Alexis LE CONTE
Romain DE CLERCQ

Compte Rendu de TP

TP C++ n°2 : Héritage – Polymorphisme

1 TABLE DES MATIERES

2 DESCRIPTION DETAILLEE DES CLASSES

Un schéma vaut mieux que de longs discours :

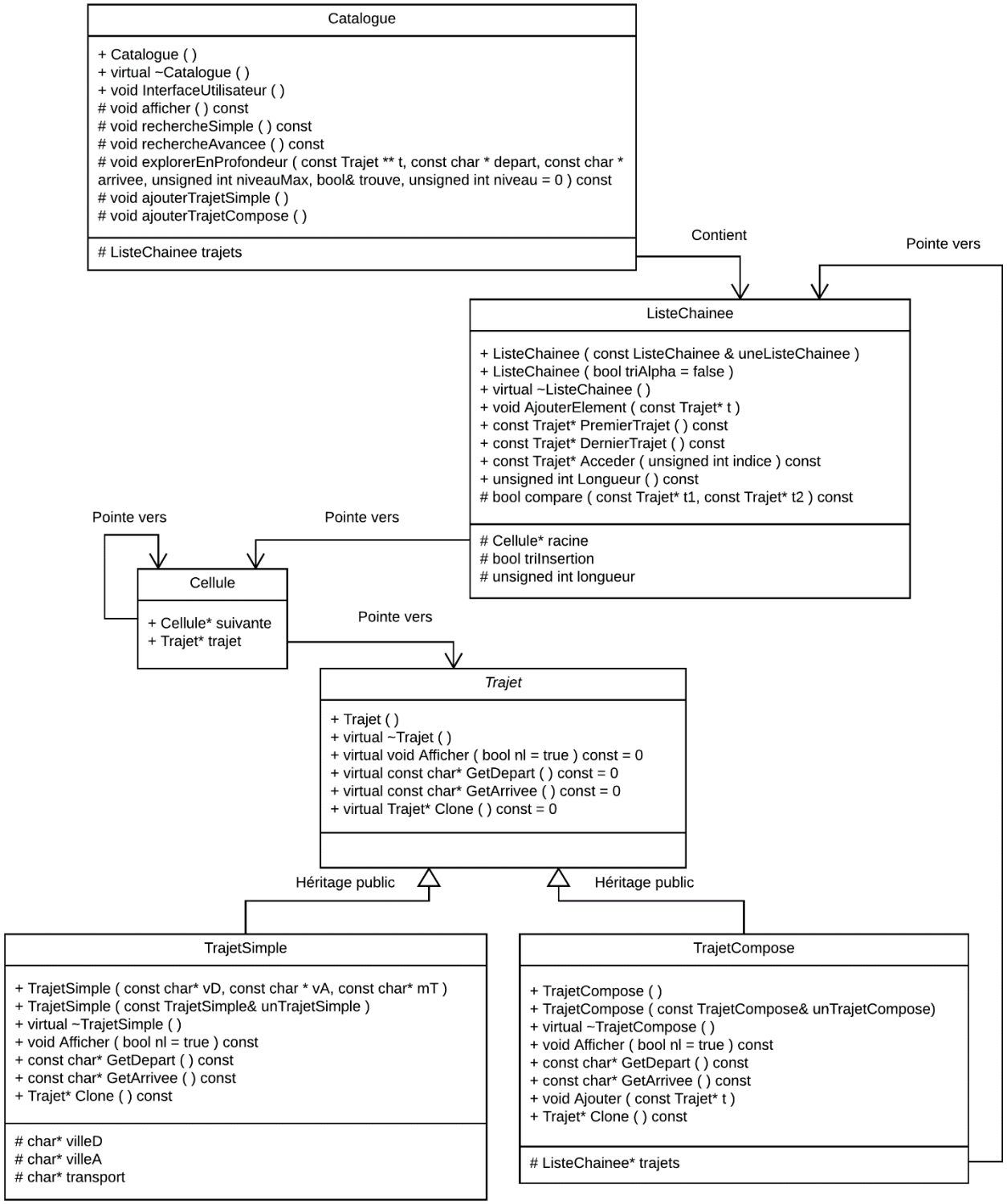


Diagramme de classes UML

La classe Catalogue est chargée du dialogue avec l'utilisateur. Elle propose une interface utilisateur en mode console qui propose à travers un menu l'accès à trois types de services :

- Afficher les trajets existants dans le catalogue
- Ajouter un trajet au catalogue
- Rechercher un parcours entre deux villes

Les autres classes représentent le modèle de données. Nous utilisons une liste chaînée pour stocker en mémoire les collections hétérogènes de trajets. Le catalogue possède une liste chaînée de trajets, de même que les instances de TrajetCompose. La liste chaînée stocke en fait des pointeurs vers des instances héritières de Trajet, qui ont été copiées en profondeur lors de leur ajout dans la liste chaînée : la liste chaînée est donc seule responsable de la gestion en mémoire des Trajet qu'elle stocke.

TrajetSimple et TrajetCompose héritent de Trajet, ce qui permet de les manipuler dans une collection hétérogène (ListeChainee) sans distinction de leur nature, au travers de l'interface offerte par la classe mère Trajet. Pour éviter la redondance d'informations, la classe Trajet n'a aucun attribut. En effet, si on avait mis par exemple les attributs villeD et villeA dans Trajet, cela aurait été redondant dans la classe TrajetCompose : on peut retrouver les villes de départ et d'arrivée à partir de la liste de trajets d'un TrajetCompose… De plus, mettre l'attribut transport dans la classe Trajet n'aurait aucun sens, car alors TrajetCompose hériterait de cet attribut, ce qui est contradictoire avec le fait qu'un TrajetCompose peut impliquer plusieurs modes de transports différents entre chaque étape.

