B3205 : Baptiste PAULETTO & Félix FONTENEAU

Compte rendu TP C++ n°2 : Héritage – Polymorphisme

L'ensemble du travail est disponible sur GitHub à l'url :

https://github.com/baptiste-pauletto1/tpHeritageC

A. Description détaillée de toutes les classes de notre application, avec le graphe d'héritage (UML) et sa justification : (ExplicationsJustifications.pdf sur GitHub)

La classe Trajet est une classe abstraite :

Deux attributs protégés :
- villeDepart qui est un char * qui correspond à la ville de départ du trajet.
- villeFin qui est un char * qui correspond à la ville d'arrivée du trajet.

Deux méthodes :
-afficher qui affiche sur la sortie standard comment se déroule le trajet (ville de départ, ville d'arrivée, potentielles villes escales, et moyens de transports).
- EnvoyerVille[Depart - Arrivee] qui retourne une chaine de charactère désignant la ville de départ/d'arrivée du trajet courant.

La classe TrajetSimple hérite de la classe Trajet :

Un attribut protégé (mais qui pourrait être privé) :
- moyenDeTransport qui est un char *.

Une méthode publique :
- EnvoyerMoyenDeTransport qui retourne une chaine de charactère désignant le moyen de transport du trajet simple.

La classe CollectionTrajet est composée de :

Trois attributs protégés :
- trajets correspond à la liste de trajets. C'est un tableau de pointeurs sur trajets constants.
- cardMax correspond à la taille maximale du tableau.
- cardActuelle est un indice permettant de savoir jusqu'où le tableau est actuellement rempli.

Trois méthodes publiques :
- Ajouter qui prend en paramètre la référence d'un trajet et l'ajoute au tableau trajets.
- EnvoyerNiemeTrajet qui renvoie le nième trajet, avec n correspondant à l'entier passé en paramètre.

La classe TrajetCompose hérite de la classe Trajet :

Un attribut protégé (mais qui pourrait être privé) :
- trajetsComposants, qui est une instance de la classe CollectionTrajet.

La classe Catalogue est composée de :

Un attribut privé :
- trajetsDisponibles, qui est une instance de la classe CollectionTrajet.

Trois méthodes :
- AjouterTrajet qui prend en paramètre la référence d'un trajet et l'ajoute à la collection de trajet : trajetsComposants.
- afficher qui va afficher la liste des tous les trajets sur la sortie standard.
- rechercherParcours qui prend en paramètre deux char *, le nom d'une ville de départ et d'une ville d'arrivée et affiche sur la sortie standard tous les trajets de la liste qui relient correctement la ville de départ et la ville d'arrivée.

Justification :

Nous avons choisi la répartition précédente car elle permet de relier les différents types de trajets au catalogue. Le point clef est la classe abstraite Trajet qui permet de pouvoir gérer une liste de trajets hétérogènes (simple ou complexes).
En effet, chaque trajetSimple ou trajetComposé est une sorte de trajet.

Chaque Trajet possède une ville de départ, une ville d'arrivée qui sont ces attributs mais aussi, chaque trajet doit pouvoir être affiché, renvoyer sa ville de départ et de fin.

La collection de trajet est notre structure de donnée permettant de pouvoir enregistrer les différents trajets sous forme d'un tableau de pointeurs de trajets constant. Cela permet de mieux gérer les trajets en mémoire. Cette structure de donnée est donc utilisée dans Catalogue mais aussi TrajetCompose. Cette encapsulation de la structure de données nous permet d'assurer l'indépendance du fonctionnement du système et de l'implémentation du stockage des données.

Un TrajetSimple va donc hériter des caractéristiques de la classe Trajet, mais va ajouter le moyen de transport, qui permet de relier la ville de départ à la ville d'arrivée.

