RUFFENACH Maël
SAADI Mehdi

RAPPORT - TRANSCONNECT

Nous proposons tout d'abord une page de connexion où seul Monsieur Dupont peut se connecter (avec une méthode qui remplace les caractères du mot de passe par des étoiles). Pour la page d'accueil, nous avons réutilisé le projet de MDD de l'année dernière pour la page de connexion, le projet BelleFleur.
Monsieur Dupont peut se connecter avec les logins suivants :

mail : eric.dupont@transconnect.fr
mot de passe : Esilv

Une fois arrivé dans le menu principal, il faut taper 1, 2… 5 pour accéder aux différents modules, puis de nouveau 1…4 pour accéder aux différentes méthodes proposées dans chaque module.

- Module Client :

Au début de notre projet, on avait envisagé d'utiliser un tableau Excel pour stocker les clients et leurs attributs. L'idée était de pouvoir gérer et manipuler facilement les données des clients sous forme de cellules dans une feuille de calcul.

Cependant, on n'a pas réussi à gérer les tableaux Excel.En conséquence, on a décidé de changer de stratégie et d'utiliser des listes pour stocker les clients. Cette approche s'est révélée plus flexible et mieux adaptée à nos besoins.La classe Client permet de représenter les clients avec plusieurs attributs importants comme l'identifiant unique pour chaque client, le nom, le prénom, la ville de résidence, l'adresse, l'adresse e-mail, le numéro de téléphone, le statut (qui peut être "Bronze", "Argent" ou "Or"), le solde du compte client, et le nombre de commandes effectuées par le client.

Pour gérer la liste des clients, on utilise la classe ListeClients. Cette classe static offre des méthodes pour ajouter, supprimer et obtenir la liste des clients. Nous avons choisi d'utiliser une classe static pour la liste car cette classe nous permet d'appeler la liste de manière simple et d'y avoir accès à tout moment dans l'exécution du code.

Pour l'affichage et le tri des clients selon différents critères, on a développé la classe ClientInterface.
Comme la liste des clients est static, si nous voulons implémenter les interfaces, la classe ne doit pas être static, donc nous revenons à un type que nous connaissons. Cette classe permet d'afficher les clients par ordre alphabétique de leurs noms, de filtrer les clients par ville spécifique, de trier les comptes clients par ordre décroissant, et de calculer et trier les rapports entre le compte client et le nombre de commandes.

Bien que notre approche initiale avec Excel n'ait pas abouti, l'utilisation des listes s'est avérée être une solution efficace pour la gestion des clients. Les méthodes développées permettent de trier et

d'afficher les informations des clients selon différents critères, répondant ainsi aux besoins de notre projet.

- Module Commande :

Constructeur :

Lorsque l'on crée une commande, nous n'indiquons seulement le client qui passe commande, la ville de départ et celle d'arrivée ainsi qu'une durée (qui correspondra à la durée de livraison). Concernant le choix de la date de livraison, nous avons décidé d'ajouter 10 jours à la date d'aujourd'hui pour toutes les livraisons.
Concernant le choix du chauffeur, nous utilisons une méthode *ChoixChauffeur(DateTime date)* qui renvoie le chauffeur disponible à la date donnée.
Concernant le choix du véhicule, nous utilisons une méthode ChoixVehicule() qui en fonction des besoins de l'utilisateur renvoie le véhicule correspondant (sous forme de question/réponse ; ex : "Quel type de véhicule souhaitez-vous ?" "nous avons : 1-Voiture, 2-Camionnette…), puis nous créons dans cette méthode le véhicule correspondant.
Pour le prix de la commande, nous utilisons une méthode *CalculerPrix()* qui calcule le prix de la commande en fonction du véhicule utilisé (ex : si c'est une voiture, chaque passager est facturé 20 euros, et 200 euros de frais sont appliqués à la voiture… pour les autres modèles). De plus, nous utilisons également la variable float duree, qui renvoie la durée du trajet (ex 2h30 entre deux villes) et nous utilisons le paramètre salaire de la classe Salarie (qui correspond au salaire horaire des salariés) pour ajouter au prix de la commande le salaire du chauffeur.

Pour calculer la duree, tout se fait dans le main. En effet, à partir des villes de départ et d'arrivée, nous appelons les méthodes *Dijkstra(Noeud1 depart, Noeud1 arrivee),* puis la méthode *CalculerDuree(List<Etapes> etapes)*. Pour trouver les nœuds de départ et d'arrivée, nous utilisons une méthode *Find*. Puis, nous pouvons utiliser les méthodes de l'algorithme de Dijkstra.

