Cours POO UML/C++ : les relations entre classes

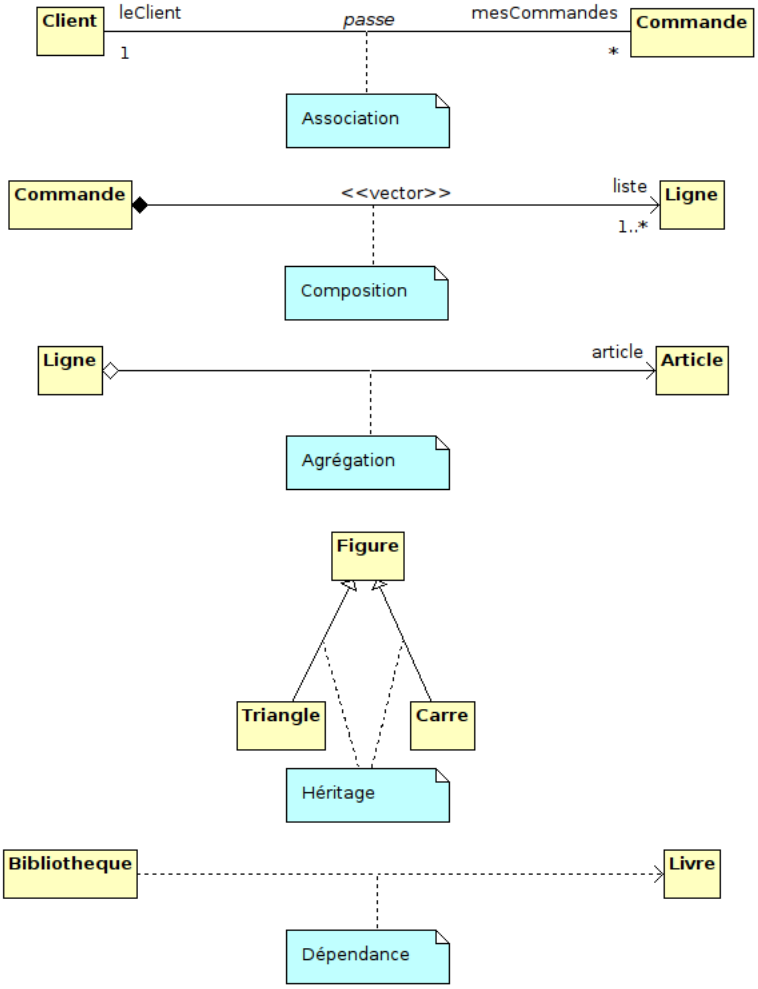
Sommaire

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

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

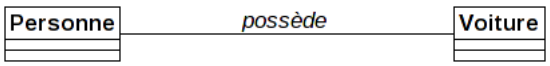
On distingue cinq différents types de **relations** de base entre les classes :

- 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).

Par exemple, 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`.

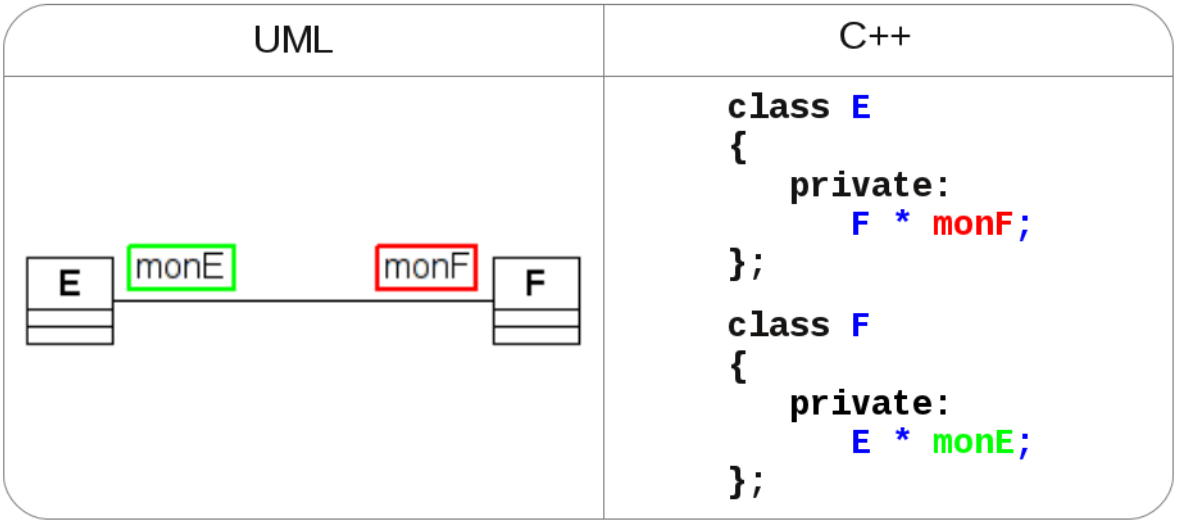


L'**agrégation** et la **composition** sont deux cas particuliers d'**association**.

