INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

2016

Compte-rendu de TP : TransportSimulator2000

POO 3IF – TP2 – HERITAGE/POLYMORPHISME

ZIGGY VERGNE – LUCAS POISSE
3IF – GROUPE 4 – B3412

I. Description des classes et diagramme de classes

1. *Description des classes*

Le projet « TransportSimulator2000 » est organisé en différentes classes distinctes :

- Une première classe, la classe « Trajet », est une classe abstraite représentant un trajet (simple ou composé). L'utilisation des abstractions et de l'héritage a ainsi permis de rassembler dans cette même classe la gestion des noms des villes (ville de départ et d'arrivée) au sein d'un même module. L'interface de cette classe comporte en outre la définition d'une énumération, utilisée pour déclarer et définir le moyen de transport d'un trajet simple (voir point suivant).

- Les classes « TrajetComposé » et « TrajetSimple » sont directement dérivées de la classe Trajet. Elles se chargent d'implémenter les méthodes abstraites définies dans la classe mère et de gérer certaines caractéristiques qui leur sont spécifiques. On retrouve par exemple la gestion des moyens de transport dans le module correspondant à la classe « TrajetSimple », ou encore celle d'une structure de donnée de type liste chaînée comportant des trajets (simples ou composés) dans la classe « TrajetComposé » (voir classe « ListeTrajets »).

- La classe « Catalogue » se charge de regrouper au sein d'un même objet les différents trajets créés par l'utilisateur, également sous la forme d'un attribut dont la structure de données est de type « «ListeTrajets ». Plusieurs services, directement liés à ceux de la liste de trajets, sont proposés au sein du catalogue, dont la possibilité d'ajouter un trajet simple ou composé (commande 1 de l'interface), d'afficher le catalogue (commande 2 de l'interface), de rechercher un trajet d'une ville A à une ville B en mode « recherche simple[1] » (commande 3 de l'interface), ou en mode « recherche complexe[2] » (commande 4 de l'interface).

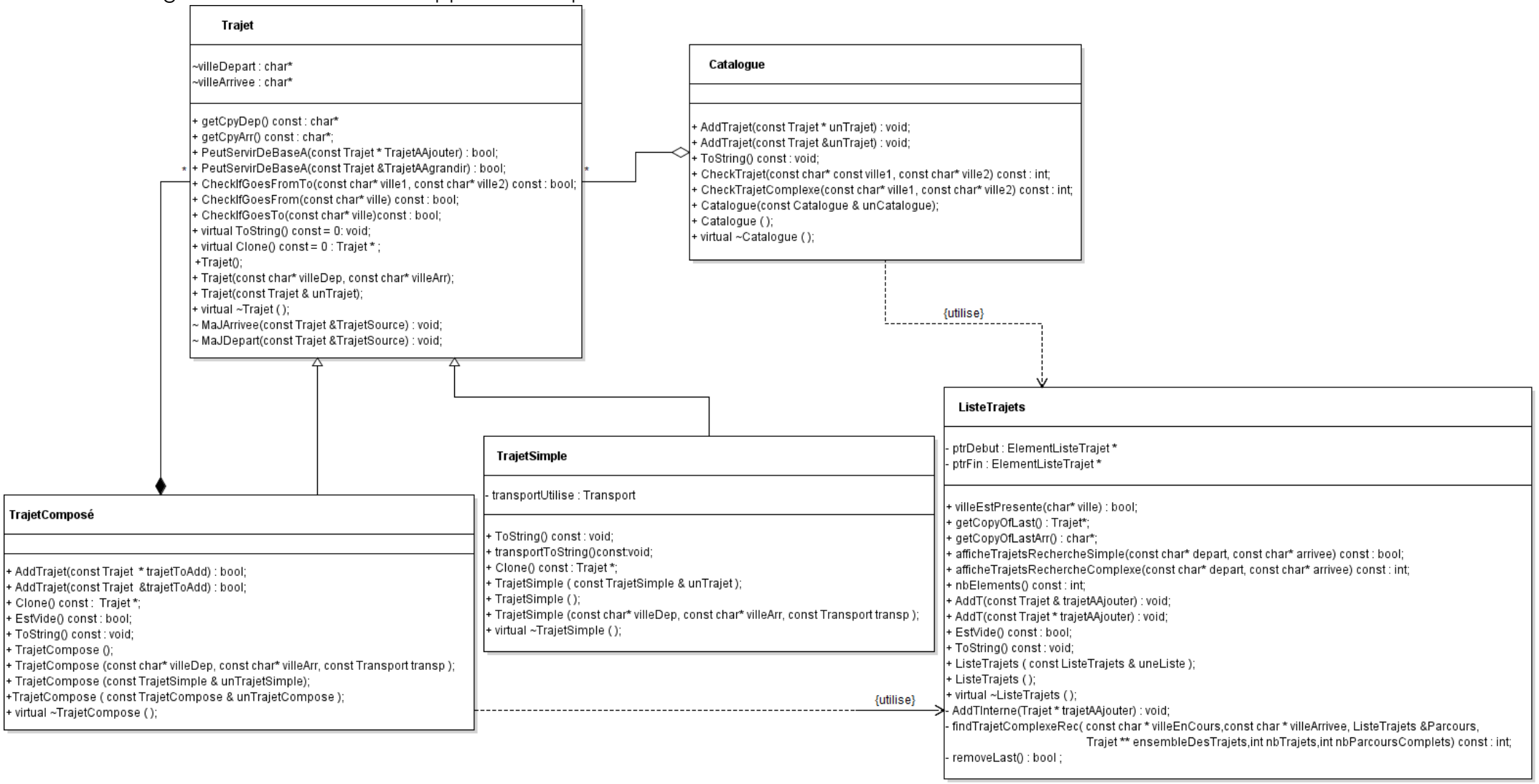
- La classe « ListeTrajets » est une classe dont le rôle est de contenir des éléments de type pointeur sur Trajet (dans le but d'exploiter un maximum les opportunités du polymorphisme C++). Chaque élément de la liste est défini à partir d'une structure, accessible depuis l'interface ListeTrajets.h, comportant un pointeur sur le trajet courant, ainsi qu'un pointeur sur l'élement suivant de la liste (nullptr si aucun ou dernier élément). La classe se contente de mémoriser sous forme d'attributs les pointeurs de tête et de queue de la liste des éléments. Plusieurs services déclarés publics sont accessibles au sein de la liste (ajouter, rechercher un trajet, vérifier si la liste est vide, renvoyer le nombre d'éléments…).

