**Programmation Orientée Objet – Prosit 3**

*Prosit Aller : 08/11/2021*

Animateur : Florian

Scribe : Théophile

Secrétaire : Célian

Gestionnaire : Paul

**Prendre connaissance de la situation et clarifier**

*Mots-clés*

* Classe
* Parcours de points
* Parcours 2D et 3D
* Programme
* Partie redondante

*Établir le contexte*

Nous avons récupéré le programme dans un repère 2D, maintenant dans un repère 3D

**Analyser le besoin**

*Analyser et décomposer le problème*

*Comment transformer un point 2D en point 3D ?*

*Comment éliminer les redondances ?*

*Comment faire le lien entre les différentes classes(héritage/header) déjà crées ?*

*Livrables*

* Code
* Note explicative
* Infrastructure du programme (.cpp, .h, répertoires)

*Contraintes*

* Code de base
* Main

**Généraliser**

*Saisir la nature du problème et déterminer la famille de problèmes à laquelle il appartient*

* Dépendances (interclasse)
* Modélisation
* Héritages

**Trouver des pistes de solutions**

*Identifier des approches de solutions possibles*

* On va devoir supprimer l’une des coordonnées pour passer de la 3D a la 2D. (Mattéo)
* Tous les points sont exprimés en 3D mais leurs Z=0 (Valentin)
* Lorsque l’on fait un point 2D venant d’un point en 3D, on ignore une des coordonnées (Pierre)
* On va créer 4 classes au total (Pierre)
* On va utiliser une classe avec un tableau de coordonnées en allocation dynamique
* La classe « Point 3D » utilisera les attributs de la classe 2D en héritage

Schéma de Mattéo

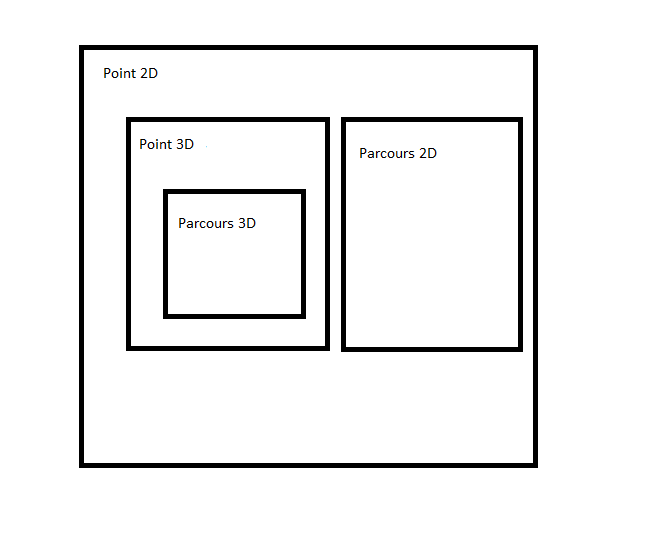
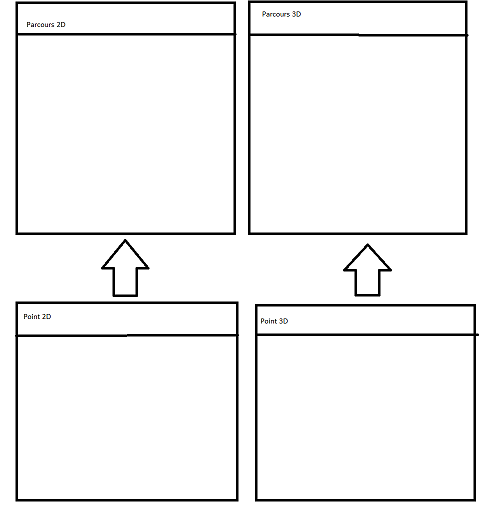


Schéma Pierre :



NB : Qu’est ce qui est redondant dans sa structure ? X et Y

* Pour le parcours 3D, on peut passer par le parcours 2D. (Célian)
* Il faudrait faire des classes non instanciable pour utiliser l’héritage (classe abstraite) ? (Florian)
* On va créer une classe abstraite ? (Paul)

**Elaborer le plan d’actions**

*Identifier les éléments d’étude nécessaires à la résolution du problème*

* *Héritage*
* *Polymorphisme*
* *Classe abstraite*
* *Classe virtuelle*

*Identifier les éléments de réalisation à produire*

* *Cf livrable*

*Liste, organiser, planifier et formaliser les actions à mener pour résoudre le problème*

* *Diagramme de classe(1h)*
* *Code (Debug/classe pour que le main fonctionne)*
* *Faire la note explicative (30 mins)*

**Phase AER :**

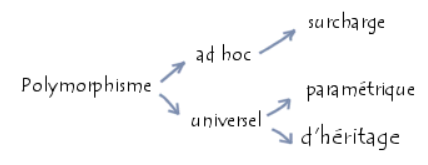
Dans ce prosit, nous avons eu l’occasion de voir différentes notions telles que :

* Le polymorphisme ;
* Les dépendances ;
* L’héritage ;
* Les classes abstraites ;
* Les fonctions virtuelles.

