**TP C++ n°2 : Héritage – Polymorphisme**

Constitution de voyages à partir d’un catalogue de trajets

1. Description des classes de l’application et du graphe d’héritage

Les différentes classes de notre application peuvent être catégorisées en deux sous-groupes. Il y a d’une part les objets que nous utilisons afin de créer le catalogue. Cela inclut la classe Catalogue, la classe Liste qui est utilisée par le catalogue, et la classe Maillon qui est utilisée par la liste. Et d’autre part nous avons les classes qui permettent la gestion des trajets, c’est-à-dire les classes Trajet, TrajetSimple et TrajetComposé. Les classes TrajetSimple et TrajetComposé héritent de Trajet. Nous aurions pu nous en passer, mais ce choix était assez logique d’un point de vue sémantique et il facilite grandement la compréhension de l’architecture de notre application. Enfin, on peut noter que pour ne pas créer une nouvelle structure de données, les trajets composés utilisent la classe Liste.

Le graphe d’héritage décrit dans le paragraphe ci-dessus est à retrouver en annexe.

**Catalogue**

Le catalogue contient une liste triée de trajets. C’est l’objet avec lequel l’utilisateur interagit directement lorsqu’il rentre ses instructions dans l’interface.

**Liste**

Une liste est une collection ordonnée de maillons, structurés sous la forme d’une liste simplement chaînée. La liste contient un pointeur sur le premier maillon ainsi qu’un pointeur sur le dernier maillon. Elle contient aussi un booléen qui indique si la liste des « propriétaire » des trajets stockés.

Nous utilisons des listes pour stocker des trajets : pour les trajets du catalogue mais aussi pour les trajets dont est constitué un trajet composé. Enfin, la recherche avancée de trajet est, elle aussi, réalisée avec des listes de trajets comme structure de données.

**Maillon**

Les maillons sont les objets dont sont constituées nos listes simplement chainées de trajets. Chaque maillon contient un pointeur sur un trajet, qui peut être simple ou composé, et un pointeur sur le prochain maillon. Le dernier maillon de la liste pointe sur le pointeur nul.

Les maillons ont aussi une variable booléenne. Cette variable est utilisée pour savoir si le maillon est « propriétaire » de son trajet. S’il l’est, il doit supprimer le trajet qui lui est associé lorsqu’il est supprimé. Nous utilisons cette fonction car nous copions seulement les pointeurs vers les trajets mais nous ne voulons pas tout supprimer lorsque l’on supprime la copie.

**Trajet**

Un trajet est constitué d’une ville de départ et d’une ville d’arrivée. C’est la classe dont héritent les trajets simples et composés.

**TrajetSimple**

La classe TrajetSimple hérite de la classe trajet. Néanmoins, un trajet simple stocke aussi le moyen de transport utilisé, en plus de la ville de départ et de celle d’arrivée.

**TrajetCompose**

La classe TrajetCompose hérite de la classe trajet. Un trajet composé et contient un pointeur sur une liste de trajets ainsi que la ville de départ et celle d’arrivée de cette liste de trajets.

C’est à l’utilisateur de faire en sorte que la ville de départ et celle d’arrivée soient cohérentes avec la liste de trajets fournis. C’est aussi à lui de faire en sorte que la liste elle-même soit cohérente, c’est-à-dire que l’arrivée d’un trajet soit le départ du suivant.

1. Description de la structure de données utilisée pour gérer les trajets

Pour gérer les trajets, nous avons utilisés une liste simplement chaînée. C’est une liste composée d’un pointeur sur le premier maillon. Ensuite, chaque maillon contient un pointeur sur un trajet et un pointeur sur le prochain maillon. Le pointeur sur le prochain maillon du dernier maillon de la liste est le pointeur nul.

Pour parcourir cette liste, il faut commencer au premier maillon puis récupérer le pointeur sur le prochain maillon, récupérer ce maillon, et ainsi de suite jusqu’à ce que le pointeur sur le prochain maillon soit le pointeur nul.

La liste comporte aussi un pointeur sur le dernier maillon. En effet, cela permet d’ajouter des maillons en queue de liste plus facilement et plus rapidement. Cela nous est très utile, notamment dans la recherche avancée où cette fonction est grandement utilisée.

