Compte Rendu de TP : C++ n°2 : Héritage et Polymorphisme

Binôme B3307 : Saad GUESSOUS et Jules DUCANGE

Introduction

L'objectif de ce TP est de développer une application permettant à un utilisateur d'intéragir avec un catalogue de trajets. L'utilisateur doit pouvoir afficher le catalogue, ajouter un trajet, et rechercher un trajet. L'application est développée en langage C++, en employant les concepts de base de la programmation orientée objet, tels que l'héritage et le polymorphisme.

Description détaillée des classes

L'application se décompose en 6 classes :

- Trajet
- TrajetSimple
- TrajetCompose
- Cellule
- ListeChainee
- Catalogue

`TrajetSimple` et `TrajetCompose` héritent de `Trajet`, qui est une classe abstraite. Cela permet donc de bénéficier des avantages de l'héritage et du polymorphisme, à savoir la manipulation de ces trajets dans le catalogue, sans distinction entre TrajetSimple et TrajetCompose. Les attributs VilleDepart, VilleArrivee et Transport sont présents uniquement dans la classe TrajetSimple, puisqu'un trajet composé est constitué de plusieurs trajets simples.

La classe `Cellule` et la classe `ListeChainee` implémentent une simple liste chaînée. `Cellule` inclus un pointeur sur `Trajet` et un sur `Cellule`. `ListeChainee` inclus un pointeur de début et de fin sur des `Cellule`. Pour simplifier l'utilisation de la liste chainée nous avons opté pour lier la classe `Cellule` et la classe `ListeChainee` par une relation d'amitié. Cela n'expose pas les cellules de la liste outre mesure et permet d'alléger la syntaxe lors de son implémentation.

> Ce choix d'implémentation pourrait être inversé si le besoin apparaissait lors d'une évolution future en implémentant les Getters et Setters nécessaires.

La classe `Catalogue`, comme son nom l'indique, permet de créer un objet permettant de stocker de façon ordonnée (par ordre alphabétique) des trajets que l'utilisateur rentre. Cette classe comprend aussi toutes les fonctionnalités utiles à l'utilisateur à savoir :

- L'affichage du catalogue
- L'ajout d'un trajet dans le catalogue
- La recherche simplifiée d'un trajet
- La recherche avancée d'un trajet

Le diagramme UML ci-dessous illustre les intéractions entres les différentes classes évoquées ci-dessus.

