Justification de la conception de notre application

Nous avons choisi de faire hériter d’une classe mère Trajet les classes TrajetSimple et TrajetCompose car elles disposent de nombreuses similitudes : une ville de départ, une ville d’arrivée et la possibilité de l’afficher. De plus, un TrajetCompose n’est qu’une juxtaposition de plusieurs TrajetsSimple. Il sera donc possible de construire un TrajetCompose à partir d’une liste de TrajetSimple.

Nous avons décidé de garder Trajet comme classe abstraite car nous ne voulons pas qu’il soit possible d’instancier un objet de type Trajet : il doit s’agir obligatoirement d’un TrajetSimple ou d’un TrajetCompose.

Classe Trajet

La classe Trajet est une classe abstraite qui est la classe mère de trajetSimple et trajetCompose. Elle possède 2 attributs en protected qui sont des chaines de caractères représentant pour chaque trajet la ville de départ et la ville d’arrivée.

Au niveau des méthodes, getVilleDepart() et getVilleArrivee() seront utiles pour avoir un accès direct aux villes de départ et d’arrivées quand nous voudrons faire des comparaisons dans la classe catalogue. La méthode AfficherTrajet() devra également être redéfinie pour chaque type de trajet tout comme le constructeur.

Classe TrajetSimple : Trajet

Au niveau des attributs, nous avons uniquement rajouté une chaine de caractères afin de stocker le moyen de transport d’un TrajetSimple. Le constructeur d’un TrajetSimple nécessite donc une ville de départ, une ville d’arrivée (appel au constructeur de Trajet) et un moyen de transport. Nous avons fait le choix de ne pas mettre d’attributs par défaut car nous voulons que le trajet soit entièrement spécifié par l’utilisateur. La méthode AfficherTrajet() permet donc d’afficher sur le terminal sur une seule ligne les 3 attributs caractéristiques d’un TrajetSimple. Un constructeur de copie est également mis en place afin de pouvoir dupliquer des trajets par la suite. Nous avons surchargé l’opérateur == et != afin de pouvoir comparer 2 TrajetSimples : ils sont égaux si et seulement si leurs 3 attributs sont égaux.

Classe TrajetCompose : Trajet

Un TrajetCompose peut se définir comme une suite de TrajetsSimples (ou de TrajetsComposes mais dans notre application nous ne considérerons que des TrajetsSimples). Dans notre constructeur, nous avons besoin d’un tableau de TrajetSimples ainsi que du nombre de Trajets qui définira notre TrajetCompose. Aussi, TrajetCompose doit vérifier que l’enchainement de chaque TrajetSimple est cohérent : la ville de départ du prochain est la ville d’arrivée du courant. Cette condition sera vérifiée automatiquement lors de la saisie. Nous avons surchargé l’opérateur == afin de pouvoir comparer des TrajetsComposes, par exemple pour vérifier qu’un TrajetCompose n’est pas déjà dans le catalogue. 2 TrajetsComposes sont égaux si et seulement si chaque Trajet Simple sont égaux 2 à 2, qu’ils sont dans le même ordre et qu’il y a le même nombre de TrajetsSimples. La méthode AfficherTrajet() parcours chaque TrajetSimple définissant le TrajetCompose et réutilise l’affichage d’un TrajetSimple. Le destructeur s’occupe de libérer l’espace mémoire réservé à listeTrajetSimple.

Classe Catalogue

Le catalogue a pour objectif de gérer l’ensemble des trajets, c’est-à-dire l’ajout de trajets, l’affichage des trajets contenus, ou encore la recherche de tous les trajets pour se rendre d’une ville A vers une ville B. Au niveau des attributs, nous avons choisi de sauvegarder les trajets ajoutés sous la forme d’un tableau dynamique de pointeurs de Trajets car la classe Trajet est abstraite, nous ne pouvons donc pas les sauvegarder directement dans un tableau. Comme le catalogue est dynamique, nous avons besoin des attributs nbTrajetsMax (nombre de trajets maximum pouvant être contenu dans le catalogue avec possibilité d’augmentation) et nbTrajetsAct (nombre de trajets actuels) qui vont nous permettre de savoir à quel moment il faudra augmenter la taille du catalogue.

Lors de l’instanciation d’un Catalogue, celui-ci peut contenir par défaut un seul trajet.

L’affichage du Catalogue se fait grâce à la méthode AfficherCatalogue(), elle parcourt tous les trajets présents et appelle leur méthode AfficherTrajet() respective.

La méthode Ajouter() qui prend un pointeur de trajet en paramètre s’occupe donc de rajouter le trajet voulu dans le catalogue. Si le trajet est déjà présent dans le catalogue, il ne sera pas ajouté. Elle renvoie une valeur de retour pour indiquer si l’ajout a pu se faire ou non.

La recherche de trajet est double : on peut faire une rechercheSimple() afin de trouver uniquement les trajets qui ont pour ville de départ et d’arrivée celles passées en paramètre. Nous pouvons aussi faire une rechercheAvancee() : avec pour paramètres la ville de départ et la ville d’arrivée, tous les Trajets et enchainements de Trajets présents dans le catalogue seront testés. Toutes les combinaisons possibles de Trajets vérifiant la ville de départ et d’arrivée seront affichées, pourvu que l’on utilise qu’une seule fois chaque Trajet. Cette méthode appelle elle-même la méthode RechercheRécursive() qui garde en paramètre les villes de départ et d’arrivée courantes pour enchainer les trajets uniquement cohérents. C’est donc une méthode récursive avec un tableau d’entiers utilisé afin de marquer les trajets déjà parcourus (algorithme de « backtracking »).

Le destructeur de Catalogue permet de libérer la mémoire réservée au tableau de pointeurs de Trajets ainsi que d’appeler les destructeurs de chaque Trajet contenus dans le Catalogue.

L’implémentation du menu se fait au niveau de la classe TestMain. Notre menu se présente sous la forme d’une boucle while avec une valeur de sortie du programme ainsi que d’autres valeurs pour manipuler le catalogue (ajout de trajets, affichage …). La particularité du menu se trouve dans la saisie d’un TrajetCompose car une 2ème boucle while est présente afin que l’utilisateur continue à ajouter des TrajetSimples dans son TrajetCompose à sa guise. L’utilisateur ne peut pas saisir de TrajetCompose incohérents au niveau de l’enchainement des villes car la ville d’arrivée du trajet précédent est directement recopiée dans la ville de départ du trajet suivant.