Dans un premier temps, nous allons parler du polymorphisme. Il faut d’abord savoir que le nom polymorphisme vient du grec et veut dire plusieurs formes. Cette caractéristique est un des concepts essentiels de la programmation orientée objet. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est relatif aux méthodes des objets.

Il existe plusieurs type de polymorphisme :

* Le polymorphisme ad hoc (aussi appelé surcharge ou overloading) ;
* Le polymorphisme paramètrique ;
* Le polymorphisme d’héritage (aussi appelé redéfinition, spécialisation ou overriding).



Nous allons donc définir dans un premier temps **le polymorphisme ad hoc**, il permet **d'avoir des fonctions de même nom**, avec des **fonctionnalités similaires**, **dans des classes sans aucun rapport entre elles** (si ce n'est bien sûr d'être des filles de la classe objet). Par exemple, la classe complexe, la classe image et la classe lien peuvent avoir chacune une fonction "afficher". Cela permettra de ne pas avoir à se soucier du type de l'objet que l'on a si on souhaite l'afficher à l'écran.

**Le polymorphisme ad hoc** permet ainsi de définir des opérateurs dont l'utilisation sera différente selon le type des paramètres qui leur sont passés. Il est donc possible par exemple de surcharger l'opérateur + et de lui faire réaliser des actions différentes selon qu'il s'agit d'une opération entre deux entiers (addition) ou entre deux chaînes de caractères (concaténation).

Nous allons maintenant passer au g. Le polymorphisme paramétrique, **appelé généricité**, représente la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents (en nombre et/ou en type). Le polymorphisme paramétrique rend ainsi possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre.

Ainsi, on peut par exemple définir plusieurs méthodes homonymes addition() effectuant une somme de valeurs.

* La méthode int addition(int, int) pourra retourner la somme de deux entiers ;
* La méthode float addition(float, float) pourra retourner la somme de deux flottants ;
* La méthode char addition(char, char) pourra définir au gré de l'auteur la somme de deux caractères.

On appelle signature le nombre et le type (statique) des arguments d'une fonction. C'est donc la signature d'une méthode qui détermine laquelle sera appelée.

Nous allons enfin parler du **polymorphisme d’héritage**. La possibilité de redéfinir **une méthode dans des classes héritant d'une classe de base** s'appelle **la spécialisation**. Il est alors possible d'**appeler la méthode d'un objet sans se soucier de son type intrinsèque** : il s'agit du polymorphisme d'héritage. Ceci permet de faire abstraction des détails des classes spécialisées d'une famille d'objet, en les masquant par une interface commune (qui est la classe de base).

Imaginons un jeu d'échec comportant des objets roi, reine, fou, cavalier, tour et pion, héritant chacun de l'objet piece. La méthode mouvement() pourra, grâce au polymorphisme d'héritage, effectuer le mouvement approprié en fonction de la classe de l'objet référencé au moment de l'appel. Cela permettra notamment au programme de dire piece.mouvement sans avoir à se préoccuper de la classe de la pièce.

Nous allons à présent parler des dépendances, il faut savoir que les dépendances sont en générale utiliser pour l’UML du c++. En effet, dans la programmation orienté objet, les objets logiciels interagissent entre eux, il y a des donc des relations entre les classes.

Il faut savoir qu’il existe quatre type 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).

Cependant, il y a une relation temporaire, c’est la dépendance (représentée par des flèches en pointillés). Il faut savoir que l’agrégation et la composition sont deux cas particuliers d’association.