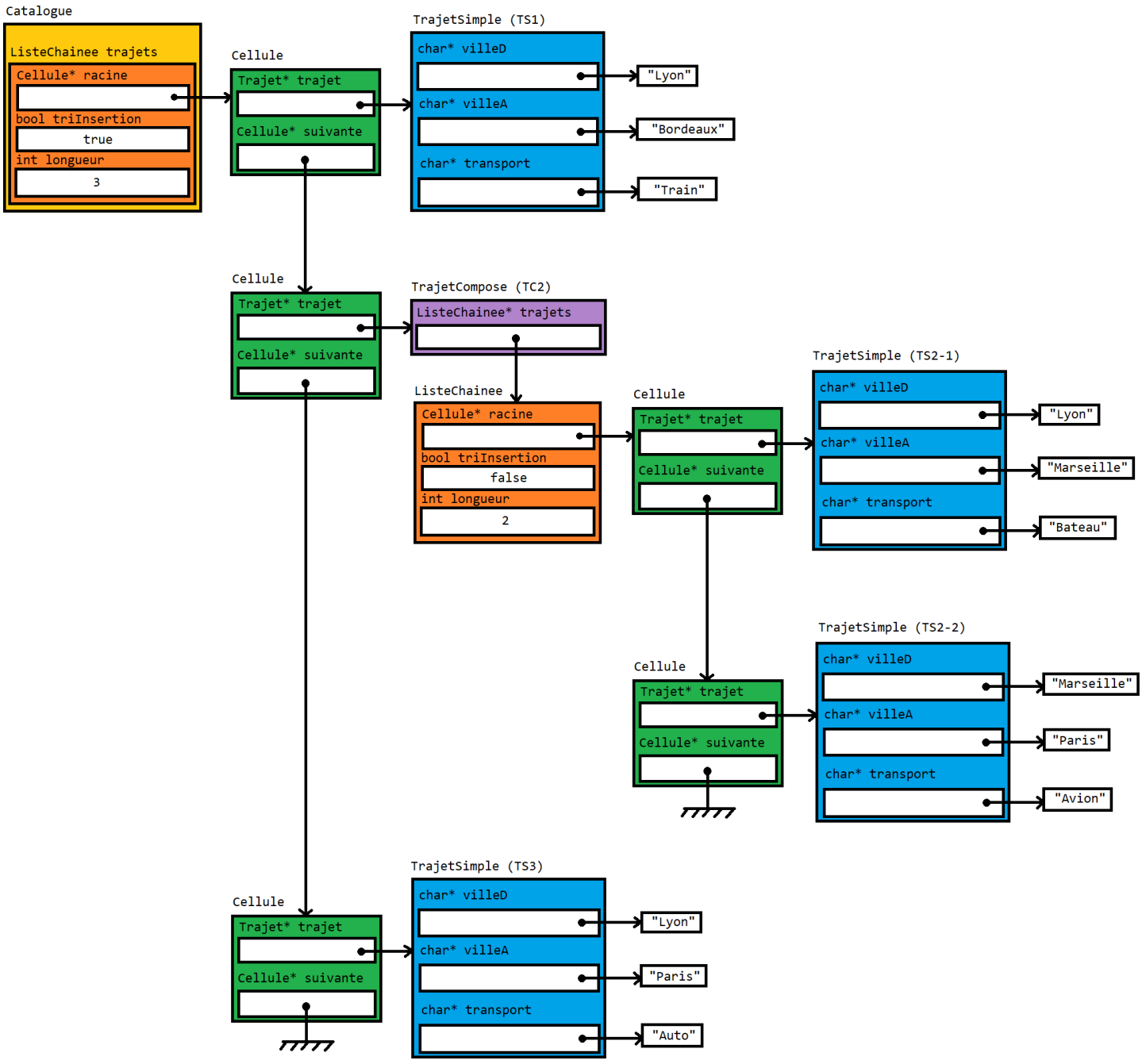
3 DESCRIPTION DETAILLEE DE LA STRUCTURE DE DONNEES

La structure de données utilisée est une liste chaînée qui stocke des pointeurs sur Trajet.

La même structure de données (classe ListeChainee) est utilisée dans la classe Catalogue et dans la classe TrajetCompose, mais avec la possibilité d'ordonner la collection hétérogène de façon différente : soit par ordre d'insertion pour respecter la sémantique d'un parcours (séquence ordonnée de trajets), soit par ordre alphabétique des noms de villes de départ et d'arrivée pour respecter la sémantique d'un catalogue ordonné (cet ordre distingue les lettres majuscules des lettres minuscules en se basant sur un alphabet ASCII). Dans ce dernier cas, on effectue un tri par insertion lors de l'ajout d'un Trajet dans la liste chaînée (correspond à la valeur « true » pour l'attribut triInsertion de la ListeChainee).

La liste chaînée est constituée de cellules (structure Cellule) simplement chaînées qui contiennent un pointeur vers un Trajet.

Comme nous l'avons dit plus haut, la liste gère ses propres instances des classes héritières de Trajet en effectuant des copies en profondeur des objets qu'on lui ajoute, ce qui évite les problèmes d'interférence du milieu extérieur sur ces objets (modification ou libération mémoire précoce, par exemple). C'est donc seulement à la destruction de la liste du catalogue que toute la mémoire sera libérée.



Dessin de la structure de données

4 LISTING DES CLASSES (CODE SOURCE)

4.1 CATALOGUE

4.1.1 Interface

```
/*************************************************************************
                          Catalogue  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//---------- Interface de la classe <Catalogue> (fichier Catalogue.h) ----
#if ! defined ( CATALOGUE_H )
#define CATALOGUE_H

//--------------------------------------------------- Interfaces utilisées
#include "ListeChainee.h"

//------------------------------------------------------------------------
// Rôle de la classe <Catalogue>
// La classe Catalogue fait le lien entre l'utilisateur final et le modèle
// de données. Elle contient une liste de Trajets (trajets)
// Elle propose trois types de services à l'utilisateur :
// - Affichage du catalogue : Affiche tous les trajets enregistrés dans le
//   catalogue
// - Ajout de trajets : Permet à l'utilisateur de saisir de nouveaux
//   trajets avec ou sans escales (TrajetSimple ou TrajetCompose) à
//   ajouter au catalogue
// - Recherche de parcours : Permet à l'utilisateur de rechercher les
//   parcours (séquences ordonnées de trajets) correspondant à un voyage
//   d'une ville donnée à une autre ville donnée
//------------------------------------------------------------------------

class Catalogue
{
//----------------------------------------------------------------- PUBLIC

public:
//---------------------------------------------------- Méthodes publiques
    void InterfaceUtilisateur ( );
    // Mode d'emploi :
    // Menu qui permet à l'utilisateur d'accéder aux services du catalogue
    // Le menu se base sur l' entrée / sortie standard
    // Contrat :
    // Rien

//------------------------------------------------- Surcharge d'opérateurs
    Catalogue& operator=(const Catalogue& unCatalogue);

//-------------------------------------------- Constructeurs - destructeur
    Catalogue ( );
    // Mode d'emploi :
    // Construction d'un catalogue de trajets vide
    // Contrat :
    // Rien

    virtual ~Catalogue ( );
    // Mode d'emploi :
    // Rien
```

```
    // Contrat :
    // Rien

//------------------------------------------------------------------ PRIVE

protected:
//----------------------------------------------------- Méthodes protégées
    void afficher ( ) const;
    // Mode d'emploi :
    // Service qui affiche le contenu du catalogue sur la sortie standard
    // Contrat :
    // Rien

    void rechercheSimple ( ) const;
    // Mode d'emploi :
    // Service qui permet de rechercher les parcours entre deux villes,
    // constitués d'un seul trajet (simple ou composé)
    // Contrat :
    // L'utilisateur rentre des données correctes dans l'entrée standard :
    // les chaines entrées doivent faire moins de 64 caractères et ne pas
    // contenir d'espaces

    void rechercheAvancee ( ) const;
    // Mode d'emploi :
    // Service qui permet de rechercher les parcours entre deux villes,
    // en effectuant des combinaisons de tous les trajets possibles
    // Contrat :
    // L'utilisateur rentre des données correctes dans l'entrée standard :
    // les chaines entrées doivent faire moins de 64 caractères et ne pas
    // contenir d'espaces

    void explorerEnProfondeur ( const Trajet ** t,
                                const char * depart,
                                const char * arrivee,
                                unsigned int niveauMax,
                                bool& trouve,
                                unsigned int niveau = 0 ) const;
    // Mode d'emploi :
    // Affiche toutes les combinaisons de trajets du catalogue qui ont
    // pour départ une ville de nom depart et pour arrivée une ville de
    // nom arrivee
    // t représente l'état courant de l'exploration
    // niveau représente la profondeur d'exploration courante
    // A la fin, trouve est à true si au moins une combinaison a été
    // trouvée, et false sinon
    // Contrat :
    // t doit être un tableau de taille niveauMax, initialisé avec des
    // pointeurs nuls
    // depart et arrivee doivent pointer vers des chaines de caractères
    // valides

    void ajouterTrajetSimple ( );
    // Mode d'emploi :
    // Service qui permet d'ajouter au catalogue un TrajetSimple entre
    // deux villes
    // Contrat :
    // L'utilisateur rentre des données correctes dans l'entrée standard :
    // les chaines entrées doivent faire moins de 64 caractères et ne pas
    // contenir d'espaces

    void ajouterTrajetCompose ( );
    // Mode d'emploi :
    // Service qui permet d'ajouter au catalogue un TrajetCompose entre
    // deux villes, avec ou sans étapes
```

```
    // Contrat :
    // L'utilisateur rentre des données correctes dans l'entrée standard :
    // les chaines entrées doivent faire moins de 64 caractères et ne pas
    // contenir d'espaces

//------------------------------------------------------ Attributs protégés
    ListeChainee trajets;
};

#endif // CATALOGUE_H
```

