Projet OLOCAP

Compte-rendu

Participants :

De Almeida Matéo

Hinshberger Titouan

Fargeot Léa

Muller Maxime

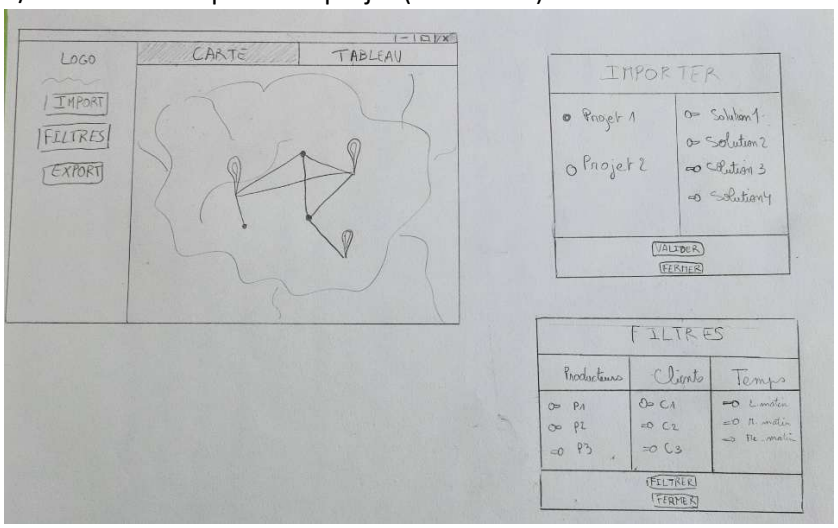
Thierry Avril

Référent Projet :

Monsieur Pierre Desport

Cahier des charges

1/ Réaliser la maquette du projet (ci-dessous)



2/ Implémenter la maquette (design et visuel global de la maquette : boutons, onglets carte/tableau, carte vide, tableau vide, popup Import et popup Filtres vides)

3/ Concevoir le diagramme de classes métier

4/ Implémenter les classes métier

5/ Récupérer les informations tirées des fichiers de données fournis (liste de producteurs, de clients, de commandes)

6/ Récupérer les informations des fichiers solutions fournis (liste des tournées)

7/ Tests des classes de récupération de données

8/ Créer des marqueurs sur la carte

9/ Charger les données sur la carte (Affichage des marqueurs producteurs et clients)

10/ Charger les données dans le tableau (tableau de producteurs, de clients et de commandes)

11/ Créer des popup d'information lorsqu'on clique sur un marqueur

12/ Implémenter les filtres, sans liaison avec l'interface graphique (filtre par producteurs, par clients et par demi-jour)

13/ Tests des classes de filtres

14/ Charger les fichiers solutions dans l'application

15/ Afficher les trajets entre les marqueurs sur la carte

16/ Créer des popup d'information lorsqu'on clique sur un trajet

17/ Afficher les tournées dans le tableau

18/ Intégrer les filtres à l'interface graphique (carte + tableau)

19/ Implémenter les vérifications de conformité des fichiers (charge finale de la tournée = 0, pick-up toujours réalisé avant le drop-off correspondant, les tournées concernent des acteurs tous présents dans le fichier de données et les acteurs sont disponibles pour la tournée)

20/ Implémenter la fonctionnalité capture d'écran

21/ Tests de l'application

22/ Générer l'exécutable du projet

Choix techniques

Pour réaliser ce projet, nous avons choisi d'utiliser le langage python car ses nombreuses bibliothèques, telles que tkintermapview, customtkinter et os, offraient toutes les fonctionnalités nécessaires pour la création de notre application.

