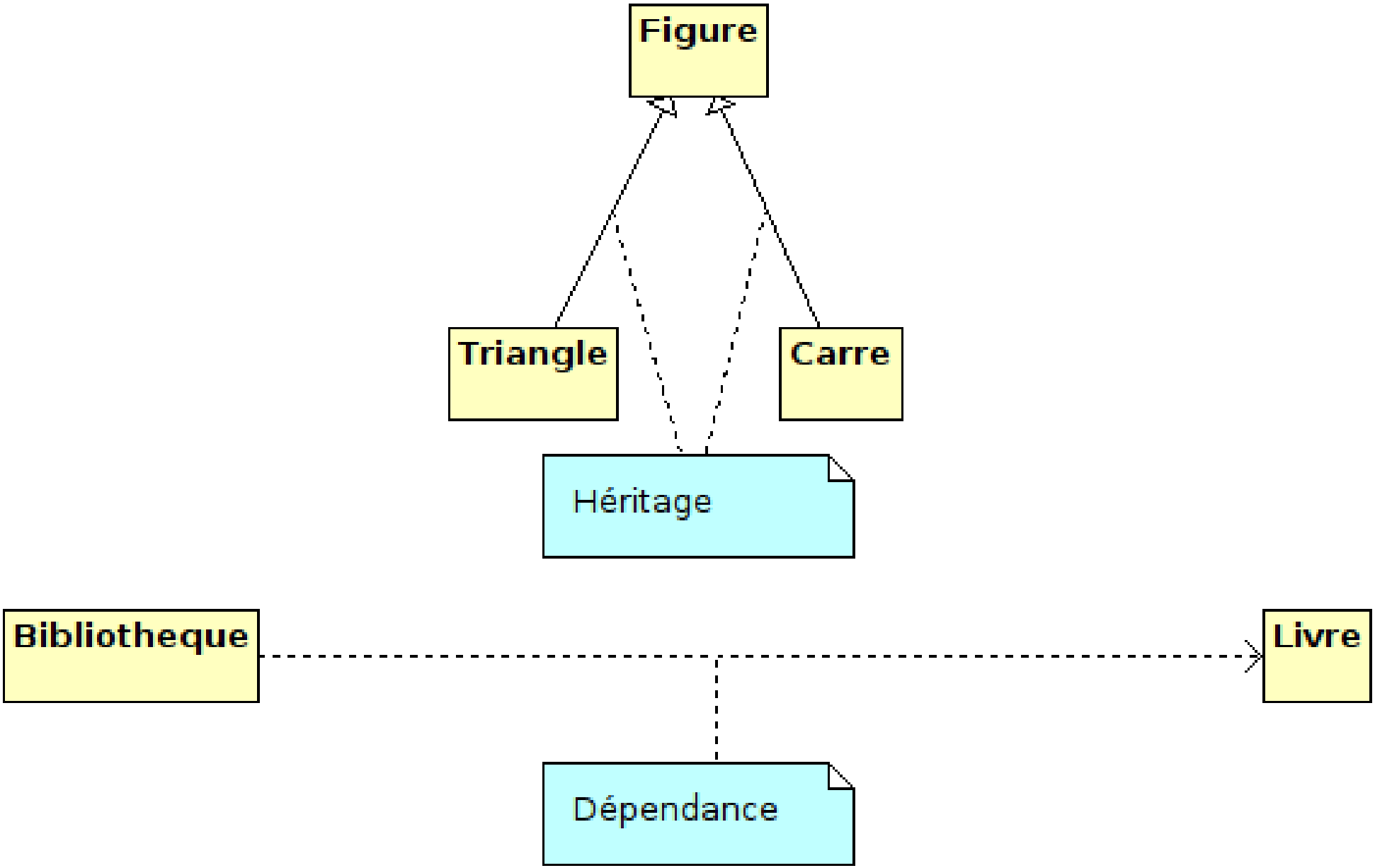
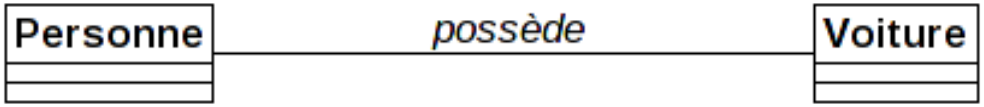