4.1.2 Réalisation

```
/*************************************************************************
                          Catalogue  -  description
                            -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//---- Réalisation de la classe <Catalogue> (fichier Catalogue.cpp) ------

//---------------------------------------------------------------- INCLUDE

//-------------------------------------------------------- Include système
using namespace std;
#include <iostream>
#include <cstring>

//------------------------------------------------------ Include personnel
#include "Catalogue.h"
#include "TrajetSimple.h"
#include "TrajetCompose.h"

//-------------------------------------------------------------- PUBLIC

//----------------------------------------------------- Méthodes publiques
void Catalogue::InterfaceUtilisateur ( )
// Algorithme :
// Propose une liste de services à l'utilisateur tant que celui-ci ne
// quitte pas l'application
{
    char choix = '0';
    cout << "Bienvenue dans le Catalogue de Trajets !" << endl;
    do
    {
        cout << endl;
        cout << "==================== MENU ====================" << endl;
        cout << "Que souhaitez-vous faire a present ?" << endl;
        cout << "0 : Afficher les trajets disponibles" << endl;
        cout << "1 : Ajouter un nouveau trajet simple" << endl;
        cout << "2 : Ajouter un nouveau trajet compose" << endl;
        cout << "3 : Recherche simple de trajet" << endl;
        cout << "4 : Recherche avancee de trajet" << endl;
        cout << "5 : Quitter l'application" << endl;
        cin >> choix;
        cout << "==============================================" << endl;
        switch (choix)
        {
            case '0':
                afficher();
                break;
            case '1':
```

```
                ajouterTrajetSimple();
                break;
            case '2':
                ajouterTrajetCompose();
                break;
            case '3':
                rechercheSimple();
                break;
            case '4':
                rechercheAvancee();
                break;
            case '5':
                cout << "Au revoir..." << endl;
                break;
            default:
                cout << "Choix invalide. Attendu : 0-1-2-3-4-5" << endl;
        }
    } while (choix != '5');
} //----- Fin de InterfaceUtilisateur

//------------------------------------------- Constructeurs - destructeur
Catalogue::Catalogue ( ) : trajets(true)
{
#ifdef MAP
    cout << "Appel au constructeur de <Catalogue>" << endl;
#endif
} //----- Fin de Catalogue


Catalogue::~Catalogue ( )
{
#ifdef MAP
    cout << "Appel au destructeur de <Catalogue>" << endl;
#endif
} //----- Fin de ~Catalogue


//------------------------------------------------------------------ PRIVE

//------------------------------------------------------ Méthodes protégées
void Catalogue::afficher ( ) const
{
    cout << "Le catalogue comporte actuellement " << trajets.Longueur();
    cout << " trajet" << (trajets.Longueur() > 1 ? "s." : ".") << endl;
    for(unsigned int i=0; i<trajets.Longueur(); ++i)
    {
        trajets.Acceder(i)->Afficher();
    }
} //----- Fin de afficher

void Catalogue::ajouterTrajetSimple ( )
{
    char villeD[64] = {};
    char villeA[64] = {};
    char transport[64] = {};
    cout << "Les entrees ne doivent pas comporter d'espaces !" << endl;
    cout << "Ville de depart : ";
    cin >> villeD;
    cout << "Ville d'arrivee : ";
    cin >> villeA;
    cout << "Moyen de transport : ";
    cin >> transport;
    TrajetSimple t(villeD, villeA, transport);
    trajets.AjouterElement(&t);
```

```
    cout << "Le trajet ";
    t.Afficher(false);
    cout << " a ete ajoute au catalogue." << endl;
} //----- Fin de ajouterTrajetSimple

void Catalogue::ajouterTrajetCompose ( )
// Algorithme :
// Entrées : ville de départ puis étapes et transports suivants
// Ajoute un premier TrajetSimple
// Puis continue à demander à l'utilisateur l'étape suivante
// jusqu'à ce qu'il valide le TrajetCompose en entrant "1"
{
    TrajetCompose tc;
    char villePrec[64] = {};
    char villeSuiv[64] = {};
    char transport[64] = {};
    cout << "Les entrees ne doivent pas comporter d'espaces !" << endl;
    cout << "Un trajet compose comporte au moins 2 villes" << endl;
    cout << "Entrez 1 pour valider la saisie du trajet" << endl;
    cout << "Ville de depart : ";
    cin >> villePrec;
    cout << "Etape suivante : ";
    cin >> villeSuiv;
    do
    {
        cout << "Moyen de transport : ";
        cin >> transport;
        TrajetSimple t(villePrec, villeSuiv, transport);
        tc.Ajouter(&t);
        strcpy(villePrec, villeSuiv);
        cout << "Etape suivante (ou 1) : ";
    } while(cin >> villeSuiv && strcmp(villeSuiv, "1") != 0);
    trajets.AjouterElement(&tc);
    cout << "Le trajet ";
    tc.Afficher(false);
    cout << " a ete ajoute au catalogue." << endl;
} //----- Fin de ajouterTrajetCompose

void Catalogue::rechercheSimple ( ) const
// Algorithme :
// Entrées : villes de départ et d'arrivée du parcours recherché
// Parcourt tous les trajets du catalogue et affiche ceux dont les villes
// de départ et d'arrivée correspondent à celles recherchées
{
    char villeD[64] = {};
    char villeA[64] = {};
    cout << "Ville de depart : ";
    cin >> villeD;
    cout << "Ville d'arrivee : ";
    cin >> villeA;
    bool trouve = false;
    for(unsigned int i=0; i<trajets.Longueur(); ++i)
    {
        const Trajet* t = trajets.Acceder(i);
        if(strcmp(t->GetDepart(), villeD) == 0 && strcmp(t->GetArrivee(), villeA) == 0)
        {
            cout << "+ Parcours : ";
            t->Afficher();
            trouve = true;
        }
    }
    if(!trouve)
    {
        cout << "Aucun trajet ne correspond a cette requete" << endl;
```

```
    }
} //----- Fin de rechercheSimple

void Catalogue::rechercheAvancee ( ) const
// Algorithme :
// Entrées : villes de départ et d'arrivée du parcours recherché
// Initialisation du tableau d'état pour la recherche en profondeur
// Algorithme récursif de parcours en profondeur (DFS)
{
    char villeD[64] = {};
    char villeA[64] = {};
    cout << "Ville de depart : ";
    cin >> villeD;
    cout << "Ville d'arrivee : ";
    cin >> villeA;

    const Trajet ** parcours = new const Trajet* [trajets.Longueur()];
    for(unsigned int i=0; i<trajets.Longueur(); ++i)
    {
        parcours[i] = 0;
    }
    bool trouve = false;
    explorerEnProfondeur(parcours, villeD, villeA, trajets.Longueur(), trouve);
    if(!trouve)
    {
        cout << "Aucun parcours ne correspond a cette requete" << endl;
    }
    delete[] parcours;
} //----- Fin de rechercheAvancee

void Catalogue::explorerEnProfondeur ( const Trajet ** parcours,
                                       const char * depart,
                                       const char * arrivee,
                                       unsigned int niveauMax,
                                       bool& trouve,
                                       unsigned int niveau ) const
// Algorithme :
// Recherche récursive en profondeur avec retour sur trace
{
    //Profondeur maximale atteinte, fin de l'exploration sur cette branche
    if(niveau == niveauMax)
    {
        return;
    }
    //Pour tous les trajets
    for(unsigned int i=0; i<niveauMax; ++i)
    {
        const Trajet * candidat = trajets.Acceder(i);
        //Dont le départ est l'arrivée du précédent trajet
        if(strcmp(candidat->GetDepart(), depart) != 0)
        {
            //Si ça n'est pas le cas, passe au trajet suivant
            continue;
        }
        //Et s'il n'a pas déjà été utilisé
        bool utilise = false;
        for(unsigned int j=0; j<niveau; ++j)
        {
            if(candidat == parcours[j])
            {
                utilise = true;
                break;
            }
        }
```

```
        if(!utilise)
        {
            //Valider le trajet
            parcours[niveau] = candidat;
            //Si l'arrivée du parcours est la destination choisie, l'afficher
            if(strcmp(candidat->GetArrivee(), arrivee) == 0)
            {
                trouve = true;
                cout << "+ Parcours : ";
                for(unsigned int j=0; j<=niveau; ++j)
                {
                    if(j > 0)
                    {
                        cout << " - ";
                    }
                    parcours[j]->Afficher(false);
                }
                cout << endl;
            }
            //Continuer l'exploration sur cette branche
            explorerEnProfondeur(parcours, candidat->GetArrivee(), arrivee, niveauMax, trouve,
                                                                                    niveau+1);
        }
    }
} //----- Fin de explorerEnProfondeur
```

