Table des matières

[1. Livrables de capture et analyse des besoins 2](#_Toc434414334)

[1.1 Planning prévisionnel du projet 2](#_Toc434414335)

[1.2 Modèle du domaine 3](#_Toc434414336)

[1.3 Glossaire 4](#_Toc434414337)

[1.4 Diagramme de cas d'utilisation 4](#_Toc434414338)

[1.5 Description textuelle structurée des cas d'utilisation 4](#_Toc434414339)

[2. Livrables de conception: 4](#_Toc434414340)

[2.1 Liste des événements utilisateur et diagramme Etats-transitions 4](#_Toc434414341)

[2.2 Diagrammes de packages et de classe 5](#_Toc434414342)

[2.3 Diagrammes de packages et de classes retro-générés à partir du code 5](#_Toc434414343)

[2.4 Choix architecturaux et design patterns utilisés 5](#_Toc434414344)

[2.5 Diagramme de séquence du calcul de la tournée à partir d'une demande de livraison 6](#_Toc434414345)

[3. Implémentation et tests 6](#_Toc434414346)

[3.1 Code du prototype et des tests unitaires 6](#_Toc434414347)

[3.2 Documentation JavaDov du code 6](#_Toc434414348)

[3.3 Diagrammes de packages et de classes retro-générés à partir du code: 6](#_Toc434414349)

[4. Bilan 6](#_Toc434414350)

[4.1 Planning effectif du projet 6](#_Toc434414351)

[4.2 Bilan humain et technique 7](#_Toc434414352)

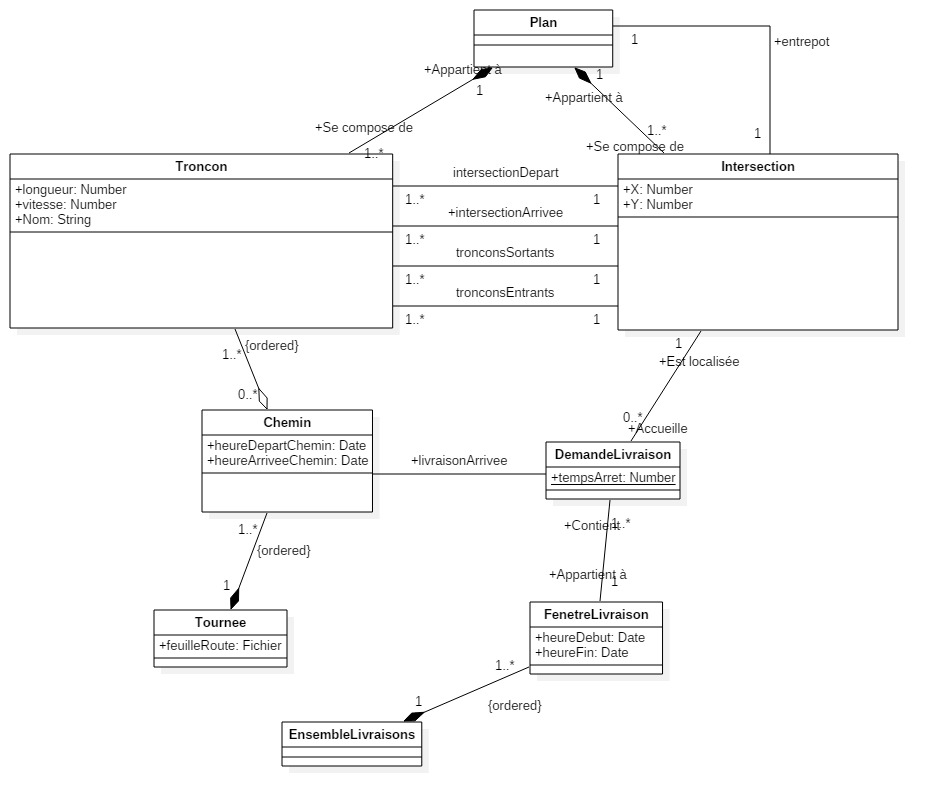
[5. Conclusion 7](#_Toc434414353)