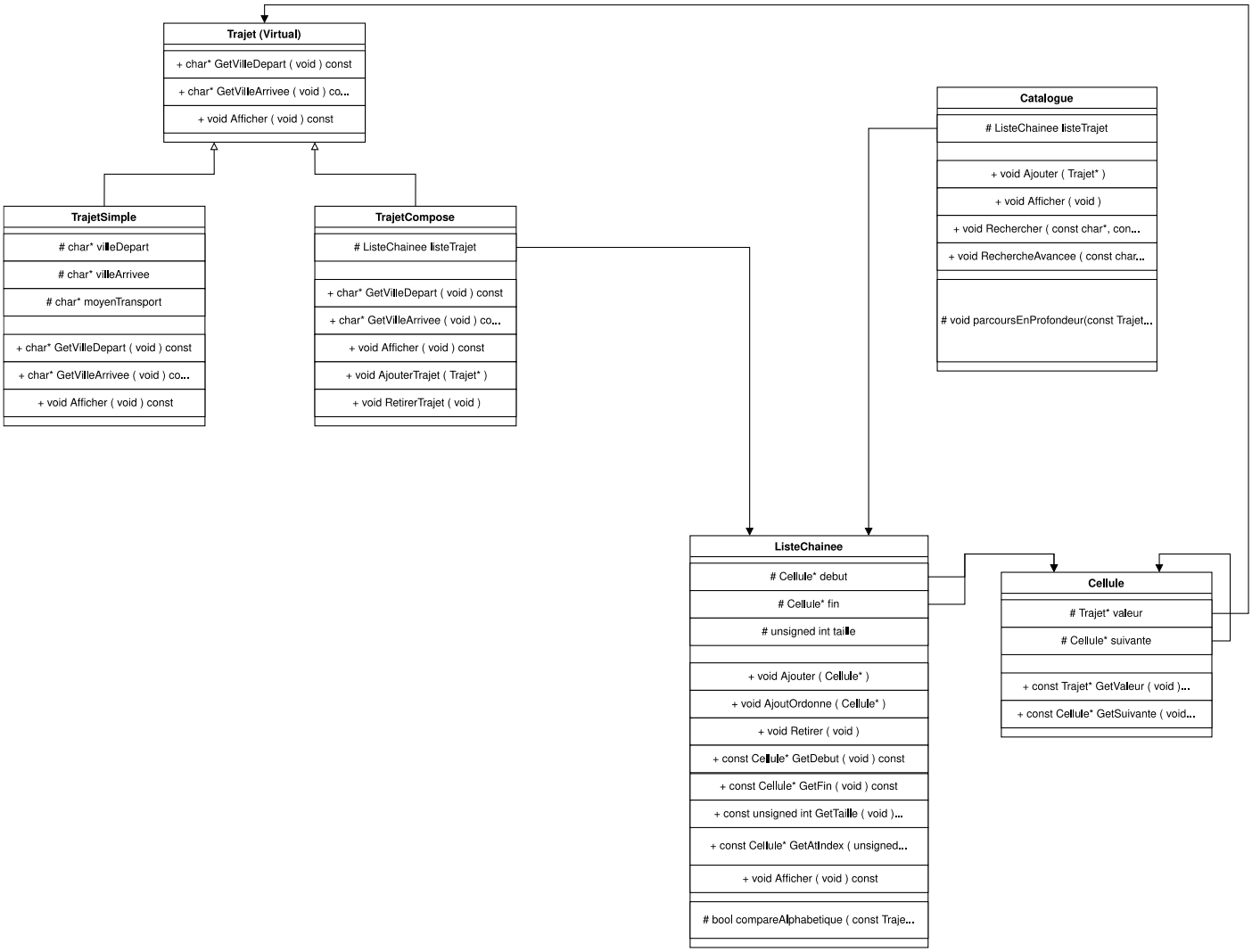


Figure 1 : Diagramme des classes UML

Description détaillée de la structure de données

Nous avons décidé de nous orienter vers une liste simplement chaînée, avec pointeur de fin, pour ce qui concerne notre structure de données.

Ce choix est justifié par plusieurs points, entre autres :

- L'ajout et la suppression de cellules de manière extrêmement simple et en O(1)
- Utilisation de la même structure de données pour l'implémentation du `Catalogue` et du `TrajetCompose`

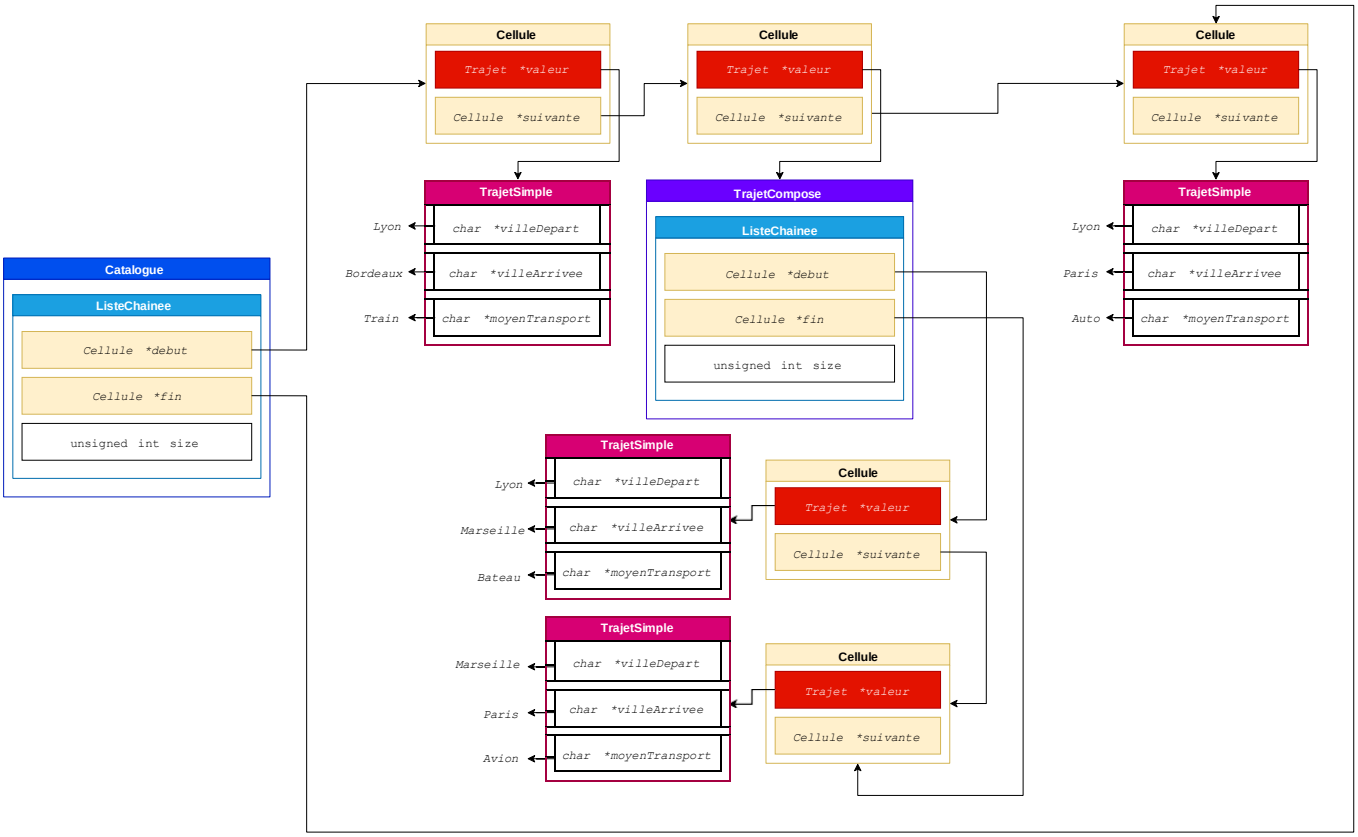
Malgré cela, cette structure de données reste très peu optimisée pour une opération de recherche (cf `Difficultés et contraintes rencontrées`).