4.2 LISTE CHAINEE

4.2.1 Interface

```
/*************************************************************************
                          ListeChainee  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//---- Interface de la classe <ListeChainee> (fichier ListeChainee.h) ----
#if ! defined ( LISTE_CHAINEE_H )
#define LISTE_CHAINEE_H

//------------------------------------------------------------------ Types
class Trajet;

// Cellule de la ListeChainee
struct Cellule {
    Trajet *trajet;        // Pointeur sur le Trajet associé à la Cellule
    Cellule *suivante;     // Chainage simple sur la cellule suivante
};

//-----------------------------------------------------------------------
// Rôle de la classe <ListeChainee>
// Une ListeChainee est une collection ordonnée de Trajets
// La ListeChainee contrôle l'allocation et la libération de mémoire
// dynamique de tous les Trajets qu'on lui rajoute, en réalisant une
// copie en profondeur de tous les trajets qu'on lui ajoute.
// Elle est constituée d'un chainage simple de Cellules qui possèdent
// un pointeur vers leur Trajet. Chaque Cellule est seule responsable du
// Trajet vers lequel elle possède un pointeur.
// Les Trajets peuvent être triés de deux manières différentes :
// - Par ordre d'ajout (Séquence ordonnée de Trajets)
```

```
//   pour respecter la sémantique d'un parcours composé de plusieurs
//   Trajets.
// - Par ordre alphabétique sur les noms de villes de départ, puis
//   d'arrivée, pour respecter la sémantique d'un catalogue.
// La ListeChainee permet d'ajouter un Trajet, et d'accéder à un Trajet :
// Accès au premier Trajet de la liste, au dernier Trajet, ou accès par
// indice.
// Attributs :
// - racine : la première cellule de la liste chainee
// - triInsertion : tri alphabétique si true, tri par ordre d'ajout sinon
// - longueur : nombre de Trajets dans la liste (nombre de cellules)
//------------------------------------------------------------------------

class ListeChainee
{
//---------------------------------------------------------------- PUBLIC

public:
//----------------------------------------------------- Méthodes publiques
    void AjouterElement ( const Trajet* t );
    // Mode d'emploi :
    // Ajoute le Trajet pointé par t à la liste de trajets, en faisant une
    // copie en profondeur du trajet à ajouter
    // Contrat :
    // t pointe vers un Trajet valide

    const Trajet* PremierTrajet ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur vers le premier Trajet de la liste chainee,
    // sans faire de copie en profondeur. C'est le Trajet associé à la
    // Cellule racine
    // Contrat :
    // La liste n'est pas vide (racine pointe vers une Cellule valide)

    const Trajet* DernierTrajet ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur vers le dernier Trajet de la liste chainee,
    // sans faire de copie en profondeur
    // Contrat :
    // La liste n'est pas vide (racine pointe vers une Cellule valide)

    const Trajet* Acceder ( unsigned int indice ) const;
    // Mode d'emploi :
    // Retourne un pointeur vers le Trajet à l'indice indice de la liste
    // chainee, sans faire de copie en profondeur
    // Retourne 0 si indice n'est pas un indice valide
    // Contrat :
    // indice est inférieur au nombre d'éléments de la liste

    unsigned int Longueur ( ) const;
    // Mode d'emploi :
    // Renvoie le nombre de Trajets que contient actuellement la liste
    // Renvoie 0 si la liste est vide
    // Contrat :
    // Rien

//------------------------------------------------- Surcharge d'opérateurs
    ListeChainee& operator=(const ListeChainee& uneListeChainee);

//-------------------------------------------- Constructeurs - destructeur
    ListeChainee ( const ListeChainee & uneListeChainee );
    // Mode d'emploi (constructeur de copie) :
    // Recopie en profondeur de uneListeChainee
    // Contrat :
```

```
    // Rien

    ListeChainee ( bool triAlpha = false );
    // Mode d'emploi :
    // Construction d'une ListeChainee ordonnée, vide
    // Le paramètre triAlpha spécifie si le tri doit se faire par ordre
    // alphabétique (si true) ou par ordre d'ajout (si false)
    // Contrat :
    // Rien

    virtual ~ListeChainee ( );
    // Mode d'emploi :
    // Libération de la mémoire allouée à l'ajout de chaque Trajet
    // Contrat :
    // Rien

//------------------------------------------------------------------ PRIVE

protected:
//----------------------------------------------------- Méthodes protégées
    bool compare ( const Trajet* t1, const Trajet* t2 ) const;
    // Mode d'emploi :
    // Compare les trajets pointés par t1 et t2 par ordre alphabétique sur
    // leurs noms de villes de départ, puis d'arrivée.
    // Contrat :
    // t1 et t2 pointent bien vers des Trajets valides

//----------------------------------------------------- Attributs protégés
    Cellule *racine;
    bool triInsertion;
    unsigned int longueur;
};

#endif // LISTE_CHAINEE_H
```