# 

# 1. Livrables de capture et analyse des besoins

## 1.1 Planning prévisionnel du projet

## 1.2 Modèle du domaine



## 1.3 Glossaire

## 1.4 Diagramme de cas d'utilisation

## 1.5 Description textuelle structurée des cas d'utilisation

# 2. Livrables de conception:

## 2.1 Liste des événements utilisateur et diagramme Etats-transitions

## 2.2 Diagrammes de packages et de classe

## 2.3 Diagrammes de packages et de classes retro-générés à partir du code

## 2.4 Choix architecturaux et design patterns utilisés

Au niveau de l’architecture générale du projet, nous avons construit l’application sur une architecture MVC (Modèle-Vue-Contrôleur). Premièrement car cette architecture était imposée, et ensuite car cela correspondait au type d’application développé ici.

Pattern Observer

Lorsqu’une tournée a été calculée par le système, l’utilisateur doit pouvoir modifier cette tournée (ajout, suppression, échange de points de livraisons). Afin que la vue puisse communiquer avec le modèle, (par exemple, que la vue soit informée de l’ajout d’une livraison à l’ensemble de livraison), nous avons utilisé le pattern *observer*. Ainsi la vue peut-être *observer* des objets Tournée et EnsembleLivraisons, qui sont quant à eux *observable*.

Pattern undo/redo

Dans le cahier des charges, il est spécifié qu’une modification de la tournée doit pouvoir être annulée. Une telle spécification appelle clairement à l’utilisation du pattern Commande, qui permettra de faire aisément des undo/redo sur les modifications.

ModeleManager

Lorsque le contrôleur communique avec le modèle, il est souhaitable que la communication se fasse au moyen d’un unique point d’entrée sur le modèle. Nous avons fait le choix d’une classe ModeleManager, qui contiendrait alors le plan actuellement chargé, les demandes de livraison actuellement chargées, et la tournée actuellement générée. C’est aussi cette classe qui contient la méthode calculerTournee. À noter que le modeleManager aurait pu être implémenté en utilisant le pattern singleton (une seule et unique instance de ModeleManager dans l’application). Cependant, dans un souci de réutilisabilité, nous avons écarté ce choix. En effet, dans le cas où, par exemple, l’application permettrait d’avoir plusieurs onglets contenant plusieurs plans différents en même temps, il serait pratique d’avoir plusieurs instances de ModeleManager.

Package XML

Les descriptions des plans de villes et les descriptions des demandes de livraisons sont réalisées au moyen de fichiers au format xml. Ces fichiers doivent être parsés, et les informations contenues doivent être analysées, puis envoyées dans le modèle. Nous avons décidé de placer le composant applicatif relatif au parsing dans un package particulier (différents des trois packages principaux), le package xml.

Pattern State

L’application que nous développons possède plusieurs états de fonctionnement (Le plan est chargé, puis les demandes de livraisons sont chargées, etc.) Les possibilités d’interaction utilisateur et les effets des interactions ne sont pas les même selon l’état de l’application. Afin que le contrôleur puisse gérer les différents états, nous avons implémenté le pattern state.

Gestion des fonctionnalités actives en fonction des états

Selon l’état du contrôleur, toutes les fonctionnalités de l’application ne doive pas pouvoir être appelée (Lorsqu’un plan n’est pas chargé, par exemple, l’utilisateur ne devrait pas pouvoir demander à l’application de charger des demandes de livraison). C’est pourquoi nous avons mis en place un système d’activation/désactivation de fonctionnalité. La fenêtre graphique de l’application implémente une méthode d’activation/désactivation de fonctionnalité pour chaque fonctionnalité (on entend ici les boutons, les items des menus, .etc.). A chaque changement d’état, l’état désactive l’ensemble des fonctionnalités et réactive les fonctionnalités permises dans cet état.

## 2.5 Diagramme de séquence du calcul de la tournée à partir d'une demande de livraison

# 3. Implémentation et tests

## 3.1 Code du prototype et des tests unitaires

## 3.2 Documentation JavaDov du code

## 3.3 Diagrammes de packages et de classes retro-générés à partir du code:

# 4. Bilan

## 4.1 Planning effectif du projet

## 4.2 Bilan humain et technique

# 5. Conclusion