Remarque : Par souci de sécurisation des données traitées, toutes les méthodes publiques ou protégées renvoyant des pointeurs renvoient des pointeurs sur des COPIES EN PROFONDEUR des données initialement concernées. La libération de la mémoire est alors à la charge de l'utilisateur de ces services.

[1] La recherche simple est ici définie comme un affichage des trajets de A à B sans envisager de compositions de trajets.
[2] La recherche complexe inclut quant à elle les compositions de trajets.

2. *Diagramme des classes*

Le diagramme des classes de l'application se présente sous la forme suivante :



Quelques remarques à propos du diagramme des classes :

- Les relations d'agrégation () et de composition () impliquent que les classes du côté de l'étoile (*) s'agrègent sous la forme d'un attribut de type pointeur sur « ListeTrajets » au sein de la classe située à l'autre extrémité de l'association. Ainsi, « TrajetComposé » et « Catalogue » possèdent toutes deux des attributs de ce type qui leur permet de stocker de manière dynamique des pointeurs sur des Trajets. Le niveau de visibilité de cet attribut Liste est fixé à « private ». Il a été choisi de différencier les associations pour le trajet composé et le catalogue, car aucun trajet composé vide ne doit pouvoir être créé par un utilisateur.

- Les membres précédés d'un « + » sont publics, ceux précédés d'un « ~ » protégés et ceux précédés d'un « - » sont privés. Par convention, les attributs sont représentés avant les méthodes.

- Les relations « utilise » symbolisent ici le fait que c'est une liste de trajets (plus précisément un pointeur d'objet liste) qui est utilisée pour stocker les trajets au sein des classes « TrajetComposé » et « Catalogue », et aucune autre structure de donnée.

- L'attribut de type « Transport » de la classe « TrajetSimple » est défini à partir d'une énumération définie dans Trajet.h.