Pour la gestion des données, nous avons fait le choix de conserver le format des fichiers fournis, et de directement les exploiter afin que les données soient toujours prêtes au chargement. Il n'y a donc pas de base de données à proprement parlé. Cela signifie qu'il n'y a pas de connexion à une base de données ni d'effacement nécessaires puisque les données sont récupérées à la décision de l'utilisateur, via le menu d'importation. Cela offre par ailleurs une certaine flexibilité puisque, tant que le contenu respecte le format défini, aucune restriction ne s'applique sur le nombre ni même le nom des fichiers. De plus, si un changement est effectué dans le format, le code est aisément adaptable. Par exemple, le séparateur (actuellement l'espace) est un attribut, et donc facilement modifiable.

Brève présentation de l'application

L'application que nous avons réalisée permet d'importer des fichiers de données (un seul à la fois), et de solutions (aucun ou plusieurs à la fois) ainsi que d'en vérifier l'intégrité, selon des critères mentionnés dans le cahier des charges. Ces données se trouvent dans le répertoire *Projets*, lui-même divisé en sous répertoires *Projet_1*, *Projet_2*… Elles peuvent être visualisées sous deux formes.

Sur une carte : par des marqueurs qui représentent les acteurs (producteur ou client) et des tracés qui représentent les trajets reliant les différents marqueurs.

Et sur un tableau : un tableau des producteurs, un des clients, un des commandes effectuées et un dernier des tournées réalisées pour chaque fichier solution importé.

Ces données sont filtrables par producteurs, par clients, par pas de temps (demi-journées) et par fichiers solution (sur la carte, possibilité de les voir tous en même temps ou un à la fois).

L'application permet également de prendre une capture d'écran de la zone où se trouvent la carte et le tableau. La capture se retrouve alors dans le dossier Exports (créé s'il n'existe pas déjà).

Réussites/Difficultés rencontrées au cours du développement du projet

Au cours de ce projet, nous avons rencontré plusieurs difficultés que nous avons fini par surmonter pour la majorité.

L'une des premières étapes fut la conception des classes métier. À partir du cahier des charges et des différentes données brutes contenues dans les fichiers .txt fournis à titre d'exemples, nous avons élaboré un diagramme UML des classes pour les objets métiers du logiciel (voir documents joints).

Le modèle validé par l'ensemble des parties du projet, le développement des classes métiers en Python a pu se dérouler sans difficulté. Des différences liées à l'évolution du projet peuvent être observées entre leur implémentation finale et le diagramme. Les attributs des classes sont tous publics, ce qui est souvent le cas en programmation Python dans le but de favoriser la simplicité et la transparence. D'autres attributs et méthodes ne figurant pas sur le diagramme ont également été ajoutés en cours du projet afin de répondre aux besoins de l'interface du logiciel. Par exemple, la classe *Tournee* contient des méthodes pour calculer les distances, durées et charges totales, maximales ou intermédiaires aux tâches.

Les classes métiers permettent de stocker de manière structurée les données brutes des fichiers afin qu'il soit plus efficace pour le logiciel de les parcourir et d'y accéder. Cependant, certaines données auraient dû être en plus stockées sous leur forme brute, telles que la liste des numéros des producteurs et clients passés lors d'une tournée. En effet, l'affichage de cette liste dans l'interface du tableau nécessite de parcourir les objets *Tache* de l'objet *Tournee* et de récupérer pour chacun l'*id* de l'objet *Acteur* associé à l'objet *Tache*.

Dans un deuxième temps, pour récupérer les données dans les fichiers, nous avons fait en sorte de créer des classes les plus indépendantes possibles. Ces dernières ne sont utilisées par l'interface graphique qu'à travers la classe *Createur* qui fait le lien entre les deux. Cela permet également de gérer le nombre d'instances de ces classes et de faire en sorte que, si un fichier est déjà chargé à l'échelle du projet en cours de sélection, il n'est pas chargé une seconde fois.

