2020 –
SEMESTRE 1

C++ - Compte rendu du TP2

INSA | INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

Stefania Curila - Daniel Blasko
B3301
3IF3

1 DESCRIPTION DE L'ENSEMBLE DES CLASSES DE LA SOLUTION LOGICIELLE

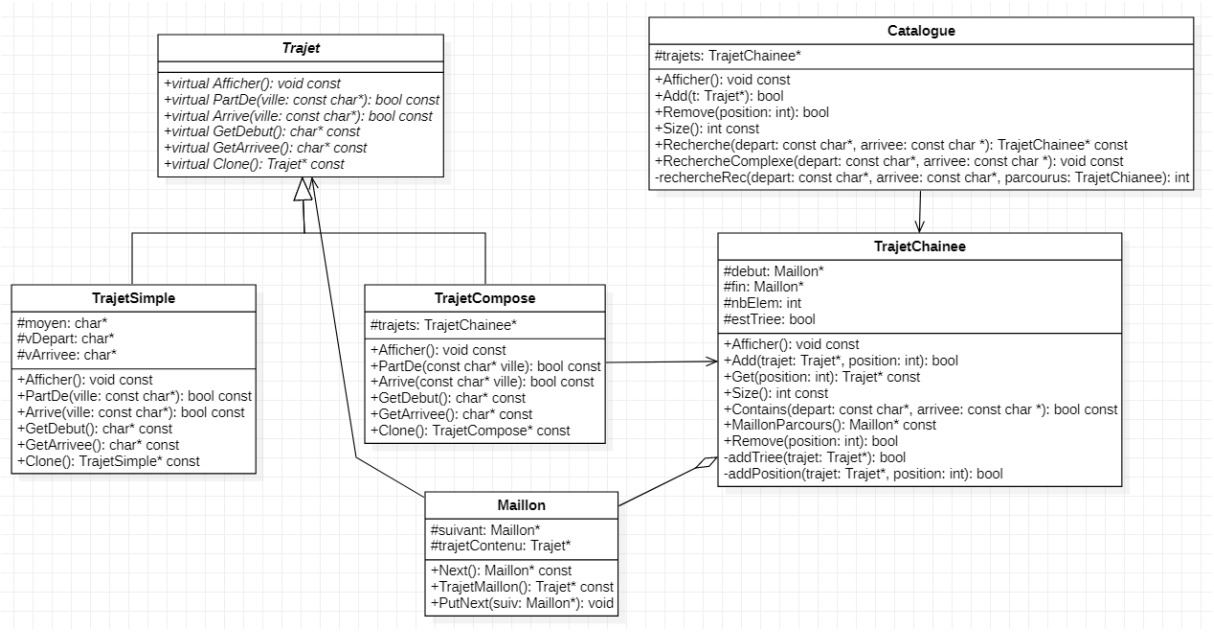

Figure 1 – Diagramme de classes de notre application

Les classes que nous avons choisies pour implémenter l'application sont les suivantes :

- ***Trajet*** : classe abstraite de base de laquelle dérivent les classes *TrajetSimple* et *TrajetComposee*. Ainsi, on a déclaré des méthodes virtuelles publiques qui définissent les comportements communs à tous les trajets et qui vont être définies dans chacune des classes dérivées avec leur comportement afférent.
- ***TrajetSimple*** : classe dérivée de la classe de base *Traje*t (héritage public). Cette classe représente un trajet direct d'une ville à une autre.
- ***TrajetCompose*** : classe dérivée de la classe de base *Trajet* (héritage public). Cette classe représente un trajet composé d'une ville à une autre, plus précisément un enchaînement ordonné de trajets simples.
- ***Maillon*** : définit une case de la liste chaînée *TrajetChainee*. Les attributs de la classe sont un pointeur vers le maillon suivant et un pointeur vers un trajet.
- ***TrajetChainee*** : représente une liste chaînée avec des pointeurs vers les maillons qui correspond au début et à la fin d'un trajet. Ce type d'objet sera utilisé dans le cadre d'un trajet composé formé d'un enchaînement de trajets simples et dans le cadre du catalogue qui utilise un enchaînement de trajets. Elle est décrite sur la page suivante.
- ***Catalogue*** : classe qui contient les différents trajets ordonnés et qui permet de les manipuler. Son attribut est un pointeur qui pointe vers un *TrajetChainee*.