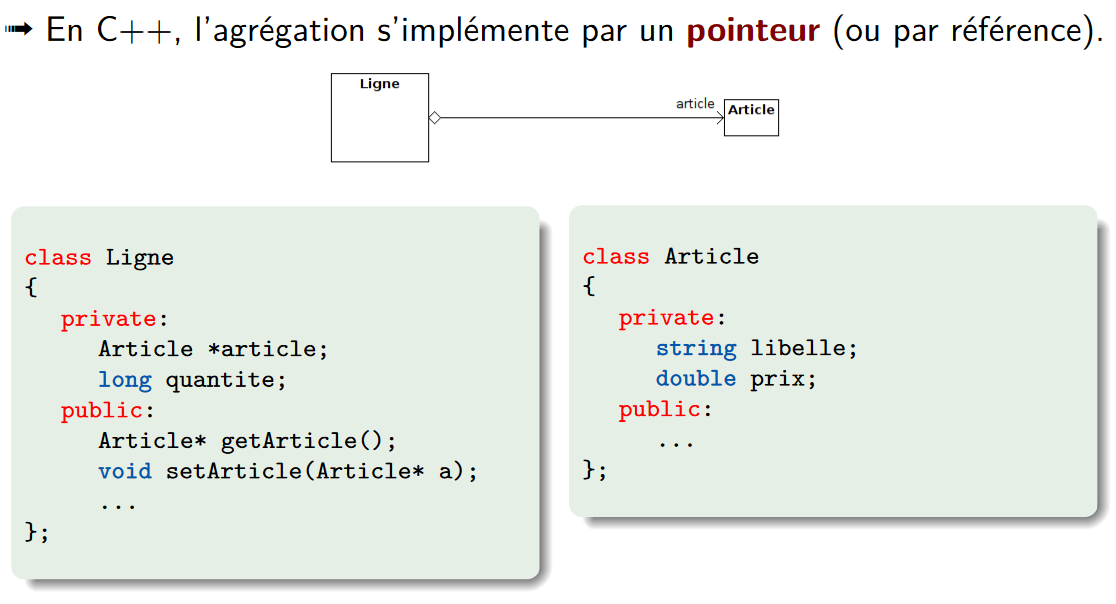
Nous allons maintenant décrire plus précisément l’association. Il faut savoir qu’une **association** est une **relation durable** entre deux classes. Par exemple, une **personne** possède une **voiture**. La relation possède est une association entre les classes Personne et Voiture. Les associations peuvent donc être nommées pour donner un sens précis à la relation.

Il faut savoir qu’en C++, une association s’implémente par le biais d’un pointeur. Les assesseurs get/set permettent de mettre en œuvre la relation.

Une image contenant texte

Description générée automatiquement

Concernant **l’agrégation**, c’est un cas particulier d’association non symétrique exprimant une relation de contenance. Par 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**.



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).

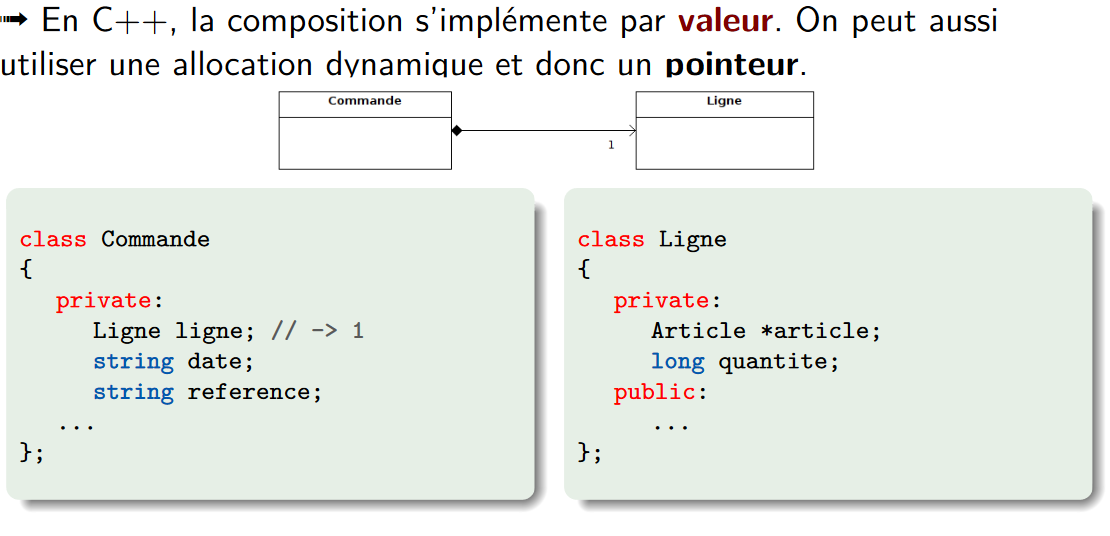
Une image contenant texte

Description générée automatiquement

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).