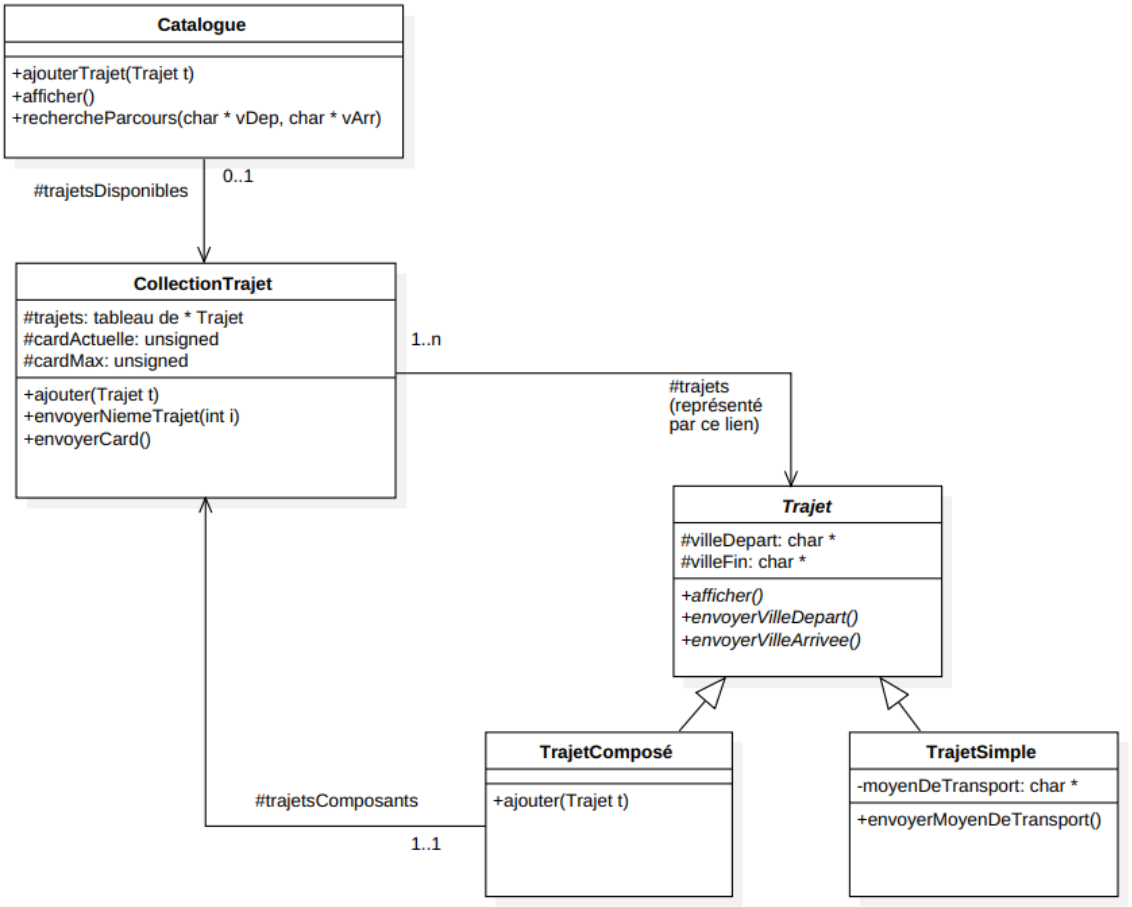
Un TrajetCompose va aussi hériter des caractéristiques de la classe Trajet, mais en se distinguant d'un trajet simple par sa composition. En effet, un trajet composé va être constitué de trajets simples et/ou de trajets composés. Il y aura donc une succession de trajets qui sont reliés par des villes étapes (pour chaque ville de fin de trajet composant différent de la ville d'arrivée, il existe une même ville de départ de trajet composant).
Ceci explique la présence d'une collection de trajet comme attribut de ladite classe.

Le Catalogue permet donc d'interagir avec l'utilisateur. Il a une Collection de trajets hétérogènes.

On peut ajouter des trajets au catalogue grâce à une méthode. On peut afficher l'ensemble des trajets du catalogue avec une méthode sans paramètres. On peut aussi rechercher un parcours reliant une ville de départ et une ville d'arrivée avec la méthode de recherche de parcours.