**Exemple**

En annexe, nous avons représenté la mémoire pour un jeu d’essai fourni. On peut voir les différentes classes de notre application dans un contexte réel. On voit que le catalogue est dans la pile, c’est avec ce catalogue que toutes les opérations sont effectuées (ajout de trajets, recherche, etc…). On voit aussi que tous les objets et variables sont dans le tas, ce n’est pas étonnant car tous les objets sont initialisés en mémoire dynamiquement en utilisant le mot clé « new ».

Ce schéma permet aussi d’illustrer plus visuellement la structure de la classe Liste, qui est utilisée dans le catalogue mais aussi pour les trajets composés. On voit l’enchaînement de maillons dans lesquels sont stockés les trajets TS1, TC2 et TS3 et les pointeurs entre les maillons. De plus, on voit les pointeurs de la liste qui pointent sur le premier et le dernier maillon. Enfin, on peut constater que les trajets composés contiennent eux aussi une liste pour stocker les trajets qui les composent.

1. Conclusion

Problèmes marquants

Durant ce projet, nous avons rencontrés certains problèmes. Tout d’abord, lors de l’analyse du cahier des charges et de la création du graphe d’héritage, nous avons dû faire certains choix. Le premier choix correspond à celui de la structure de l’héritage que nous voulions implémenter : nous avions pensés à créer seulement deux classes, trajet (qui impliquerait un trajet simple) et trajet composé. Néanmoins, cette solution n’était pas la meilleure car elle posait des problèmes d’attributs et il n’était pas évident qu’un trajet composé héritait de trajet simple. La logique voulait que les trajets simples et composés soient tous les deux des types de trajets, c’est pourquoi nous avons créé une classe mère Trajet.

Ensuite, il a fallu faire un choix pour le type de structure de données que nous voulions utiliser dans notre liste. Nous n’avions pas de contraintes fortes car la liste est parcourue entièrement à chaque fois que l’on veut l’afficher mais aussi lorsque l’on recherche un élément car on cherche tous les trajets qui ont certaines caractéristiques. Nous hésitions entre un tableau simple et une liste chaînée mais nous avons choisi la liste chaînée car il était plus facile de la rendre dynamique pour ajouter des trajets.

Enfin, nous avons voulu implémenter la recherche avancée faite sans utiliser de nouvelle structure de données : nous voulions utiliser la liste simplement chaînée. Nous avions aussi pour objectif d’effectuer une recherche avancée qui évite les boucles.

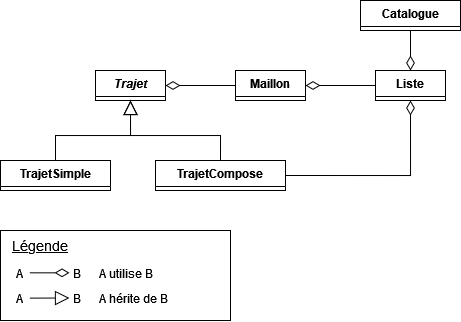
Axes d’évolution

Pour conclure, nous pensons avoir réussi à mettre en œuvre le cahier des charges qui nous a été donné. Malgré tout, il reste quand même des possibilités d’évolution et d’amélioration pour ce projet.

On peut penser à la classe TrajetCompose qui n’assure pas automatiquement sa cohérence au niveau de la liste de trajets et des villes de départ et d’arrivée. Pour ce faire, nous pourrions coder un constructeur qui prendrait en paramètre une liste et qui initialiserait les villes sans intervention de l’utilisateur. Un autre axe d’amélioration serait l’interface utilisateur qui est pour le moins austère et qui pourrait être rendue plus agréable.

Pour finir, la recherche avancée pourrait être plus optimisée. Il y a effectivement, beaucoup de constructions et destructions de listes et de trajets composés, ce qui impacterait les performances s’il y avait beaucoup de trajets. Néanmoins, nous n’avons pas poussé l’étude de ce problème plus loin.

1. Annexe

Graphe d’héritage de l’application

Dessin de la mémoire pour le jeu d’essai fourni