4.2.2 Réalisation

```
/*************************************************************************
                          ListeChainee  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//-- Réalisation de la classe <ListeChainee> (fichier ListeChainee.cpp) --

//---------------------------------------------------------------- INCLUDE

//-------------------------------------------------------- Include système
using namespace std;
#include <iostream>
#include <cstring>

//------------------------------------------------------ Include personnel
#include "ListeChainee.h"
#include "Trajet.h"

//------------------------------------------------------------------ PUBLIC

//----------------------------------------------------- Méthodes publiques
void ListeChainee::AjouterElement ( const Trajet* t )
```

```
// Algorithme :
// Allocation mémoire d'une nouvelle cellule et copie en profondeur de t
// Insertion par tri alphabétique, ou insertion en fin de liste
{
    //Allocation mémoire
    Cellule *cellule = new Cellule;
    cellule->suivante = 0;
    cellule->trajet = t->Clone();
    //Insertion en début si la liste est vide
    if(racine == 0)
    {
        racine = cellule;
    }
    //Insertion par tri alphabétique
    else if(triInsertion)
    {
        Cellule *courante = racine;
        //Cas particulier : insertion en début de liste
        if(!compare(racine->trajet, cellule->trajet))
        {
            cellule->suivante = racine;
            racine = cellule;
        }
        //Cas particulier : la liste n'a qu'un seul élément
        else if(racine->suivante == 0)
        {
            racine->suivante = cellule;
        }
        //Cas général : insertion en milieu ou en fin de liste
        else
        {
            const Trajet* t1 = courante->suivante->trajet;
            const Trajet* t2 = cellule->trajet;
            while(courante->suivante && compare(t1, t2))
            {
                courante = courante->suivante;
                if(courante->suivante)
                {
                    t1 = courante->suivante->trajet;
                }
            }
            cellule->suivante = courante->suivante;
            courante->suivante = cellule;
        }
    }
    //Insertion en fin de liste
    else
    {
        Cellule *courante = racine;
        while(courante->suivante)
        {
            courante = courante->suivante;
        }
        courante->suivante = cellule;
    }
    ++longueur;
} //----- Fin de AjouterElement

const Trajet* ListeChainee::PremierTrajet ( ) const
{
    return racine->trajet;
} //----- Fin de PremierTrajet

const Trajet* ListeChainee::DernierTrajet ( ) const
```

```
{
    return Acceder(longueur - 1);
} //----- Fin de DernierTrajet

const Trajet* ListeChainee::Acceder ( unsigned int indice ) const
// Algorithme :
// Parcours de la liste chainée jusqu'à atteindre l'élément recherché
{
    if(indice >= longueur)
    {
        return 0;
    }
    Cellule *courante = racine;
    for(unsigned int i=0; i<indice; ++i)
    {
        courante = courante->suivante;
    }
    return courante->trajet;
} //----- Fin de Acceder

unsigned int ListeChainee::Longueur ( ) const
{
    return longueur;
} //----- Fin de Longueur

//------------------------------------------------- Surcharge d'opérateurs

//-------------------------------------------- Constructeurs - destructeur
ListeChainee::ListeChainee ( const ListeChainee & uneListeChainee )
 : racine(0), triInsertion(uneListeChainee.triInsertion), longueur(0)
// Algorithme :
// Ajout par copie en profondeur de tous les éléments de uneListeChainee
{
#ifdef MAP
    cout << "Appel au constructeur de <ListeChainee>" << endl;
#endif
    for(unsigned int i=0; i<uneListeChainee.Longueur(); ++i)
    {
        AjouterElement(uneListeChainee.Acceder(i));
    }
} //----- Fin de ListeChainee

ListeChainee::ListeChainee ( bool triAlpha )
 : racine(0), triInsertion(triAlpha), longueur(0)
{
#ifdef MAP
    cout << "Appel au constructeur de <ListeChainee>" << endl;
#endif
} //----- Fin de ListeChainee


ListeChainee::~ListeChainee ( )
// Algorithme :
// Parcours de la liste chainée en libérant la mémoire au fur et à mesure
{
#ifdef MAP
    cout << "Appel au destructeur de <ListeChainee>" << endl;
#endif
    Cellule* courante = racine;
    while(courante)
    {
        delete courante->trajet;
        Cellule* suivante = courante->suivante;
        delete courante;
```

```
        courante = suivante;
    }
} //----- Fin de ~ListeChainee


//------------------------------------------------------------------ PRIVE

//------------------------------------------------------ Méthodes protégées
bool ListeChainee::compare ( const Trajet* t1, const Trajet* t2 ) const
// Algorithme :
// Comparaison par ordre alphabérique des villes de départ puis d'arrivée
{
    int cmpDep = strcmp(t1->GetDepart(), t2->GetDepart());
    int cmpArr = strcmp(t1->GetArrivee(), t2->GetArrivee());
    return cmpDep < 0 || (cmpDep == 0 && cmpArr < 0);
} // ----- Fin de compare
```