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 :

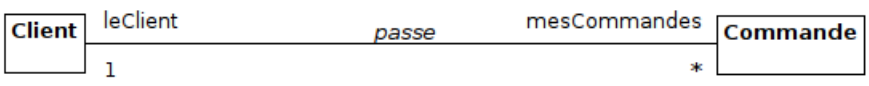


Ici, la relation est bidirectionnelle (pas de flèche), on a une navigabilité dans les deux sens. Ici, A « **connaît** » B et B « **connaît** » A. On peut remarquer que l'association se code de la même manière qu'une agrégation.

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 n), n..* (n ou plus) ou * (plusieurs).

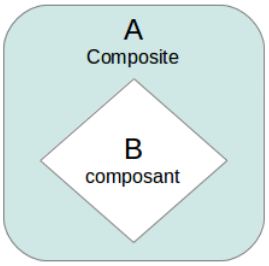
Ce diagramme de classe ci-dessous illustre une relation d'association 1 vers plusieurs (*) entre `Client` et `Commande` :



On utilisera des accesseurs *get* et *set* pour mettre en place la relation (voir plus loin).

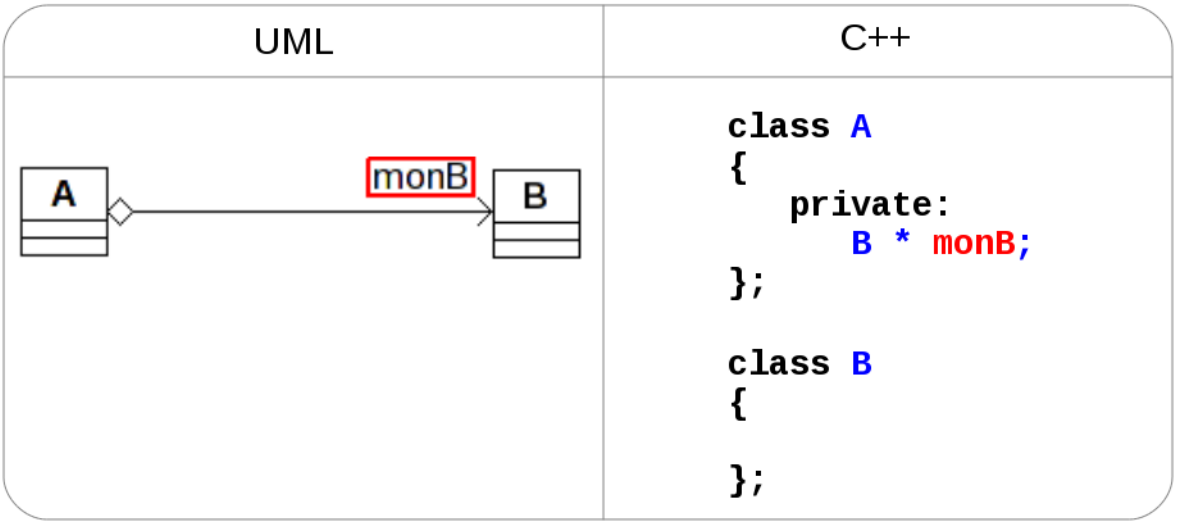
La relation d'agrégation

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

Ici, `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** dans la classe `A`.

