Table des matières

[1. Introduction 2](#_Toc434700488)

[2. Livrables de capture et analyse des besoins 2](#_Toc434700489)

[1.1 Planning prévisionnel du projet 2](#_Toc434700490)

[1.2 Modèle du domaine 2](#_Toc434700491)

[1.3 Glossaire 2](#_Toc434700492)

[1.4 Diagramme de cas d'utilisation 4](#_Toc434700493)

[1.5 Description textuelle structurée des cas d'utilisation 4](#_Toc434700494)

[2. Livrables de conception: 4](#_Toc434700495)

[2.1 Liste des événements utilisateur et diagramme Etats-transitions 4](#_Toc434700496)

[2.2 Diagrammes de packages et de classe 5](#_Toc434700497)

[2.3 Diagrammes de packages et de classes retro-générés à partir du code 5](#_Toc434700498)

[2.4 Choix architecturaux et design patterns utilisés 5](#_Toc434700499)

[2.5 Diagramme de séquence du calcul de la tournée à partir d'une demande de livraison 6](#_Toc434700500)

[3. Implémentation et tests 7](#_Toc434700501)

[4. Bilan 7](#_Toc434700502)

[4.1 Planning effectif du projet 7](#_Toc434700503)

[4.2 Bilan humain et technique 7](#_Toc434700504)

# 1. Introduction

L'application est inspirée d'un projet réel, piloté par le Grand Lyon entre 2012 et 2015; qui vise à optimiser la mobilité durable en ville (voir www.optimodlyon.fr). Nous nous focaliserons ici sur la partie concernant le fret urbain et l'optimisation des tournées de livraisons en ville.

Pour ce faire, nous avons adopté une méthodologie de développement orienté objet. Nous avons commencé par une capture et une analyse des besoins, ensuite une conception détaillée pour en finir avec une implémentation et des tests de l'application en question. Pour satisfaire le cahier des charges avec les fonctionnalités demandées, nous avons consacré 4 semaines de travail avec une charge de xxx heures. Le groupe s'est mis d'accord sur le langage orienté objet java vu sa performance et sa réputation mondiale et le framework SWING pour la partie IHM.