2 DESCRIPTION ET ILLUSTRATION DE LA STRUCTURE DE DONNEES MISE EN PLACE POUR GERER LES LISTES DE TRAJETS

Pour manipuler des listes ordonnées d'objets de type abstrait Trajet, nous avons décidé d'implémenter une liste chaînée. En effet, l'étude du sujet nous a fait comprendre que les principales manipulations sur ces listes seront l'insertion et le parcours – cette structure de données était donc particulièrement adaptée à nos besoins. Elle ne manipule que les comportements (méthodes) du type abstrait Trajet, et donc les comportements communs aux trajets simples et composés héritant de cette classe.

Pour faciliter les opérations d'insertion en fin de liste, nous avons choisi de l'implémenter sous la forme d'une liste chainée avec un pointeur vers le début et la fin de la liste. Chaque maillon de la liste est le support d'une donnée de type pointeur vers un Trajet, et contient également un pointeur vers le maillon suivant.

Enfin, nous avons pris en compte notre possibilité de décider de la règle ordonnant la liste utilisée dans le catalogue. Pour simplifier les opérations de recherche, nous souhaitions pouvoir ordonner la liste chaînée sur la ville de départ des trajets, en ordre alphabétique croissant, dans ce cas. Souhaitant tout de même pouvoir également garder un ordre chronologique selon l'insertion (dans le cas d'utilisation des trajets composés), notre structure de données sait s'adapter aux deux besoins : en fonction de son attribut booléen *estTriee*, elle est triée selon l'ordre d'insertion si la valeur de ce dernier est fausse, et alphabétiquement sur la ville de départ si l'attribut vaut vrai.

Voici un diagramme illustrant la représentation de la structure de données en mémoire pour le cas spécifique précisé dans l'énoncé :