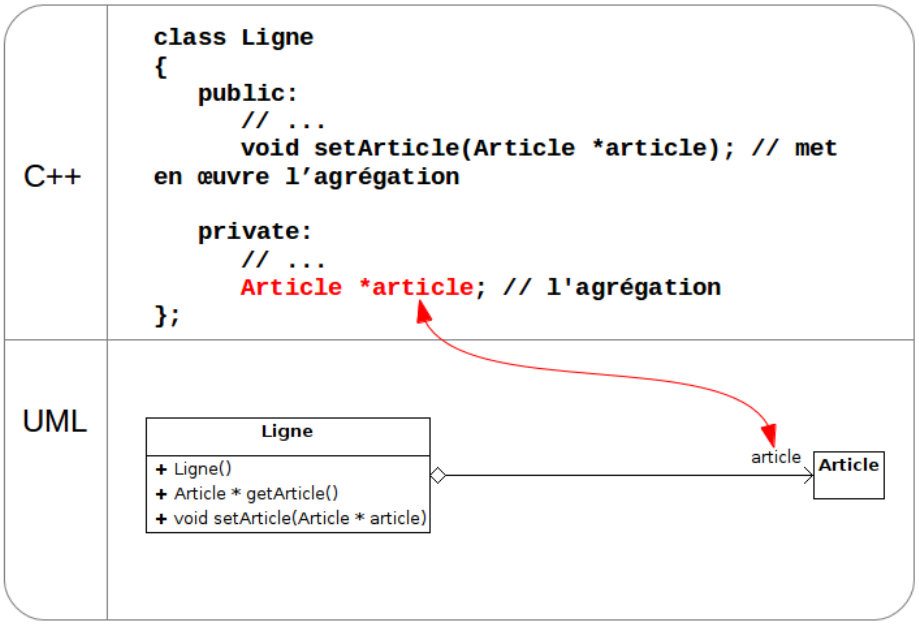
La flèche sur la relation précise la navigabilité. Ici, A « **connaît** » B mais pas l'inverse. Les relations peuvent être bidirectionnel (pas de flèche) ou unidirectionnel (avec une flèche qui précise le sens).

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

Les accesseurs `getArticle()` et `setArticle()` permettent de gérer la relation `article`.

La relation de composition (voir plus loin) 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 est bien le bon choix.

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



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
```

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

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

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

Ligne.cpp

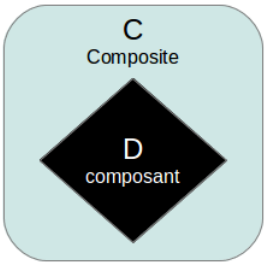
(2) Sans cette ligne, on va obtenir des erreurs à la compilation car celui-ci ne connaît pas "suffisamment" le type `Article` : `erreur: invalid use of incomplete type 'struct Article'`. Pour corriger ces erreurs, il suffit d'**inclure la déclaration (complète) de la classe `Article`** qui est contenue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

Voici un exemple d'utilisation de ces deux classes :

```
Article a1; // un objet Article
Ligne l1; // un objet Ligne

l1.setArticle(&a1); // met en place l'agrégation entre l1 et a1
```

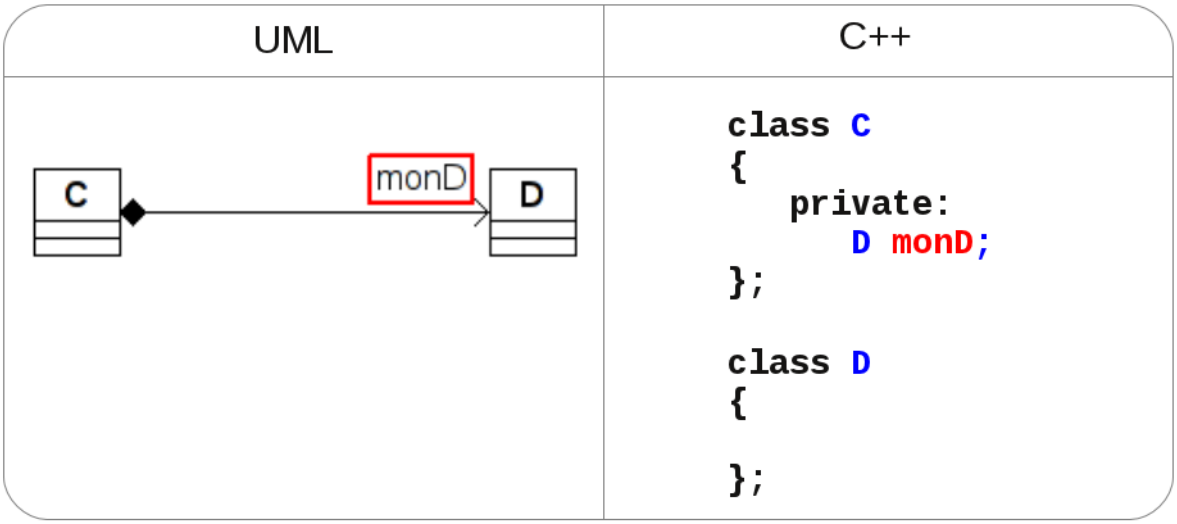

La relation de composition

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).

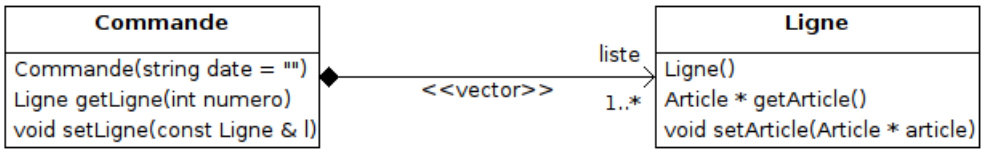


Composition

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



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



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.

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
```