4.3 TRAJET

4.3.1 Interface

```
/*************************************************************************
                          Trajet  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//------- Interface de la classe <Trajet> (fichier Trajet.h) -------------
#if ! defined ( TRAJET_H )
#define TRAJET_H

//------------------------------------------------------------------------
// Rôle de la classe <Trajet>
// La classe Trajet représente un trajet orienté entre deux villes,
// le départ et l'arrivée, comprenant ou pas des étapes intermédiaires.
// C'est une classe abstraite qui sert à manipuler dans une collection
// hétérogène ses classes filles (TrajetSimple et TrajetCompose), sans
// distinction de leur nature
//------------------------------------------------------------------------

class Trajet
{
//------------------------------------------------------------------ PUBLIC

public:
//----------------------------------------------------- Méthodes publiques
    virtual void Afficher ( bool nl = true ) const = 0;
    // Mode d'emploi :
    // Affiche sur la sortie standard une description du trajet.
    // Le paramètre nl (nouvelle ligne) spécifie si l'affichage doit être
    // suivi d'un retour à la ligne. Il y a retour à la ligne si nl = true
    // Contrat :
    // Cette méthode doit être redéfinie dans les classes filles de Trajet

    virtual const char* GetDepart ( ) const = 0;
    // Mode d'emploi :
    // Retourne un pointeur sur une chaine de caractères représentant la
    // ville de départ pour ce trajet
    // Contrat :
    // Cette méthode doit être redéfinie dans les classes filles de Trajet
```

```
    // Le trajet doit posséder une ville de départ

    virtual const char* GetArrivee ( ) const = 0;
    // Mode d'emploi :
    // Retourne un pointeur sur une chaine de caractères représentant la
    // ville d'arrivée pour ce trajet
    // Contrat :
    // Cette méthode doit être redéfinie dans les classes filles de Trajet
    // Le trajet doit posséder une ville d'arrivée

    virtual Trajet* Clone ( ) const = 0;
    // Mode d'emploi :
    // Retourne une copie (en profondeur) du trajet
    // Contrat :
    // Cette méthode doit être redéfinie dans les classes filles de Trajet

//------------------------------------------------- Surcharge d'opérateurs
    Trajet& operator=(const Trajet& unTrajet);

//-------------------------------------------- Constructeurs - destructeur
    Trajet ( );
    // Mode d'emploi :
    // Constructeur appelé par les classes filles uniquement
    // Cette classe (abstraite) n'est pas instanciable
    // Contrat :
    // Rien

    virtual ~Trajet ( );
    // Mode d'emploi :
    // Rien
    // Contrat :
    // Rien

//------------------------------------------------------------------ PRIVE

protected:

};

#endif // TRAJET_H
```

4.3.2 Réalisation

```
/*************************************************************************
                          Trajet  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//------- Réalisation de la classe <Trajet> (fichier Trajet.cpp) ---------

//---------------------------------------------------------------- INCLUDE

//-------------------------------------------------------- Include système
using namespace std;
#include <iostream>

//------------------------------------------------------ Include personnel
#include "Trajet.h"

//------------------------------------------------------------------ PUBLIC
```

```
//------------------------------------------ Constructeurs - destructeur
Trajet::Trajet ( )
{
#ifdef MAP
    cout << "Appel au constructeur de <Trajet>" << endl;
#endif
} //----- Fin de Trajet


Trajet::~Trajet ( )
{
#ifdef MAP
    cout << "Appel au destructeur de <Trajet>" << endl;
#endif
} //----- Fin de ~Trajet
```

4.4 TRAJET SIMPLE

4.4.1 Interface

```
/*************************************************************************
                          TrajetSimple  -  description
                            -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//--- Interface de la classe <TrajetSimple> (fichier TrajetSimple.h) -----
#if ! defined ( TRAJET_SIMPLE_H )
#define TRAJET_SIMPLE_H

//------------------------------------------------- Interfaces utilisées
#include "Trajet.h"

//------------------------------------------------------------------------
// Rôle de la classe <TrajetSimple>
// TrajetSimple hérite publiquement de Trajet
// Un TrajetSimple décrit un trajet orienté entre deux villes, impliquant
// un moyen de transport, et ne comportant pas d'étape intermédiaire
// Un TrajetSimple comporte trois attributs sous forme de chaines de
// caractères : une ville de départ (villeD), une ville d'arrivée (villeA)
// et un moyen de transport (transport)
//------------------------------------------------------------------------

class TrajetSimple : public Trajet
{
//----------------------------------------------------------------- PUBLIC

public:
//-------------------------------------------------------- Méthodes publiques
    void Afficher ( bool nl = true ) const;
    // Mode d'emploi :
    // Affiche sur la sortie standard une description du trajet simple :
    // ville de départ, ville d'arrivée et moyen de transport utilisé.
    // Le paramètre nl (nouvelle ligne) spécifie si l'affichage doit être
    // suivi d'un retour à la ligne. Il y a retour à la ligne si nl = true
    // Contrat :
    // Rien

    const char* GetDepart ( ) const;
```

```
    // Mode d'emploi :
    // Retourne un pointeur sur la chaine de caractères représentant la
    // ville de départ, sans effectuer de copie en profondeur
    // Contrat :
    // Rien

    const char* GetArrivee ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur sur la chaine de caractères représentant la
    // ville d'arrivée, sans effectuer de copie en profondeur
    // Contrat :
    // Rien

    Trajet* Clone ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur sur une copie en profondeur de ce trajet
    // Attention, il y a allocation dynamique de mémoire lors de la copie
    // Contrat :
    // Rien

//------------------------------------------------- Surcharge d'opérateurs
    TrajetSimple& operator=(const TrajetSimple& unTrajetSimple);

//-------------------------------------------- Constructeurs - destructeur
    TrajetSimple ( const TrajetSimple & unTrajetSimple );
    // Mode d'emploi (constructeur de copie) :
    // Recopie en profondeur de unTrajetSimple
    // Contrat :
    // unTrajetSimple est un TrajetSimple correctement construit

    TrajetSimple ( const char* vD, const char * vA, const char* mT );
    // Mode d'emploi :
    // Recopie en profondeur les chaines de caractères dans les attributs
    // correspondants
    // vD est un pointeur sur une chaine représentant la ville de départ
    // vA est un pointeur sur une chaine représentant la ville d'arrivée
    // mT est un pointeur sur une chaine représentant le transport
    // Contrat :
    // Les chaines de caractère doivent être des chaines valides au sens
    // du C : elles se terminent par un caractère nul ('\0')

    virtual ~TrajetSimple ( );
    // Mode d'emploi :
    // Libère toute la mémoire allouée lors de la construction
    // Contrat :
    // Rien

//------------------------------------------------------------------ PRIVE

protected:
//----------------------------------------------------- Attributs protégés
    char* villeD;
    char* villeA;
    char* transport;
};

#endif // TRAJET_SIMPLE_H
```