Par 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.



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.

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

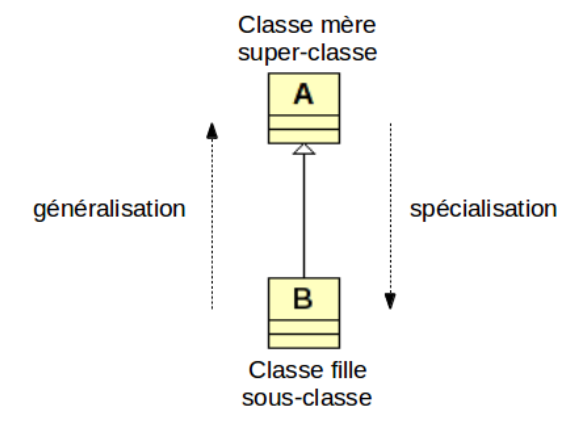
Pour pouvoir conserver plusieurs instances (c’est-à-dire plusieurs objets), on doit utiliser un conteneur de type vector, list ou map par exemple. Cela pourra être précisé dans le diagramme UML par un stéréotype.

Une image contenant texte

Description générée automatiquement

Parfois, une association doit posséder des propriétés. Par 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.

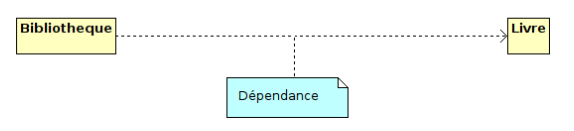
L’héritage permet d’ajouter des éléments (propriétés et/ou comportement) à une classe existante pour en obtenir une nouvelle plus précise. Par exemple : Un wagon-bar est un wagon avec quelque chose en plus (un bar).

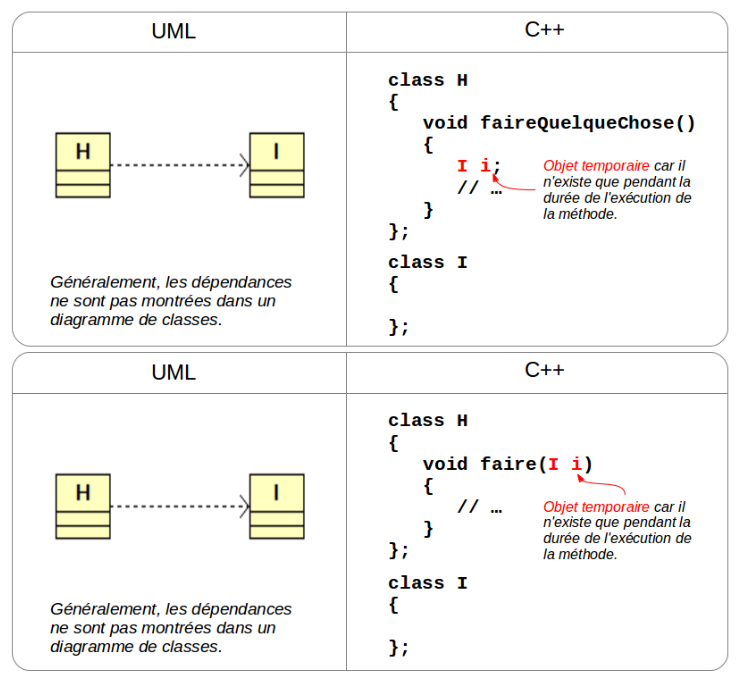


Une image contenant texte

Description générée automatiquement

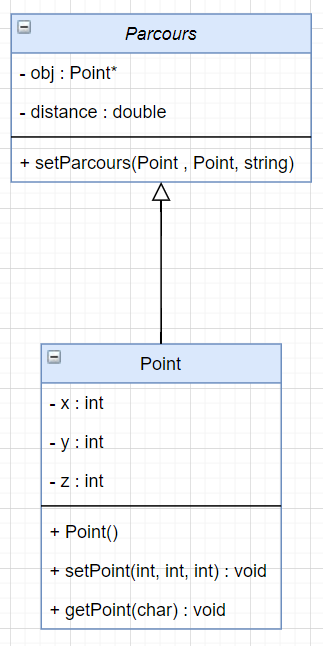
Lorsqu’un objet « utilise » temporairement les services d’un autre objet, cette relation d’utilisation est une dépendance entre classes. Une dépendance s’illustre par une flèche en pointillée dans un diagramme de classes en UML. Par exemple : un objet livre passé en paramètre de la méthode emprunter() 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.