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()
{
}

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

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

Commande.cpp

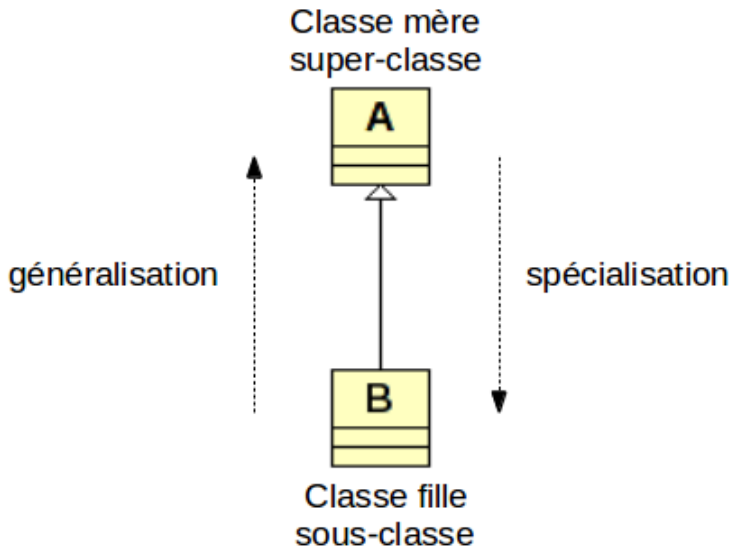
Voici un exemple d'utilisation de ces deux classes :

```
Article a1;
Article gratuit;
Ligne l1;
Ligne cadeau;
Commande c;

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

c.setLigne(l1);
c.setLigne(cadeau);
```


La relation d'héritage

L'**héritage** est un **concept fondamental de la programmation orientée objet**. Elle se nomme ainsi car le principe est en quelque sorte le même que celui d'un arbre généalogique. Ce principe est fondé sur des **classes « filles »** qui héritent des caractéristiques des **classses « mères »**.

L'héritage permet **d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise**. Il permet donc **la spécialisation ou la dérivation de types**.



B **hérite** de A : "un B **est** un A avec des choses en plus". Toutes les instances de B sont aussi des instances de A.

On dit aussi que B dérive de A. A est une généralisation de B et B est une spécialisation de A.

UML	C++

	`class A`
	`{`
	
	`};`
	
	`class B : public A`
	`{`
	
	`};`

L'héritage est **une relation entre classes** qui a les propriétés suivantes :
- si B hérite de A et si C hérite de B alors C hérite de A
- une classe ne peut hériter d'elle-même
- si A hérite de B, B n'hérite pas de A
- il n'est pas possible que B hérite de A, C hérite de B et que A hérite de C
- le C++ permet à une classe C d'hériter des propriétés des classes A et B (héritage multiple)

Rappels : Le C++ permet de préciser le ***type d'accès des membres*** *(attributs et méthodes) d'un objet. Cette opération s'effectue au sein des classes de ces objets.*

Il faut maintenant tenir compte de la situation d'héritage :