Figure
Triangle
Carre
Héritage
Bibliotheque
Livre
Dépendance

L'association

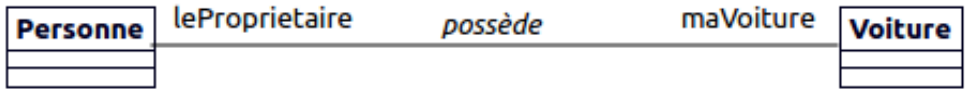
- Une **association** représente une **relation durable** entre deux classes.
- *Exemple :* Une personne peut posséder des voitures.
- La relation possède est une association entre les classes `Personne` et `Voiture`.
- Les associations peuvent donc être nommées pour donner un sens précis à la relation.



➠ Ici, la relation est bidirectionnelle, on a une **navigabilité** dans les deux sens. Ici, `Personne` « **possède** » `Voiture` et `Voiture` « **est possédé** » 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**.


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

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

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**.


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

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

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

La composition en C++ (fin)

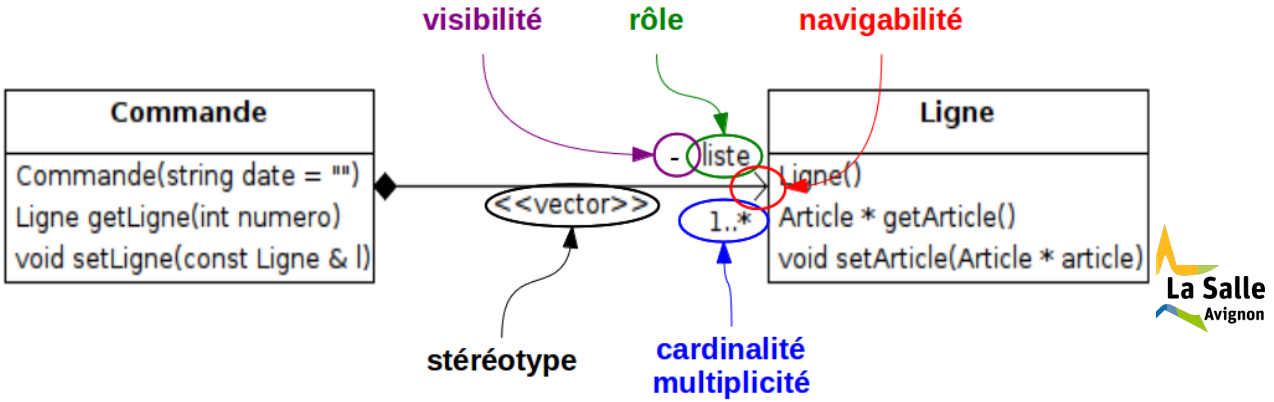
➠ En C++, la composition pourrait s'implémenter par **pointeur** en s'assurant d'allouer et de libérer l'instance souhaitée.

```
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

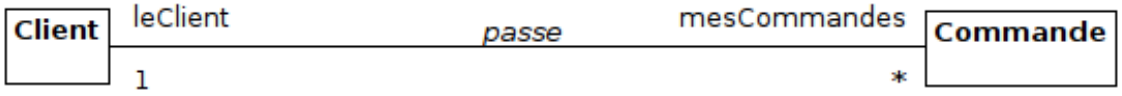
➠ Aux extrémités d'une relation, il est possible de préciser :

- 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` (indiqué dans le diagramme UML par un **stéréotype**).

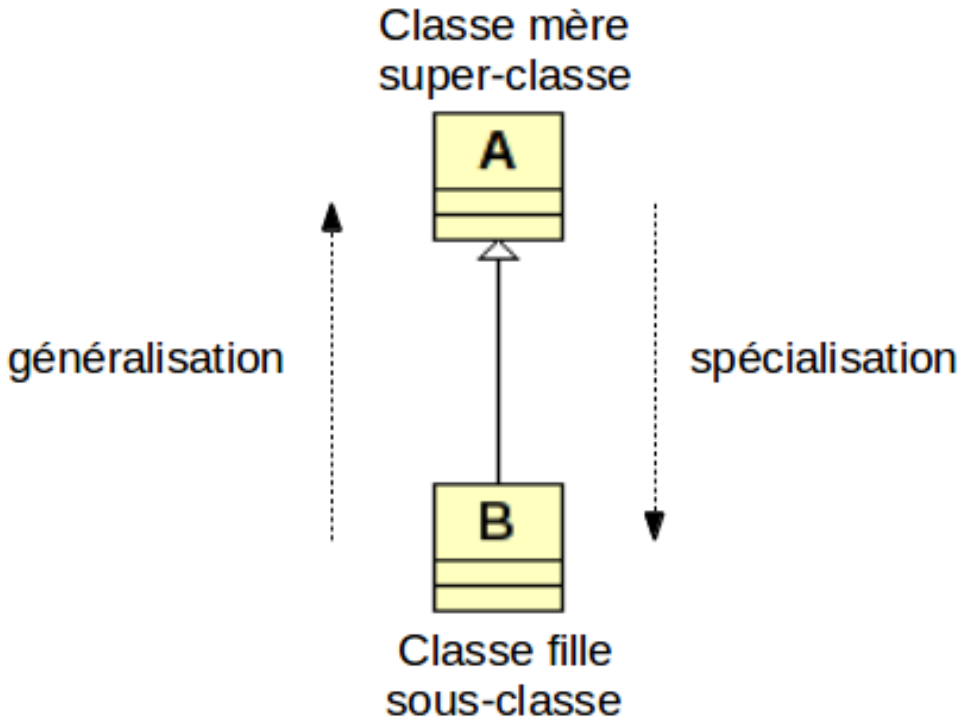