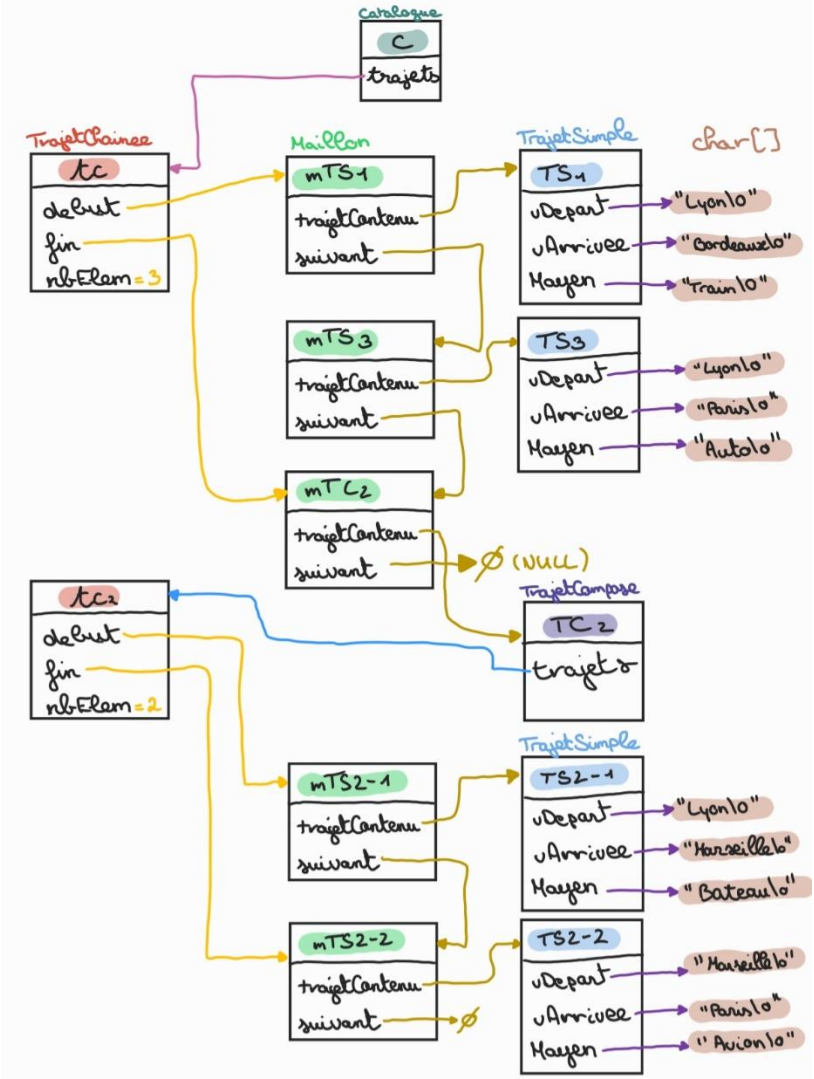

Figure 2 - schéma de la mémoire pour notre structure de données dans l'exemple de l'énoncé.

3 CONCLUSION ET RETROSPECTION SUR LE TRAVAIL REALISE

3.1 PRINCIPAUX PROBLEMES RENCONTRES ET COMMENT NOUS LES AVONS RESOLUS

Lors du développement de la solution logicielle, nous avons évidemment fait face à plusieurs problèmes. Certains ont demandé plus de recherche et d'approfondissement afin d'être résolus.

Par exemple, afin que notre méthode de recherche simple du catalogue puisse renvoyer une liste chainée en résultat (pour faciliter l'ajout de fonctionnalités à l'avenir), il fallait rendre possible la copie de trajets simples ou composés. Cela sous-entend aussi d'implémenter le constructeur de copie de la liste chainée, étant donné qu'un trajet composé contient une liste chainée. Nous avons alors fait face à un problème : notre application ne manipule que des objets de type Trajet, que ce soit dans une liste chainée ou le catalogue. De ce fait, nous ne pouvions pas juste appeler le constructeur de copie de l'objet de type Trajet à copier : cela appellerait le constructeur de copie de la classe Trajet, et n'effectuerait donc pas la copie correspondant au type spécialisé de l'objet, qui est très différente pour un trajet simple ou complexe ! Pour cette limite spécifique du polymorphisme en C++, nous avons donc recherché plusieurs et conclu que le standard dans l'industrie est d'ajouter une méthode « `virtual Trajet* Clone() const = 0;` » à la déclaration de la classe abstraite Trajet. Les deux classes en héritant devront obligatoirement l'implémenter. Chacune renvoie alors le résultat de l'appel de son propre constructeur de copie. Cette méthode virtuelle de Trajet nous permet ainsi d'appeler le bon constructeur de copie spécialisé sur n'importe quel objet référencé par un pointeur de type Trajet – nous pouvons ainsi copier un objet héritant de Trajet en profitant des avantages du polymorphisme, et ainsi, nous avons pu aussi construire à l'aide de cette méthode le constructeur de copie d'une liste chaînée.

Un autre problème auquel nous avons fait face a été la présence de quelques fuites de mémoire. Celles-ci ont été détectées à l'aide de Valgrind, qui signalait de la mémoire « definitely lost » car nous avions oublié d'appeler le destructeur de quelques éléments. Le fait que cette erreur ait été commise à un seul endroit bien spécifique a rendu la détection de la source du problème très difficile. Pour faciliter ce débogage, nous avons utilisé GDB et avons ajouté deux options de compilation à notre Makefile. Le flag « valgrind » compile avec l'option -d pour l'utilisation de Valgrind ou GDB, et l'option « debug » active l'option MAP pour le traçage des constructeurs et destructeurs. Combiner ces différents outils nous a permis de finalement trouver les pointeurs que l'on écrasait avant d'appeler delete dessus.