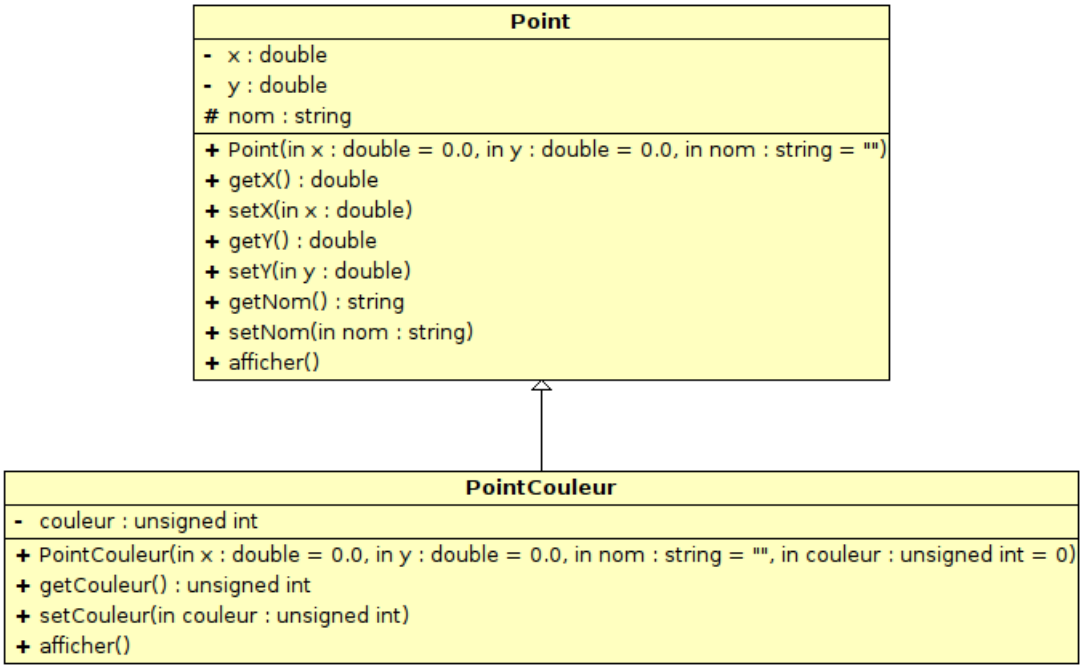
- **public** : les membres publics peuvent être utilisés dans et par n'importe quelle partie du programme.
- **privé** (*private*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **non par ceux d'une autre classe même dérivée**.
- **protégé** (*protected*) : les membres privés d'une classe ne sont accessibles que par les objets de cette classe et **par ceux d'une classe dérivée**.

Exemple :

On veut manipuler des **points**. Un **point** est défini par **son abscisse** (`x`), **son ordonnée** (`y`) et **son** `nom`. L'abscisse et l'ordonnée d'un point sont des **réels** (`double`) et le nom est une **chaîne de caractères** (`string`). Par défaut, un point sera défini avec les coordonnées (`0,0`) et un nom vide (`""`). La classe `Point` possèdera des accesseurs *get* et *set* pour chaque attribut. On ajoutera aussi un service `afficher()` car un point pourra s'**afficher** de la manière suivante : `<nom:x,y>`

On désire ensuite manipuler des **points** en **couleur**. Un **point** en **couleur** est un **point** avec quelque chose en plus : **sa** `couleur`. On va donc utiliser **l'héritage** pour créer la classe `PointCouleur` en y ajoutant un attribut `couleur` qui sera un **entier strictement positif**. La couleur doit représenter une valeur sur 32 bits composée de 4 octets (la transparence, la composante Rouge, la composante Verte et la composante Bleue). Par défaut, un point en couleur sera défini avec les coordonnées (`0,0`), un nom vide (`""`) et une couleur égale à `0` (`#00000000`). La classe `PointCouleur` possèdera elle aussi des accesseurs *get* et *set* pour chaque attribut. On modifiera le service `afficher()` car un point en couleur pourra s'**afficher** de la manière suivante : `<nom:x,y> [#couleur]`

On obtient le diagramme de classe :



Voici un exemple d'implémentation en C++ de cet héritage :

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

using namespace std;
```



```
class Point
{
   private: // accessible par personne même pas PointCouleur
      double x;
      double y;

   protected: // accessible par personne sauf PointCouleur
      string nom;

   public: // accessible par tout le monde
      Point(double x=0.0, double y=0.0, string nom="");

      // Accesseurs
      double getX() const;
      void setX(double x);
      double getY() const;
      void setY(double y);
      string getNom() const;
      void setNom(string nom);

      // Service(s)
      void afficher() const;
};