II. Détail de la structure de donnée utilisée (« ListeTrajets »)

1. *Représentation de la mémoire dans un exemple concret*

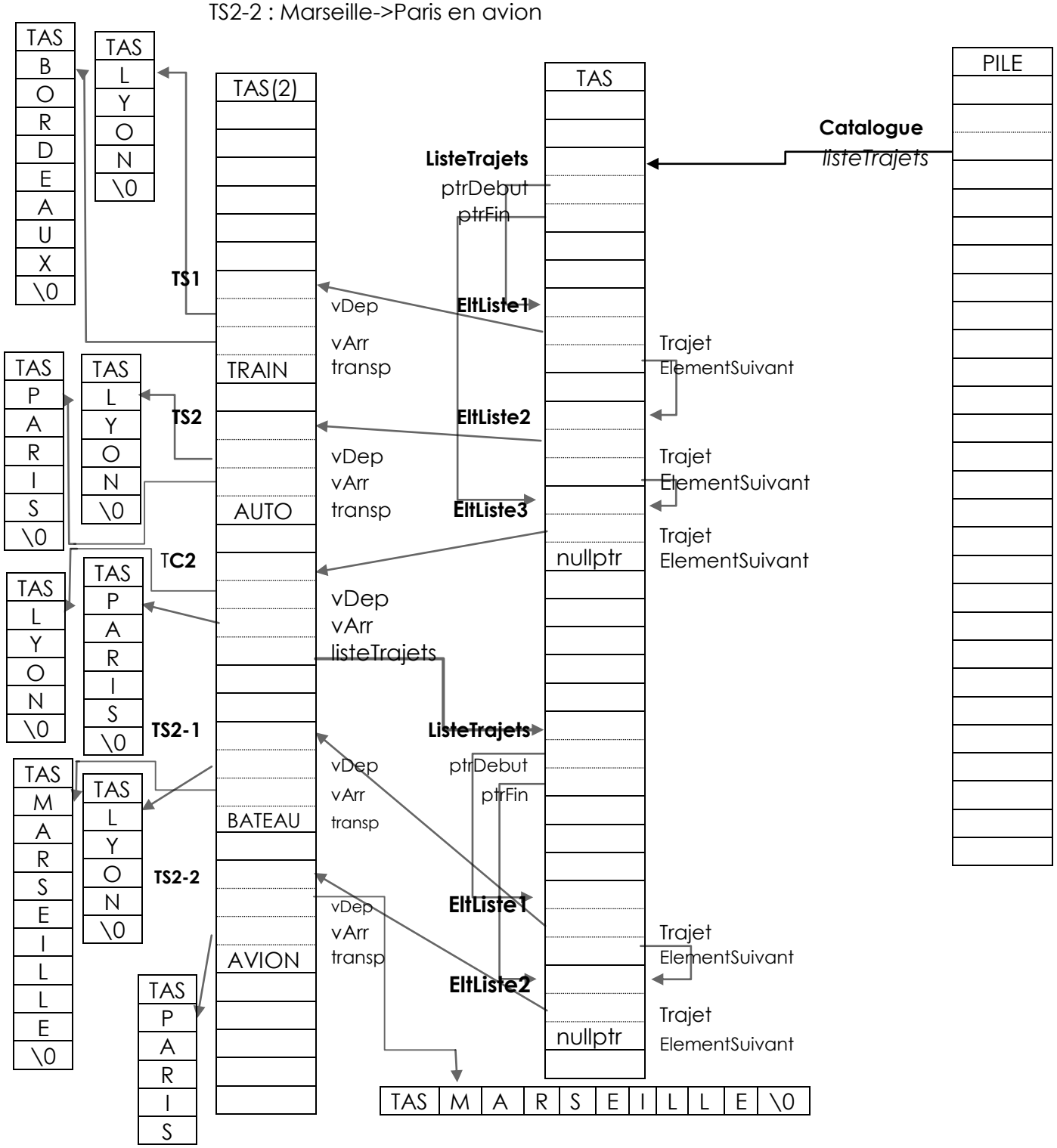
TS1 : Lyon->Bordeaux en train

TS3 : Lyon->Paris en auto

TC2 : Trajet composé de :

TS2-1 : Lyon->Marseille en bateau

TS2-2 : Marseille->Paris en avion



Remarques :

- Une flèche sortante d'une case mémoire indique un pointeur sur une donnée mémoire allouée dynamiquement.

- Les attributs de type « Transport » de Trajet.h de chaque trajet simple sont ici représentés par les littéraux définis statiquement dans cette interface.

III. Listing des classes et documentation

1. *Classe TRAJET*

Description : Un trajet abstrait, reliant deux villes (départ et arrivée) entre elles.

A. Méthodes

Type retour/ Nom méthode	Paramètres	Visib.	Mode d'utilisation
Char* getCpyDep	Aucun	Pub.	Renvoie une copie profonde de la ville de départ
Char* getCpyArr	Aucun	Pub.	Renvoie une copie profonde de la ville d'arrivée
Bool PeutServirDeBaseA	const Trajet* TrajetAAjouter	Pub.	Méthode vérifiant si le trajet appelant peut être ajouté à la suite du Trajet en paramètre
Bool PeutServirDeBaseA	const Trajet& trajetAAgrandir	Pub.	Méthode vérifiant si le trajet appelant peut être ajouté à la suite du Trajet en paramètre
Bool CheckIfGoesFromTo	const char* ville1, const char* ville2	Pub.	Méthode vérifiant si le trajet mène de la ville 1 à la ville 2
Bool CheckIfGoesFrom	const char* ville	Pub.	Méthode permettant de savoir si le trajet part d'une ville
Bool CheckIfGoesTo	const char* ville	Pub.	Méthode permettant de savoir si le trajet arrive à une ville
Void ToString	Aucun	Pub.	Méthode abstraite d'affichage des trajets
Trajet* Clone	Aucun	Pub.	Méthode abstraite permettant la copie d'un trajet
Trajet	Aucun	Pub.	Constructeur non paramétré, génère un trajet partant d'INCONNU vers INCONNU
Trajet	const char* villeDep, const char* villeArr	Pub.	Constructeur d'un trajet, prend en paramètre les noms des villes de départ/arrivée
Trajet	const Trajet & unTrajet	Pub.	Constructeur par copie
~Trajet	Aucun	Pub.	Destructeur de la classe, désalloue les membres dynamiques
Void MaJArrivee	const Trajet &TrajetSource	Prot.	Change la valeur de la ville d'arrivée par celle de TrajetSource
Void MaJDepart	const Trajet &TrajetSource	Prot.	Change la valeur de ville de Départ par celle de TrajetSource