Il y a la classe *FileManager* qui s'occupe de parcourir l'arborescence des projets et de créer les dossiers/fichiers. Elle interagit directement avec le gestionnaire de fichiers. La classe *DataExtractor*, quant à elle, récupère le contenu des fichiers et les renvois sous forme de listes de lignes. Enfin, la classe *CreerClasses* se charge de créer les objets métiers en interprétant le contenu des fichiers (sous forme de listes de lignes). L'utilisation de méthodes privées et de *get* pour accéder aux données, permet de simplifier la tâche à l'utilisateur de la classe. Le *Createur* appelle les méthodes de ces classes dans d'autres *get* qui seront les seuls accès de l'interface aux classes de gestion de données. Il permet également une maintenance de l'application plus aisée, en centralisant les créations d'objet.

Les difficultés les plus importantes rencontrées lors de l'implémentation de ces quatre classes furent la synchronisation des instances et leur gestion. Par exemple, l'utilisation d'une méthode *load_donnees* dans *CreerClasses* garantit un appel unique aux méthodes privées qui créent les producteurs, les clients et les commandes. Les listes d'objets sont ensuite récupérées par des *get*.

Cependant, pour la création des tournées depuis les fichiers solution, dans la mesure où plusieurs peuvent être importés en même temps, ce principe ne pouvait pas s'appliquer. C'est là que le *Createur* est le plus utile puisque c'est lui qui temporise le nombre d'appels à la méthode.

Ensuite, l'interface fut une partie assez délicate du projet de par la prise en main d'un module qui nous était inconnu. Nous avons en effet opté pour le module graphique CustomTkinter, une version plus « récente » du module Tkinter de Python, qui nous a permis de mettre en place une application plus ergonomique et agréable à utiliser. Pour la carte, TkinterMapView nous a paru être le choix le plus logique, notamment grâce à ses fonctionnalités permettant de créer des marqueurs ainsi que des trajets simplement, de même que sa capacité à différencier aisément chaque élément. La création de ces deux modules revient à Tom Schimansky.

Si l'implémentation du menu principal, qui reste statique peu importe le type de visualisation choisi, s'est déroulée sans encombre, la gestion des changements de visualisation ainsi que celle des Popups s'est avérée plus ardue.

En effet, lors de l'affichage d'un Popup (Import ou Filtre), pour une raison dont nous n'avons toujours pas déterminé la cause, il nous est impossible de faire suivre la fenêtre par le popup créé, ce qui donne un affichage quelque peu désagréable si l'utilisateur souhaite bouger la fenêtre en ayant le popup d'ouvert.

Par ailleurs, il a fallu coordonner l'utilisation des imports de fichiers entre la carte et le tableau, afin de prévenir l'obsolescence des données dans l'un des deux affichages, et éviter à l'utilisateur de faire plusieurs fois la même manipulation.

De plus, lors des changements de vue, les marqueurs et les trajets restaient visibles, y compris lorsque l'utilisateur n'était plus sur l'application. Pour y remédier, nous cachons les informations dès que la fenêtre active n'est plus notre application.

Mais en dépit de tout cela, les plus grandes difficultés furent celles qui concernaient les données importées ainsi que leur utilisation pendant la création d'objets *Markers* et *Trajets*. Les premiers furent les plus faciles à traiter. C'est lors de l'inclusion des données des classes Tournées et Tâches que cela s'est compliqué. Nous avons dû faire un parcours de boucle sur plusieurs listes imbriquées pour pouvoir en ressortir en une seule fois : La tâche, la tournée à laquelle elle appartient, ainsi que les points de départ et d'arrivée qui dépendent de la tâche précédente.

Pour le calcul des distances, une fonction calcule la distance entre deux points sur Terre par leurs coordonnées GPS. Elle implémente pour cela la formule de haversine qui permet de calculer la distance du grand cercle, aussi appelée distance orthodromique (voir documents).

Grâce à tout cela, les données utilisées pour afficher des informations sur la carte et les tableaux sont définies. Pour la carte, *PopupImport* utilise la classe *AfficherCarte* qui affiche les producteurs, clients et trajets liés au(x) fichier(s) solution importé(s). Les tableaux sont ensuite notifiés de la présence de données lors du changement de vue et offrent la possibilité de les visualiser.