# 2. Livrables de capture et analyse des besoins

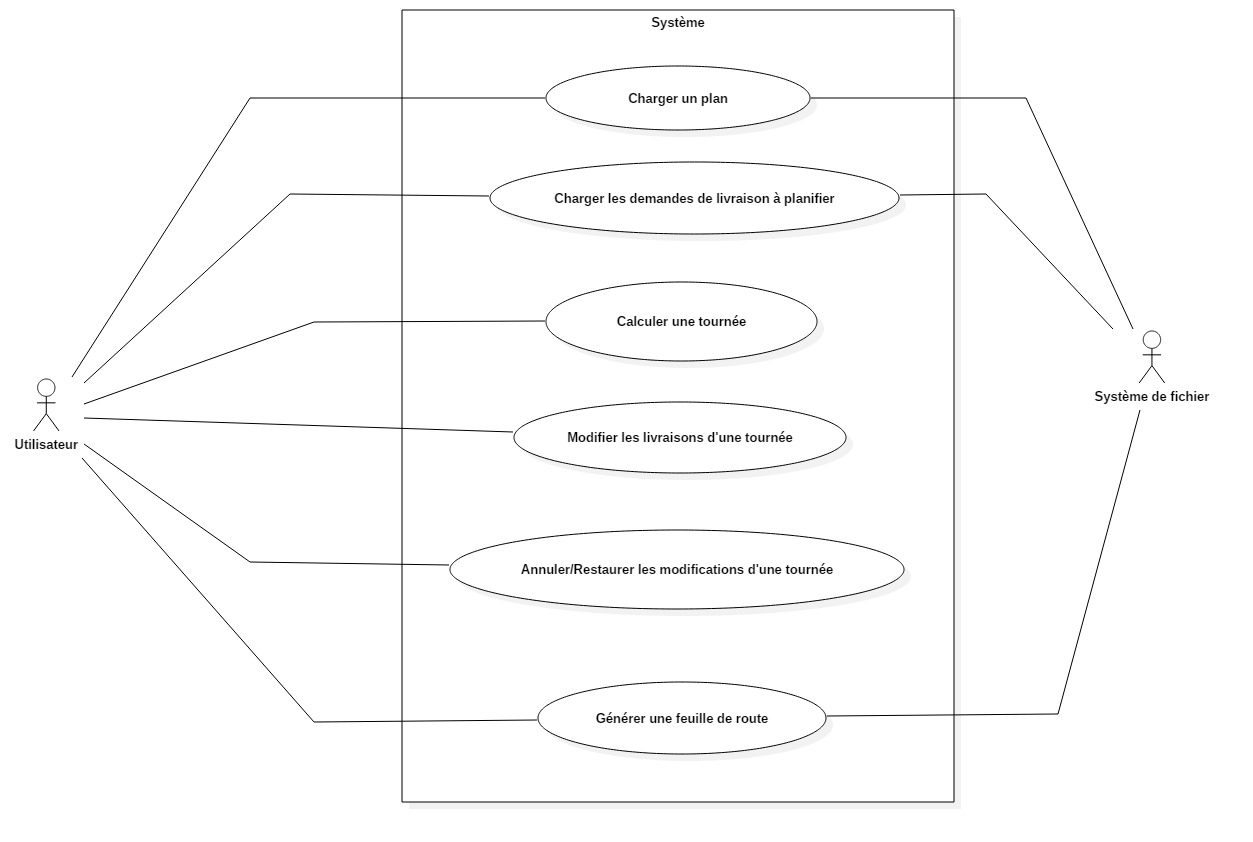
## 2.1 Planning prévisionnel du projet

## 2.2 Modèle du domaine

## 2.3 Glossaire

* Chemin : Chemin entre deux livraisons.
* Heure de départ (d’un Chemin) : Heure à laquelle le livreur quitte la livraison précédente (donc temps de livraison non compris) ou l’entrepôt.
* Heure d’arrivée (d’un Chemin): Heure à laquelle le livreur arrive à la livraison suivante, nommée livraison arrivée (temps de livraison non compris).
* livraisonArrivee : La dernière livraison d'un chemin.
* Demande de livraison : Une demande de livraison.
* Temps d’arrêt (d’une demande de livraison) : Temps d'arrêt pour une livraison, ici fixé à 10 minutes pour chacune d'entre elles.
* Fenêtre de livraison : Regroupement des demandes de livraisons partageant la même fenêtre temporelle de livraison.
* Heure de début (d’une fenêtre de livraison) : Heure à partir de laquelle les demandes de livraison d’une fenêtre de livraison sont attendues.
* Heure de fin (d’une fenêtre de livraison) : Heure jusqu’à laquelle les demandes de livraison d’une fenêtre de livraison sont attendues.
* Ensemble de livraisons : Regroupement, ordonné temporellement, des fenêtres de livraison, qui contiennent-elles même les demandes de livraison.
* Intersection : Intersection de deux ou plusieurs tronçons. Une intersection peut correspondre à un point de livraison.
* x (d’une intersection) : Coordonnée X d’une l'intersection dans le plan.
* Y (d’une intersection): Coordonnée Y d’une l'intersection dans le plan.
* Tronçons sortants (d’une intersection) : Ensemble des tronçons quittant une intersection.
* Tronçons entrants (d’une intersection) : Ensemble des tronçons arrivant sur une intersection.
* Tronçon : Route empruntable entre deux intersections.
* Longueur (d’un tronçon) : longueur physique d'un tronçon.
* Vitesse (d’un tronçon) : vitesse moyenne du livreur sur le tronçon.
* Intersection de départ (d’un tronçon) : intersection à partir de laquelle commence un tronçon
* Intersection d’arrivée (d’un tronçon) : intersection sur laquelle arrive un tronçon.
* Plan : Cartographie les intersections et les tronçons.
* Tournée : Ensemble ordonné de chemins. Détermine la totalité du trajet du livreur depuis l'entrepôt jusqu'à l'entrepôt, en passant par les points de livraison.
* Feuille de route : fichier au format texte généré donnant la liste des livraisons à faire, dans l'ordre, avec l'adresse, horaires de livraison (arrivée et départ) et l'itinéraire entre chaque livraison.
* Entrepôt : Intersection de départ et d'arrivée d'une tournée.

## 2.4 Diagramme de cas d'utilisation



## 

## 2.5 Description textuelle structurée des cas d'utilisation

# 3. Livrables de conception:

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

## 3.2 Diagrammes de packages et de classe

## 3.3 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 séparer le code relatif au traitement xml dans deux packages séparés : Un package xml, qui concerne uniquement la gestion globale de fichier au format xml (ouverture, sauvegarde, sélection de fichier), et un package xmlModele, qui contient toutes les méthodes de parsing et de remplissage des objets métier. Ainsi, le package xml est générique et pourrait être réutilisé dans une autre application. Le package xmlModele, quant à lui, est intrinsèquement lié au modèle.

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.

Architecture de la vue

La partie graphique de l’application est constituée d’un objet Fenetre, qui contient un état de l’application dans la vue, modélisé par l’objet Vue. Nous avons essayé d’implémenter notre vue de la manière la plus générique possible. Par exemple, dans le cas où l’application permettrait d’avoir plusieurs onglets contenant plusieurs plans différents en même temps, l’objet Vue déjà existant peut facilement être assimilé à un onglet. Il n’y aurait pas beaucoup de changement à faire dans le code.

Nous avons aussi réalisée une classe de vue dédiée à chacun (ou presque) des objets du modèle, et mis en place des liens entre ces classes qui sont semblables aux liens entre les classes du modèle. Ainsi on peut facilement paramétrer l’apparence graphique de chacun des éléments, tout en conservant la logique métier.

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

# 4. Implémentation et tests

## 

# 5. Bilan

## 5.1 Planning effectif du projet

## 5.2 Bilan humain et technique