La liste chaînée possède deux méthodes d'ajout. La première méthode, `Ajouter()` permet d'ajouter une cellule en fin de liste, et est utilisée pour l'implémentation des trajets composés. La deuxième, `AjouterOrdonne()` permet d'ajouter une cellule tout en respectant la condition de tri alphabétique desvaleurs (i.e trajets) des cellules.

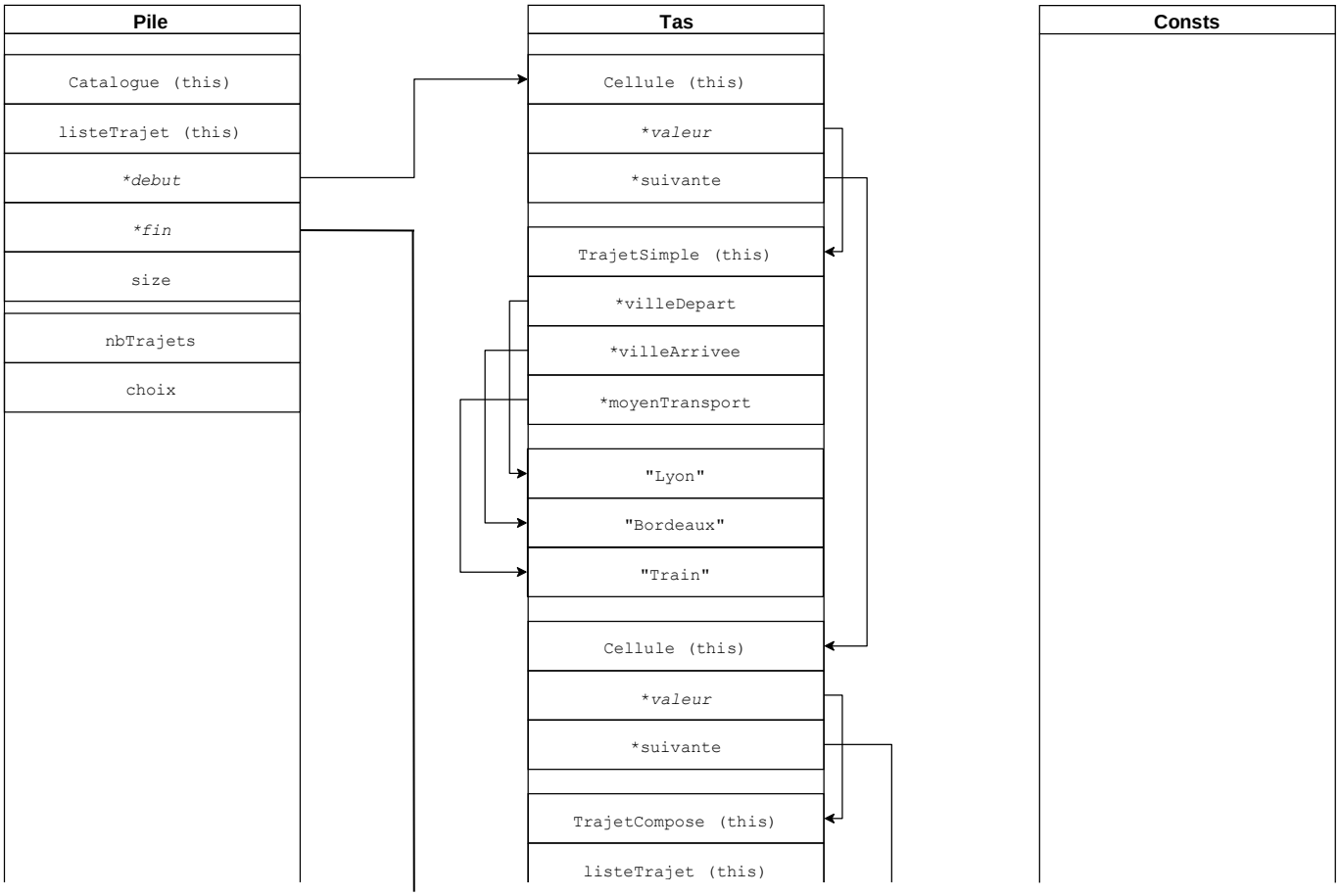
La liste chaînée possède aussi un attribut `taille` et une méthode `GetTaille()`, cette méthode est utile dans le parcours en profondeur pour la recherche avancée.