L'affichage des tableaux en eux-mêmes n'a pas posé de problème particulier, c'est plutôt la coordination des menus déroulant qui a demandé réflexion et surtout de nombreux essais. Finalement, nous avons un premier menu déroulant qui s'affiche après l'importation de données, et un second si l'option tournées (qui n'apparait que lorsqu'au moins un fichier solution est importé) est sélectionnée. Ce second menu propose un choix entre les différents fichiers solutions importés.

En ce qui concerne les filtres, le cahier des charges imposait que les données affichées sur la carte et dans les tableaux puissent être filtrées par producteur, par client et par pas de temps (demi-journées). Cela a pu être fait à travers une fenêtre d'options de filtrage *PopupFiltre* qui permet à l'utilisateur de choisir facilement et rapidement ses critères qu'il désire voir pris en compte lors du filtrage, à savoir : les producteurs, les clients, les demi-jours, ainsi que la provenance des tournées (autrement dit : de quel fichier solution chargé souhaite-t-on voir les tournées ?).

Le filtrage des tournées est effectué à la validation de la fenêtre, par l'intermédiaire d'une fonction qui prend en paramètres l'ensemble des tournées chargées ainsi que les choix de l'utilisateur, et renvoie les tournées qui satisfont tous ces choix.

Suite à cela, la mise à jour de l'affichage des données sur la carte et les tableaux est effectuée.

Pour la carte, il faut, en plus des acteurs choisis par l'utilisateur, afficher ceux dont les tournées passent par chez eux pour un souci de cohérence graphique. Sinon, toutes les autres données qui ne correspondent pas au filtrage sont cachées.

Pour les tableaux, la mise à jour se fait indirectement en modifiant des listes en attribut de *AfficherTableau,* qui sont vérifiées à chaque construction.

Le développement du filtrage pour la partie interface, bien que longue, n'a pas été très compliquée. La partie fonctionnelle (première partie réalisée) non plus, même si elle a demandé une certaine réflexion et décision sur la manière de filtrer. Des tests ont été réalisés sur cette partie pour être sûr de son bon fonctionnement.

Enfin, la partie post-filtrage (mise à jour) a été plus difficile que prévu car il fallait modifier temporairement des éléments graphiques sur la carte sans modifier leur fonctionnement d'origine. Pour des questions de performance, nous ne pouvions guère nous permettre de détruire ces éléments lorsqu'ils devaient disparaître, c'est pourquoi nous avons dû faire preuve d'astuce.

Nous avons réalisé la vérification de la conformité des fichiers à importer, une fois tout le reste terminé. Cela nous a permis d'avoir une vue plus claire de comment les données sont importées et utilisées dans notre application, ce qui a rendu évidente la façon dont devait se dérouler l'annulation, notamment, d'une importation, si l'utilisateur venait à la demander après l'apparition d'erreurs.

Pour aller plus loin avec cette application, de nombreuses améliorations peuvent être apportées, en voici quelques-unes :

- Actuellement, les trajets se déroulant entre deux même points se superposent, il serait souhaitable que, lorsque plusieurs fichiers solution sont importés, l'on puisse faire la différence entre les trajets des différentes tournées et fichiers.

- Ajout de la possibilité de modifier un trajet, ou plus généralement, une tournée, directement dans l'application (avec ou non application de ce changement directement dans le fichier .txt). On peut imaginer que c'est une bonne solution si une idée de modification vient au cours de l'utilisation de l'application. On la réalise, on la teste sur la carte. Si elle convient, on l'incorpore dans le fichier de manière définitive, sinon, on l'annule.

- Enfin, le code en lui-même peut être sujet à réflexion pour améliorer encore plus que nous nous sommes efforcés de le faire, les performances de l'application.