Pour le prosit, nous avons donc fait ce diagramme de classe. Comme nous pouvons le voir, nous avons deux classes :

* Parcours ;
* Point.



Nous allons maintenant expliquer le concept des classes. Il faut savoir que c’est la notion la plus importante de la programmation orientée objet. Nous pouvons les comparer à des moules ou des patrons qui permettent de créer des objets en série sur un même modèle. Il y a deux choses à connaître dans les classes :

* Les accesseur (getter) sont des méthodes qui permettent de récupérer la valeur d’un attribut.
* Les mutateurs (setter) sont des méthodes qui permettent de modifier la valeur d’un attribut.

Nous utilisons ces méthodes car elles font parti du concept d’encapsulation, notamment lorsque nous voulons contrôler les accès en écriture ou en lecture des attributs.

Nous allons développer un peu plus la notions d’encapsulation. En POO, il y a différents types d’attributs :

* L’attribut public (+ en UML) autorise l'accès pour tous. Sur l'exemple **Personne** précédent, le constructeur ainsi que la méthode **quiSuisJe** peuvent être utilisés partout sur une instance de Personne ;
* L’attribut privée (private) (- en UML) restreint l'accès aux seuls corps des méthodes de cette classe. Sur l'exemple précédent, les attributs privés **my\_prenom** et **my\_nom** ne sont accessibles sur une instance de **Personne** que dans les corps des méthodes de la classe **Personne**. Ainsi, la méthode **quiSuisJe** a le droit d'accès sur ses attributs **my\_prenom** et **my\_nom**. Elle aurait aussi l'accès aux attributs **my\_prenom** et **my\_nom** d'une autre instance de **Personne** ;
* L’attribut protégé (protected) (# en UML) comme private mais l'accès est aussi autorisé aux corps des méthodes des classes qui héritent (directement ou indirectement) de cette classe.

Nous allons maintenant parler des classes abstraites et des fonctions virtuelles. Une fonction virtuelle est une fonction définit dans une classe (autrement dit une méthode) qui est destinée à être redéfinit dans les classes qui en héritent. Une fonction ne possédant qu’une déclaration, sans code implémenté, est dite abstraite ou virtuelle pure. Les méthodes virtuelles pures sont destinées à être définies dans les classes dérivées.

Une image contenant table

Description générée automatiquement

Nous allons enfin finir avec les classes abstraites. Une classe abstraite est une classe si est seulement si elle n’est pas instanciable. Elle sert de base pour d’autres classes dérivée (hérités). En C++, une classe est abstraite si elle contient au moins une méthode déclarée virtuelle pure, c'est-à-dire commençant par virtual et terminée par = 0. Ce type de classe n'est pas instanciable. On peut néanmoins créer des pointeurs ou des références du type de la classe abstraite qui pointeront ou référenceront des objets d'une classe dérivée non abstraite. Une fonction virtuelle pure doit être définie ou redéclarée explicitement virtuelle pure.

Voici la corbeille d’exercice :

Classe.h

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement



Classe.cpp

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Corbeille.cpp

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte, plaque, capture d’écran

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Voici la résolution du prosit :

Classe.h

Une image contenant texte

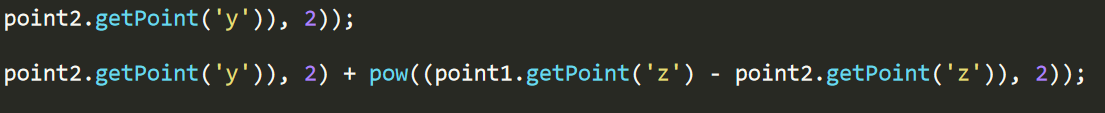
Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement



Prosit3.cpp

Une image contenant texte

Description générée automatiquement Une image contenant texte

Description générée automatiquement Une image contenant texte

Description générée automatiquement Une image contenant texte

Description générée automatiquement

Pistes d’amélioration :

Dans prosit, nous aurions pu chercher à faire une modélisation, soit en créer une interface graphique. Comme celle -ci par exemple :

<https://www.youtube.com/watch?v=ih20l3pJoeU>

Nous aurions aussi pu faire une simulation Unity pour modéliser les points dans le repère.