4.4.2 Réalisation

```
/*************************************************************************
                          TrajetSimple  -  description
                             -------------------
  début      : 20/11/2017
```

```
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//-- Réalisation de la classe <TrajetSimple> (fichier TrajetSimple.cpp) --

//---------------------------------------------------------------- INCLUDE

//-------------------------------------------------------- Include système
using namespace std;
#include <iostream>
#include <cstring>

//------------------------------------------------------ Include personnel
#include "TrajetSimple.h"

//-------------------------------------------------------------- PUBLIC

//----------------------------------------------------- Méthodes publiques
const char* TrajetSimple::GetDepart ( ) const
{
    return villeD;
} // ----- Fin de GetDepart

const char* TrajetSimple::GetArrivee ( ) const
{
    return villeA;
} // ----- Fin de GetArrivee

void TrajetSimple::Afficher ( bool nl ) const
{
    cout << "De " << villeD << " a " << villeA << " en " << transport;
    if(nl)
    {
        cout << endl;
    }
} // ----- Fin de Afficher

Trajet* TrajetSimple::Clone ( ) const
// Algorithme :
// Allocation dynamique d'une copie du trajet en utilisant le constructeur
// de copie
{
    return new TrajetSimple(*this);
} // ----- Fin de Clone

//------------------------------------------- Constructeurs - destructeur
TrajetSimple::TrajetSimple ( const TrajetSimple & unTrajetSimple )
// Algorithme :
// Allocation en mémoire de chaque attribut du trajet avec la taille
// nécessaire pour réaliser une copie en profondeur des attributs de
// unTrajetSimple correspondants
// Puis copie en profondeur des attributs de unTrajetSimple
{
#ifdef MAP
    cout << "Appel au constructeur de copie de <TrajetSimple>" << endl;
#endif
    villeD = new char[strlen(unTrajetSimple.villeD)+1];
    villeA = new char[strlen(unTrajetSimple.villeA)+1];
    transport = new char[strlen(unTrajetSimple.transport)+1];
    strcpy(villeD, unTrajetSimple.villeD);
    strcpy(villeA, unTrajetSimple.villeA);
    strcpy(transport, unTrajetSimple.transport);
} //----- Fin de TrajetSimple (constructeur de copie)
```

```
TrajetSimple::TrajetSimple ( const char* vD, const char* vA, const char* mT )
// Algorithme :
// Allocation en mémoire de chaque attribut du trajet avec la taille
// nécessaire pour réaliser une copie en profondeur des paramètres
// correspondants
// Puis copie en profondeur des paramètres dans les attributs
{
#ifdef MAP
    cout << "Appel au constructeur de <TrajetSimple>" << endl;
#endif
    villeD = new char[strlen(vD)+1];
    villeA = new char[strlen(vA)+1];
    transport = new char[strlen(mT)+1];
    strcpy(villeD, vD);
    strcpy(villeA, vA);
    strcpy(transport, mT);
} //----- Fin de TrajetSimple

TrajetSimple::~TrajetSimple ( )
{
#ifdef MAP
    cout << "Appel au destructeur de <TrajetSimple>" << endl;
#endif
    delete[] villeD;
    delete[] villeA;
    delete[] transport;
} //----- Fin de ~TrajetSimple
```

4.5 TRAJET COMPOSE

4.5.1 Interface

```
/*************************************************************************
                          TrajetCompose  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//--- Interface de la classe <TrajetCompose> (fichier TrajetCompose.h) ---
#if ! defined ( TRAJET_COMPOSE_H )
#define TRAJET_COMPOSE_H

//------------------------------------------------- Interfaces utilisées
#include "Trajet.h"

//------------------------------------------------------------------ Types
class ListeChainee;

//------------------------------------------------------------------------
// Rôle de la classe <TrajetCompose>
// TrajetCompose hérite publiquement de Trajet
// Un TrajetCompose décrit un trajet orienté entre deux villes, comportant
// éventuellement des étapes intermédiaires.
// Un TrajetCompose comporte en pratique une liste ordonnée de Trajets
// (trajets). Cette liste doit être non vide.
// Les trajets de la liste doivent se suivre : l'arrivée de l'un doit
// être le départ du suivant.
//------------------------------------------------------------------------
```

```
class TrajetCompose : public Trajet
{
//------------------------------------------------------------------ PUBLIC

public:
//----------------------------------------------------- Méthodes publiques
    void Afficher ( bool nl = true ) const;
    // Mode d'emploi :
    // Affiche sur la sortie standard une description du trajet compose :
    // affiche la description de tous les sous-trajets de sa liste.
    // Le paramètre nl (nouvelle ligne) spécifie si l'affichage doit être
    // suivi d'un retour à la ligne. Il y a retour à la ligne si nl = true
    // Contrat :
    // La liste de trajets doit être non vide

    const char* GetDepart ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur sur une chaine de caractères représentant la
    // ville de départ, sans effectuer de copie en profondeur
    // La ville de départ est la ville de départ du premier sous-trajet
    // Contrat :
    // La liste de trajets doit être non vide

    const char* GetArrivee ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur sur une chaine de caractères représentant la
    // ville d'arrivée, sans effectuer de copie en profondeur
    // La ville d'arrivée est la ville d'arrivée du dernier sous-trajet
    // Contrat :
    // La liste de trajets doit être non vide

    Trajet* Clone ( ) const;
    // Mode d'emploi :
    // Retourne un pointeur sur une copie en profondeur de ce trajet
    // Attention, il y a allocation dynamique de mémoire lors de la copie
    // Contrat :
    // Rien

    void Ajouter ( const Trajet* t );
    // Mode d'emploi :
    // Ajoute un sous-trajet à la suite de la liste de Trajets
    // Le paramètre t est un pointeur sur le trajet à ajouter, il est
    // copié en profondeur.
    // Contrat :
    // Rien

//------------------------------------------------- Surcharge d'opérateurs
    TrajetCompose& operator=(const TrajetCompose& unTrajetCompose);

//-------------------------------------------- Constructeurs - destructeur
    TrajetCompose ( const TrajetCompose & unTrajetCompose );
    // Mode d'emploi (constructeur de copie) :
    // Recopie en profondeur de unTrajetCompose
    // Contrat :
    // Rien

    TrajetCompose ( );
    // Mode d'emploi :
    // Construit un TrajetCompose avec une liste de sous-trajets vide
    // Contrat :
    // Rien

    virtual ~TrajetCompose ( );
```

```
    // Mode d'emploi :
    // Libération de la mémoire allouée à la construction
    // Contrat :
    // Rien

//------------------------------------------------------------------ PRIVE

protected:
//----------------------------------------------------- Attributs protégés
    ListeChainee* trajets;
};

#endif // TRAJET_COMPOSE_H
```