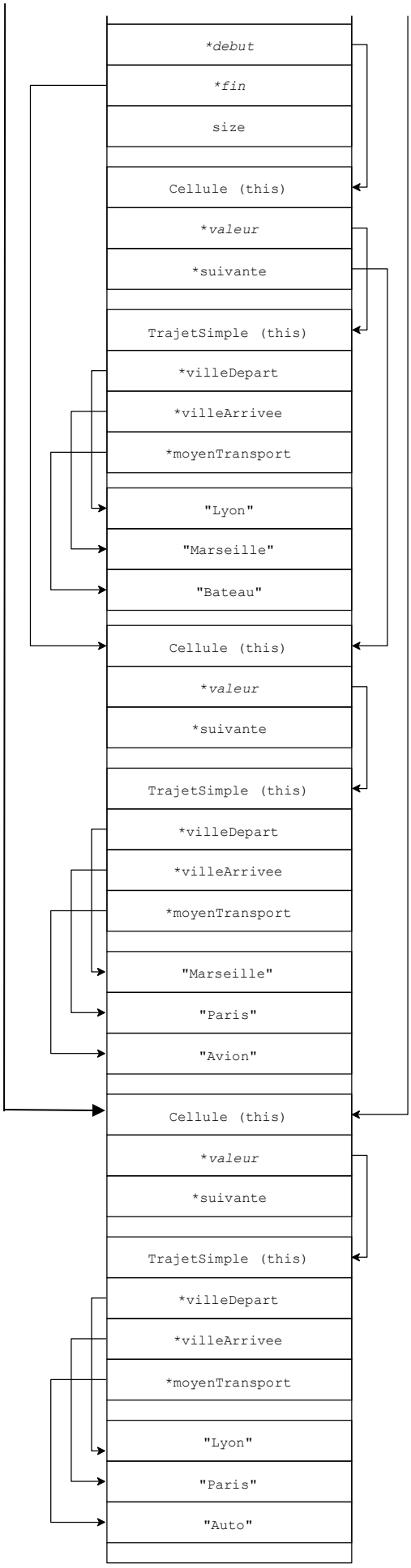
Voici ci-dessous un dessin de la structure de données, et un schéma de mémoire en 3 colonnes, réalisés pour le jeu de test fourni dans le sujet du TP :

Représentation simplifiée de la mémoire.