B. Attributs

Type/Nom	Visibilité	Description
char* villeDepart	Prot.	Chaîne représentant la ville de départ
char* villeArrivee;	Prot.	Chaîne représentant la ville d'arrivée

2. *Classe TRAJET SIMPLE*

Description : Un Trajet simple, d'une ville A à une ville B, caractérisé par un moyen de transport (énumération).

A. Méthodes

Type retour/ Nom méthode	Paramètres	Visib.	Mode d'utilisation
void ToString	Aucun	Pub.	Affiche le trajet simple
void transportToString	Aucun	Pub.	Affiche l'énumération Transport correspondante au trajet simple
Trajet * Clone	Aucun	Pub.	Renvoie un pointeur vers une copie de l'élément appelant
TrajetSimple	const TrajetSimple & unTrajet	Pub.	Constructeur de copie d'un trajet simple
TrajetSimple	Aucun	Pub.	Génére un trajet simple allant de INCONNU à INCONNU en moyen de transport inconnu
TrajetSimple	const char* villeDep, const char* villeArr, const Transport transp	Pub.	Constructeur d'un trajet simple à partir des villes de départ et d'arrivée, et du mode de transport
~TrajetSimple	Aucun	Pub.	Destructeur d'un trajet simple

B. Attributs

Type/Nom	Visibilité	Description
Transport transportUtilise	Priv.	Mode de transport utilisé

3. *Classe TRAJET COMPOSE*

Description : Un trajet composé de plusieurs trajets simples via une liste chaînée d'objets en mémoire.

//A compléter après nettoyage…

4. *Classe LISTE TRAJETS*

Description : Liste contenant des pointeurs vers des éléments de type Trajet.

//A compléter après nettoyage…

5. *Classe CATALOGUE*

Description : "Catalogue" contenant une liste d'objets de type Trajet, sous forme de liste chaînée.

//A compléter après nettoyage…

6. *Enumération TRANSPORT*

Description : Enumération recensant tous les moyens de transport sélectionnables par l'utilisateur.

IV. Conclusion et retour sur le TP

Ce TP nous aura permis de nous familiariser davantage avec le polymorphisme C++, et avec le fonctionnement des mécanismes d'allocation et d'abstraction.

Nous aurons passé un certain temps à choisir les structures de données à utiliser, ainsi qu'à définir les différentes classes de cette application (travail entrepris au cours de la séance 1).

De plus, nous avons cherché, au cours de la réalisation, à maintenir un degré d'encapsulation fort, ce qui nous a incités à « protéger » les zones mémoires allouées dynamiquement dont les adresses étaient renvoyées par des méthodes de classe. L'utilisation des mécanismes d'abstraction a également constitué une part importante de notre temps de réflexion : la copie de liste, par exemple, a nécessité l'utilisation d'une méthode virtuelle pure, Clone(), renvoyant un pointeur de type adapté utilisé pour copier un trajet d'une liste à l'autre.

Enfin, le dernier challenge de ce TP aura été d'implémenter de manière récursive la fonctionnalité de « recherche complexe », tenant compte des compositions de trajets et recourant au « backtracking ».

En conclusion, l'implémentation de plusieurs améliorations reste possible à partir du modèle courant : par exemple, il serait possible d'ajouter des trajets composés imbriqués les uns dans les autres. Une telle fonctionnalité entraînerait plusieurs modifications à effectuer dans l'interface de l'application, pour faciliter la mise en œuvre de cette action par l'utilisateur. De plus, une éventuelle sérialisation (persistance) des données pourrait être réalisée en enregistrant les trajets du catalogue sur un fichier, dans un format spécifique (XML paraîtrait ici adapté pour représenter les trajets inclus dans des trajets composés). Il ne resterait alors qu'à mettre en place des services d'initialisation et de sauvegarde de l'application qui se chargeraient respectivement de charger et de sérialiser les données entrées par l'utilisateur. La mise en œuvre de cette fonctionnalité requiert cependant d'inclure certaines librairies C++ externes (XercesXML par exemple).