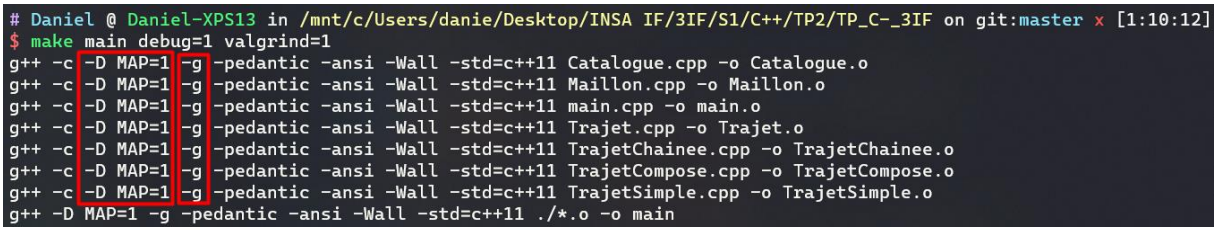

Figure 3 - Utilisation du Makefile avec les options de débogage afin de trouver les causes des fuites de mémoire.

Finalement, une autre problématique a été la création de l'algorithme de recherche avancée. En effet, la problématique faisait instinctivement penser au parcours d'un arbre, mais nous n'avions pas implémenté de structure de données correspond à un arbre. Nous avons alors décidé d'interpréter les données de notre liste chaînée (le catalogue) comme celles d'un arbre, où chaque ville est un nœud et chaque trajet représente une arête. On part de la racine, qui est le nœud de la ville de départ de la recherche, et on cherche tous ses fils. Nous cherchons donc toutes les villes où se finissent les trajets partant de la ville de départ dans le catalogue. Puis on recommence récursivement sur chaque fils, en mémorisant pour chaque appel les trajets déjà empruntés pour éviter de boucler sur des cycles. Cette démarche permet de parcourir l'arbre en profondeur jusqu'à trouver un chemin atteignant la ville d'arrivée de la recherche ou une feuille de l'arbre. Nous testons tous les parcours de l'arbre possibles, en comptant et affichant le nombre de chemins entre la ville de départ et d'arrivée, sans se bloquer dans des cycles.

3.2 Axes d'evolution pour la solution rendue

Notre solution n'est toutefois pas parfaite. Il serait possible d'imaginer des axes d'évolution pour celle-ci à l'avenir, notamment s'il devenait possible d'utiliser plus d'outils.

Par exemple, pour une utilisation réelle, elle représente pour l'instant peu d'intérêt : pour maintenir un réel catalogue de trajets, on aurait à laisser tourner le programme en permanence et espérer que notre machine ne s'éteindra pas. Un axe d'évolution utile serait donc d'introduire de la persistance des données, par exemple en écrivant les données du catalogue dans un fichier et en tenant ce fichier à jour.

De plus, il a été souligné dans l'énoncé que l'IHM n'est pas le point essentiel de l'application et l'expérience utilisateur n'est donc pas optimale dans notre solution. Il serait possible d'améliorer l'IHM de l'application, que ce soit en gérant encore mieux les erreurs de saisie de l'utilisateur, en la rendant plus intuitive et dynamique dans le terminal (via des librairies comme CTerminfo ou CGC par exemple) ou même dans l'idéal en implémentant une interface graphique (avec QT par exemple).

Aussi, il nous a été proposé de partir du principe qu'il n'y a pas de redondances dans le catalogue de trajets par exemple. Toutefois, un axe d'évolution possible serait aussi de ne pas faire autant confiance à l'utilisateur et d'effectuer ce type de vérifications tout de même, ou encore de faire les vérifications de l'égalité entre deux villes en ne tenant pas compte de la casse.

Enfin, dans l'optique de rendre l'application plus extensible et généralisable, il serait intéressant de rendre notre liste chaînée générique, ou même idéalement d'utiliser la classe Vector de C++ afin de gérer de tels types de données.