4.5.2 Réalisation

```
/*************************************************************************
                          TrajetCompose  -  description
                            -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//- Réalisation de la classe <TrajetCompose> (fichier TrajetCompose.cpp) -

//------------------------------------------------------------------ INCLUDE

//-------------------------------------------------------- Include système
using namespace std;
#include <iostream>

//------------------------------------------------------ Include personnel
#include "TrajetCompose.h"
#include "ListeChainee.h"

//------------------------------------------------------------------ PUBLIC

//----------------------------------------------------- Méthodes publiques
void TrajetCompose::Afficher ( bool nl ) const
// Algorithme :
// Affiche tous les sous-trajets séparés par un tiret
{
    for(unsigned int i=0; i<trajets->Longueur(); ++i)
    {
        if(i > 0)
        {
            cout << " - ";
        }
        trajets->Acceder(i)->Afficher(false);
    }
    if(nl)
    {
        cout << endl;
    }
} //----- Fin de Afficher

const char* TrajetCompose::GetDepart ( ) const
{
    return trajets->PremierTrajet()->GetDepart();
} //----- Fin de GetDepart

const char* TrajetCompose::GetArrivee ( ) const
{
```

```
    return trajets->DernierTrajet()->GetArrivee();
} //----- Fin de GetArrivee

Trajet* TrajetCompose::Clone ( ) const
// Algorithme :
// Allocation dynamique d'une copie du trajet en utilisant le constructeur
// de copie
{
    return new TrajetCompose(*this);
} // ----- Fin de Clone

void TrajetCompose::Ajouter ( const Trajet *t )
{
    trajets->AjouterElement(t);
} //----- Fin de Ajouter

//------------------------------------------- Constructeurs - destructeur
TrajetCompose::TrajetCompose ( const TrajetCompose & unTrajetCompose )
// Algorithme :
// Recopie en profondeur de la liste de trajets
{
#ifdef MAP
    cout << "Appel au constructeur de copie de <TrajetCompose>" << endl;
#endif
    trajets = new ListeChainee(*unTrajetCompose.trajets);
} //----- Fin de TrajetCompose (constructeur de copie)

TrajetCompose::TrajetCompose ( )
{
#ifdef MAP
    cout << "Appel au constructeur de <TrajetCompose>" << endl;
#endif
    trajets = new ListeChainee;
} //----- Fin de TrajetCompose

TrajetCompose::~TrajetCompose ( )
{
#ifdef MAP
    cout << "Appel au destructeur de <TrajetCompose>" << endl;
#endif
    delete trajets;
} //----- Fin de ~TrajetCompose
```

4.6 Module Main

4.6.1 Interface

```
/*************************************************************************
                           Main  -  description
                             -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//--------- Interface du module <Main> (fichier Main.h) ------------------
#if ! defined ( MAIN_H )
#define MAIN_H

//------------------------------------------------------------------------
// Rôle du module <Main>
// Définit la fonction main, point d'entrée de l'application
```

```
//-------------------------------------------------------------------------

////////////////////////////////////////////////////////////////  PUBLIC
//------------------------------------------------- Fonctions publiques
int main ( );
// Mode d'emploi :
// Point d'entrée de l'application
// Contrat :
// Cette fonction ne doit pas être appelée, le système s'en charge

#endif // MAIN_H
```

4.6.2 Réalisation

```
/*************************************************************************
                          Main  -  description
                            -------------------
  début      : 20/11/2017
  copyright  : (C) 2017 par aleconte, rdeclercq
  e-mail     : alexis.le-conte@insa-lyon.fr, romain.de-clercq@insa-lyon.fr
*************************************************************************/

//--------- Réalisation du module <Main> (fichier Main.cpp) --------------

////////////////////////////////////////////////////////////////  INCLUDE
//------------------------------------------------------ Include personnel
#include "Main.h"
#include "Catalogue.h"

////////////////////////////////////////////////////////////////  PUBLIC
//------------------------------------------------- Fonctions publiques
int main ( )
// Algorithme :
// Lancement de l'interface utilisateur d'un catalogue vide
{
       Catalogue c;
       c.InterfaceUtilisateur();
       return 0;
} //----- fin de main
```

5 DIFFICULTES RENCONTREES ET AXES D'EVOLUTION

5.1 NOTRE STRUCTURE DE DONNEES N'EST PAS OPTIMALE POUR LA RECHERCHE

Par rapport à un tableau dynamique, les listes chaînées sont plutôt optimisées pour l'insertion et la suppression, mais pas pour la recherche. Nous accédons aux éléments de notre liste chaînée par indice, avec une complexité horrifiante de O(n) en comparaison d'une complexité de O(1) pour les tableaux dynamiques. Pour un parcours complet de notre collection, la complexité est de $O(n^2)$ contre O(n) pour un tableau… l'écart se creuse.

Nous n'utilisons pour le moment que des listes de trajets relativement courtes, cependant, si l'utilisateur venait à créer des trajets composés longs ou beaucoup de trajets dans le catalogue, le temps d'exécution deviendrait non négligeable. Notre algorithme de recherche avancé est en effet très coûteux en temps d'accès dans notre liste chaînée. On pourrait pour limiter ce problème rajouter un pointeur vers la fin de chaîne par exemple, pour avoir accès en temps constant au dernier trajet d'un TrajetCompose ou encore utiliser un chaînage double pour amortir le temps d'accès par indice, ou même changer de structure de données et

utiliser à la place un tableau dynamique pour accéder en temps constant à un Trajet à partir d'un indice. Mais cela pose comme toujours la question de l'optimisation de la mémoire : la mémoire dans un tableau doit être contiguë, et si on veut éviter de réallouer tout notre tableau à chaque ajout il faut se donner une politique de gestion de la mémoire qui évite de gaspiller trop de mémoire inutilement en allouant trop d'espace de réserve…

5.2 L'INTERFACE UTILISATEUR DEMANDE A ETRE AMELIOREE

Si l'utilisateur rentre un trajet erroné, ou bien qu'il souhaite en supprimer un car il n'est plus réalisable, il ne peut pas encore le faire. En effet, il n'existe pas de méthode pour modifier ou enlever un trajet du catalogue. Pourtant, notre structure de données actuelle (liste chaînée) résisterait très bien à la suppression ou à la modification d'éléments.

Actuellement, l'utilisateur peut rentrer n'importe quel moyen de transport. On pourrait par exemple souhaiter aller de Paris à Tokyo à dos de ~~Maranzana~~ coccinelle. Une solution simple serait de créer une énumération afin de n'autoriser que certains types de moyens de transport, mais limiterait les possibilités de choix de l'utilisateur et compliquerait l'utilisation de l'application.

Il n'est pour le moment pas possible de créer un trajet composé en sélectionnant des trajets déjà existants. Ce pourrait être une évolution possible de l'application qui rendrait l'ajout de trajets plus agréable et plus rapide.

L'interface est actuellement extrêmement simplifiée. Dans l'optique d'une application, nous pourrions réaliser une interface bien plus complète, agréable à voir et à utiliser.

Enfin, nous ne vérifions pas que l'utilisateur entre des données valides : c'est-à-dire des chaines de caractères sans espaces, et de moins de 64 caractères. En cas d'un usage non conforme, notre application est extrêmement vulnérable (buffer overflow).