Représentation complète de la mémoire.




*debut
*fin
size
Cellule (this)
*valeur
*suivante
TrajetSimple (this)
*villeDepart
*villeArrivee
*moyenTransport
"Lyon"
"Marseille"
"Bateau"
Cellule (this)
*valeur
*suivante
TrajetSimple (this)
*villeDepart
*villeArrivee
*moyenTransport
"Marseille"
"Paris"
"Avion"
Cellule (this)
*valeur
*suivante
TrajetSimple (this)
*villeDepart
*villeArrivee
*moyenTransport
"Lyon"
"Paris"
"Auto"

> Cette représentation est faite une fois tous les trajets insérés, quand l'utilisateur se trouve devant le menu de l'application.

Difficultés et contraintes rencontrées

Difficultés liées à la structure de données choisie

La liste chaînée est une structure de données très optimale pour l'insertion et la suppression d'éléments. Toutefois, l'accès à un élément à partir d'un indice demande une complexité linéaire en O(n). Cela implique par exemple que la recherche simple, qui consiste à parcourir tout le catalogue et de récupérer les trajets correspondant à la recherche, possède une complexité en O(n).

Dans notre jeu d'essai, constitué seulement d'une dizaine de trajets, l'impact de cette complexité sur le temps d'exécution n'est pas flagrant, voire même négligeable. Toutefois, si l'application devait être utilisée avec une (très) longue liste de trajets, cette complexité en O(n) deviendrait très rapidement assez contraignante.

Pour ce qui est de la recherche avancée, le problème est d'autant plus important. En effet, notre algorithme de recherche avancée est un algorithme récursif qui parcourt en profondeur tout le catalogue. Cette méthode de recherche avancée a au final une complexité en $O(n^2)$, qui serait encore une fois, pas du tout convenable pour une liste de trajets un minimum imposante en terme de taille.

> Il est à noter que l'implémentation de la liste chainée comprends un pointeur de fin ce qui limite le parcours de la liste dans certains cas.

Autres difficultés

Le modèle de représentation des données impose une utilisation importante des fonctionnalités de polymorphisme du C++. Cela rend certaines implémentations plus complexes, par exemple il est impossible d'utiliser un constructeur de copie au niveau de la classe `Trajet`.

> **Note :** cela pourrait être résolu par l'implémentation d'une méthode purement virtuelle qui clone l'objet courant.

Axes d'évolution et d'amélioration :

Le plus gros point d'amélioration serait celui de l'interface utilisateur. En effet, malgré que celle-ci soit fonctionnelle et relativement ergonomique, elle reste toutefois assez limitée et pas très optimisée. Par exemple, si au lancement du programme l'utilisateur insère une lettre ou un caractère autre qu'un chiffre, il n'y a pas vraiment de gestion d'erreur et le comportement de l'application peut être assez hasardeux.

De même, la taille des buffers pour les villes de départ, d'arrivée et pour le moyen de transport est de 2048 caractères, ce qui normalement devrait être suffisant, étant donné que la ville avec le nom le plus long contient « seulement » 51 lettres. Toutefois si l'utilisateur dépasse cette limite, le programme ne fonctionnera pas correctement.

Un autre point assez dérangeant au niveau des inputs est que l'on ne peut pas rentrer de ville contenant un espace, par exemple si l'utilisateur rentre « New York », la ville de départ sera « New » et celle d'arrivée sera « York ».

On peut également noter qu'il est actuellement impossible pour l'utilisateur de supprimer ou modifier des trajets existants. Ce qui dans une application réelle serait indispensable.

Finalement une interface graphique aurait été bien plus agréable pour l'utilisateur mais cela demandebeaucoup plus de temps et s'éloigne du sujet du TP.

Conclusion

Ce projet nous a permis d'une manière assez condensée de découvrir toutes les bases essentielles de la programmation orientée objet en C++, en développant une application très concrète et utile dans le monde réel.

Étant le premier projet de cette première année d'Informatique, ce projet nous a également permis d'apprendre la gestion de travail en groupe pour des projets de développement. Notamment via l'utilisation des outils de gestion des sources `git` et `github`.

Le développement de la recherche avancée était un exercice d'algorithmique intéressant et agréable à mener. Cela faisant appel à des connaissances d'autres modules, comme la récursivité ou les graphes.