```
class Client
{
 private:
  list<Commande*> mesCommandes; // -> *
 ...
};
```

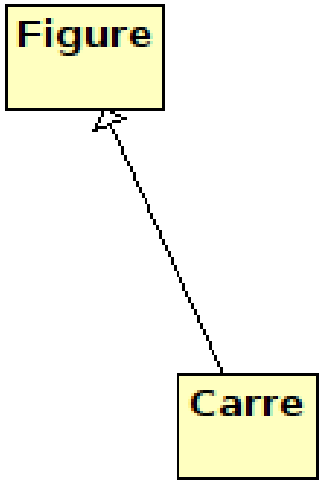
```
class Commande
{
 private:
  Client *leClient; // -> 1
 ...
};
```

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++

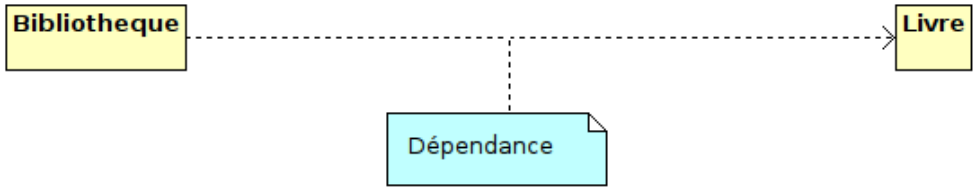



```
class Figure
{
   private:
      double x, y, z;
   public:
      Figure(double x=0, double y=0,
           double z=0) : x(x), y(y), z(
           z) {}
};
```

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

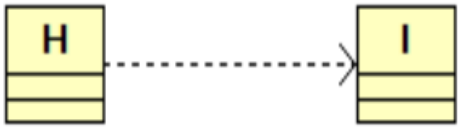
La dépendance

- 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.

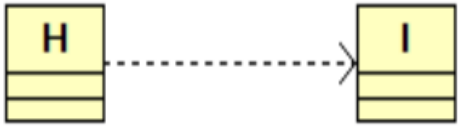


La dépendance en C++

UML	C++


Généralement, les dépendances ne sont pas montrées dans un diagramme de classes.
	`class H`
	`{`
	`  void faireQuelqueChose()`
	`  {`
	`  I i;` — *Objet temporaire car il n'existe que pendant la durée de l'exécution de la méthode.*
	`  // …`
	`  }`
	`};`
	`class I`
	`{`
	`};`

UML	C++


Généralement, les dépendances ne sont pas montrées dans un diagramme de classes.
	`class H`
	`{`
	`  void faire(I i)`
	`  {`
	`  // …` — *Objet temporaire car il n'existe que pendant la durée de l'exécution de la méthode.*
	`  }`
	`};`
	`class I`
	`{`
	`};`

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**.