Pour créer ces méthodes, nous avons créé 4 classes : la classe Noeud1 qui représente la définition d'un noeud (défini par un symbole), la classe Lien qui représente les données entre chaque noeuds (la distance en km et la durée qui sépare 2 noeuds, donc 2 villes), la classe Étape, qui représente le calcul fait à chaque étape sur un noeud du graphe avec l'algorithme. Cette classe contient un lien entre 2 nœuds, la distance en km, la durée et le nœud précédent au nœud actuel. D'ailleurs, la méthode Dijkstra renvoie une liste d'étapes. Dans notre code, chaque étape est représentée sous la forme : "Paris" -> "Marseilles", par exemple, qui correspond à deux villes reliées entre elles. Enfin, nous avons la classe Graphe, classe qui construira l'itinéraire grâce à Dijkstra et qui affichera également la distance parcourue (en sommant les distances des différents liens) et qui affichera le graphe. De plus, dans le main, nous avons une liste de noeuds List<Noeuds1> listnoeuds qui contient toutes les villes (les noeuds) ainsi que leurs liens grâce à la méthode *AjouterArrivant()* qui permet de connecter 2 noeuds avec leurs informations comme la durée ou la distance.

Pour créer ces méthodes, nous avons repris le modèle de notre professeur et ses 4 classes (une implémentation qu'il avait réalisé) en suivant les méthodes qu'il conseillait de créer, comme *CalculDetails(Etape etape)*, qui permet de chercher tous les noeuds adjacents à partir d'un noeud courant, ou encore la méthode *ChercherPrécédent(Noeud1 noeud, List<Etape> list)*, qui renvoie la réponse finale en cherchant le plus court chemin avec un parcours dans le sens inverse d'étapes trouvées. Puis nous avons implémenté Dijkstra en suivant l'algorithme donné par le cours ainsi que nos connaissances et les cours de "Algo & Graphe" de l'année dernière. Et enfin, nous avons créé les

méthodes afin d'afficher le graphe et à partir de la liste d'étapes, de renvoyer la durée totale du trajet sous la forme d'un décimal (ex : 2,5 pour 2h30), ou encore la distance totale en km.
Toujours dans cet outil commande, nous avons implémenté du code pour modifier une commande ou la consulter à l'aide de notre liste static de commande.

Aussi, pour ajouter des nœuds à notre liste de nœuds, nous les ajoutons directement dans la console. En effet, comme nous sommes l'un sur Mac et l'autre sur Windows, nous avions eu des problèmes de lecture du fichier .csv et donc il était compliqué d'avancer, donc nous avons décidé de partir sur un ajout directement dans le code. De même pour les clients et pour le stockage des commandes et des salariés. Nous avons également rencontré un problème similaire lors de l'implémentation des tests unitaires, tests que sous Mac, nous ne pouvons pas réaliser.

- Module Salarié :

Le module salarié fonctionne de la même manière que le module Client en termes de liste vu qu'il est également lié à une liste et à une interface pour les calculs de Moyenne, par exemple.

En plus de cela, nous avons également joint une partie arbre n-aire pour l'affichage de l'organigramme.
En effet, pour ce faire, nous avons une classe Arbre, qui contient comme seul attribut une racine de type Noeud(salarie) qui contient donc la racine de l'entreprise, à savoir le PDG. En plus de cette classe, nous avons évidemment une classe Noeud qui contient comme attributs un salarié et une liste de nœuds "enfants" qui correspondent donc aux subordonnés d'un directeur (par exemple le directeur des opérations et ses "enfants" les chauffeurs). Nous utilisons une méthode *AjouterSousDirecteur()* pour créer les liens hiérarchiques de l'entreprise afin de créer l'arbre. Une fois tous les nœuds créés et connectés (création que nous faisons dans le main), tous les nœuds sont connectés à la racine avec les bons liens hiérarchiques et nous pouvons afficher l'arbre.
Nous avons également des méthodes afin de retirer ou d'ajouter un nouveau salarié dans l'organigramme lorsque l'on a son supérieur. D'ailleurs pour l'ajout de salariés, nous avons décidé de ne pouvoir embaucher que des chefs d'équipe et des chauffeurs. Et aussi, si l'on supprime un noeud de l'arbre, tous ses enfants sont supprimés de l'arbre, c'est un choix que nous avons fait.

- Module Autre :

Pour le module autre, nous avons décidé d'implémenter 4 méthodes :

- *Afficher le CA de l'entreprise* : c'est une méthode qui retourne l'ensemble de l'argent qui a été dépensé par les clients depuis le début.
- *Afficher les livraisons à venir pour un chauffeur* : qui renvoie pour un chauffeur donné les dates des livraisons qu'il doit effectuer, les prochaines livraisons.
- *Ajouter une ville* : qui permet d'ajouter une ville au graphe, et donc une nouvelle adresse de livraison pour les commandes.
- *Ajouter une nouvelle connexion entre 2 villes :* qui permet de renvoyer un lien entre 2 villes, un nouvel itinéraire à notre choix d'itinéraire qui sera pris en compte par Dijkstra.