Point::Point(double x, double y, string nom) : x(x), y(y), nom(nom)
{
}

double Point::getX() const
{
    return x;
}

void Point::setX(double x)
{
    this->x = x;
}

double Point::getY() const
{
    return y;
}

void Point::setY(double y)
{
    this->y = y;
}

string Point::getNom() const
{
    return nom;
}
```



```
void Point::setNom(string nom)
{
    this->nom = nom;
}

void Point::afficher() const
{
    cout << "<" << nom << ":" << x << "," << y << ">";
}

class PointCouleur : public Point
{
   private: // accessible par personne même pas Point
      unsigned int couleur;

   public: // accessible par tout le monde
      PointCouleur(double x=0.0, double y=0.0, string nom="", unsigned int couleur=0);

      // Accesseur(s)
      unsigned int getCouleur() const;
      void setCouleur(unsigned int couleur);

      // Service(s)
      void afficher() const; // redéfinition par surcharge

      // Tests des accès aux membres hérités
      void modifierX(double x);
      void modifierY(double y);
      void modifierNom(string nom);
};

// Appeler en premier le contructeur de la classe mère (ici Point)
PointCouleur::PointCouleur(double x, double y, string nom, unsigned int couleur) : Point(x,
    y, nom), couleur(couleur)
{
}

unsigned int PointCouleur::getCouleur() const
{
    return couleur;
}

void PointCouleur::setCouleur(unsigned int couleur)
{
    this->couleur = couleur;
}

void PointCouleur::afficher() const
{
    Point::afficher(); // appel de la méthode afficher de la classe mère Point
    cout << " [#" << std::hex << std::uppercase << couleur << "]"; // ajoute l'affichage de
       la couleur en hexa
```



```
}

void PointCouleur::modifierX(double x)
{
    //this->x = x; // erreur: 'double Point::x' is private
}

void PointCouleur::modifierY(double y)
{
    //this->y = y; // erreur: 'double Point::y' is private
}

void PointCouleur::modifierNom(string nom)
{
    this->nom = nom; // accessible par héritage car nom est protected
}

int main()
{
    Point p1; // ou Point p1(0,0,"");

    //p1.x = 1; // erreur: 'double Point::x' is private
    //p1.nom = "A"; // erreur: 'std::string Point::nom' is protected
    p1.afficher(); // produit : <:0,0>

    Point p2(2, 2, "P2");

    p2.afficher(); // produit : <P2:2,2>

    PointCouleur p3(2.5, 5.5, "P3", 0xFFFFFFFF);

    p3.afficher(); // produit : <P3:2.5,5.5> [#FFFFFFFF]

    // Tests :
    p3.setX(1.2); // accessible par héritage : PointCouleur est un Point
    p3.setY(3.5); // idem
    p3.afficher(); // produit : <P3:1.2,3.5> [#FFFFFFFF]

    p3.modifierX(0.); // voir la méthode
    p3.modifierY(0.); // idem
    p3.modifierNom("O"); // idem
    p3.afficher(); // produit : <O:1.2,3.5> [#FFFFFFFF]

    return 0;
}
```

L'héritage en action

Conclusion :

En utilisant l'héritage, il est donc possible **d'ajouter des caractéristiques** (ici une couleur), **d'utiliser les caractéristiques héritées** (x, y et nom) et de **redéfinir les méthodes héritées** (ici afficher).

Généralement, cette **redéfinition (*overriding*)** se fait par **surcharge** et permet de modifier le comportement hérité. Voir aussi le polymorphisme.

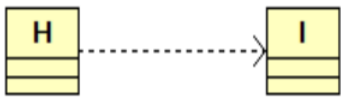
La relation de dépendance

Lorsqu'un objet "utilise" temporairement les services d'un autre objet, cette relation d'utilisation est une dépendance entre classes.

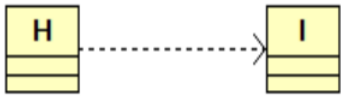
La plupart du temps, les dépendances servent à montrer qu'une classe utilise une autre comme argument dans la signature d'une méthode. On parle aussi de lien temporaire car il ne dure que le temps de l'exécution de la méthode. Cela peut être aussi le cas d'un objet local à une méthode.

Une dépendance s'illustre par une flèche en pointillée dans un diagramme de classes en UML :

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`
	`{`
	`};`

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.

Bilan

On doit souvent répondre à la question : **quelle relation choisir ?**

Finalement, cela reste assez simple si on considère que :

- les relations d'association, d'agrégation et de composition illustre une relation de type "**avoir**" ;
- la relation d'héritage illustre une relation de type "**être**".

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**.