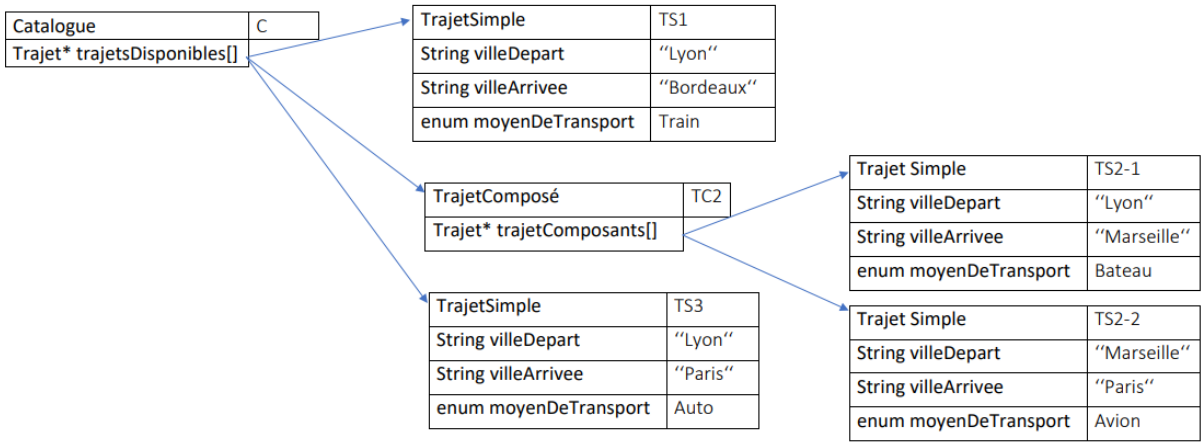
D'autre part, nous avons fait le choix de maximiser la présence de ''const'' dans notre réalisation afin d'avoir des contraintes fortes sur les diverses classes utilisant les trajets. En effet, cela nous permet d'affirmer que toutes les classes qui utilisent les Trajet (qu'ils soient composés ou non) ne peuvent en modifier le contenu et nous pensons que c'est un point important pour une réalisation gérée par des actions utilisateurs au travers d'un menu.

Enfin, vous retrouverez davantage d'explications sur les méthodes/classes (contrats, algorithmes) dans le code, grâce à la mise en place du squelette.



B. Dessin précis de la mémoire pour le jeu d'essai suivant : (Structure données.pdf sur GitHub)

(Volontairement CollectionTrajet a été masqué pour exprimer son encapsulation et montrer que le système n'en est pas dépendant et ne fait que l'utiliser.)



C. Listing des classes définies dans notre application

- Classe Catalogue (utilise Trajet.h, CollectionTrajet.h):
  - Interface : Catalogue.h - Réalisation : Catalogue.cpp
- Classe Trajet (classe abstraite) :
  - Interface : Trajet.h – Pas de réalisation puisque classe abstraite
- Classe TrajetSimple (utilise Trajet.h):
  - Interface : TrajetSimple.h – Réalisation : TrajetSimple.cpp
- Classe TrajetCompose (utilise Trajet.h et CollectionTrajet.h) :
  - Interface : TrajetCompose.h – Réalisation : TrajetCompose.cpp
- Classe CollectionTrajet (structure de données, utilise Trajet.h) :
  - Interface : CollectionTrajet.h – Réalisation : CollectionTrajet.cpp

La suite de la question est réalisée, conformément à ce qui est demandé dans l'énoncé, dans l'entête de chaque interface/réalisation afin d'être directement lié au contenu.

D. Conclusion

Les problèmes marquants que nous avons rencontrés sont notamment liés aux pointeurs. En effet, durant notre conception nous imaginions être en droit d'utiliser les types tel que <string> ou encore <vector>, mais quand nous avons dû passer aux char * et aux tableaux de pointeurs sur Trajet cela n'a pas été de toute simplicité. Des petits moments sur valgrind et puis tout était de nouveau prêt à fonctionner comme nous l'entendions.
D'autre part, nous avons aussi eu quelques problèmes avec le second algorithme, qui est encore aujourd'hui un obstacle puisque nous avons tenté de le réaliser via plusieurs méthodes (backtracking, récursivité, brute force avec double boucle).
Cependant nous ne sommes pas parvenus à avoir des résultats satisfaisants et nous parvenions toujours à trouver des cas qui donnaient du fil à retordre à notre réalisation. C'est pourquoi nous avons fait le choix de ne pas l'implémenter dans la version du rendu afin d'offrir la possibilité au client d'avoir une version stable de notre création et ne pas avoir à se soucier d'une potentielle méthode problématique.
Les axes d'évolution sont donc les suivants :

- Terminer cet algorithme de recherche avancé afin d'être un peu plus proche d'une utilisation réaliste de notre système pour un utilisateur.
- Etoffer les vérifications afin d'empêcher l'insertion de doublon pour les trajets composés.
- Avoir une interface utilisateur un peu plus évoluée qu'un simple terminal, ici, ce serait simplement une amélioration esthétique par l'utilisation de librairies permettant de faire des petites applications en C++.
- Enfin, passer tous ces char * et tableaux de pointeurs en String et Vector afin de permettre une maintenance de notre application dans l'avenir bien plus naturelle et en concordance avec ce qui se fait aujourd'hui plutôt que de se restreindre à des méthodes moins intuitives.