

Département de g

énie informatique et génie logiciel

**LOG 2810 STRUCTURES DISCRETES**

**Rapport de laboratoire #2 TP2 : AUTOMATE**

|  |  |
| --- | --- |
| éléments évalués | Points |
| Qualité du rapport : respect des exigences du rapport, qualité de la présentation des solutions | 2 |
| Qualité du programme | 3 |
| Composants implémentés |  |
| C1 | 3 |
| C2 | 3 |
| C3 | 2 |
| C4 | 3 |
| C5 | 2 |
| C6 | 2 |
| Total | 20 |

Soumis par

Alexandre THIMONIER 1782235

Freddy SOSSA 1784114

Timothée LABORDE 1782257

Le 2017-11-27

# Introduction

Au cours de ce travail, nous aurons à mette en application les notions étudiées en classe. Il s’agit entre autres de ceux concernant la construction et le fonctionnement des automates. Il s’agira dans le cadre de ce TP, de traiter des requêtes de livraison soumises et de s’assurer d’optimiser la livraison des colis. Pour cela, il faut en effet construire un automate et valider les requêtes soumises.

**Compréhension du sujet :**

Le projet vise plus particulièrement à effectuer des livraisons de colis par des drones et à optimiser les livraisons en tenant compte du nombres cycles effectués pour les livraisons ainsi que le choix de drone pour obtenir un temps de livraison minimal, tel un logiciel de localisation et de géo-positionnement par satellite. Pour cela nous disposons d’un choix parmi deux types de drones à savoir les drones à faible capacité et ceux à grande capacité.

Chaque type de drone possède une limite de poids en ce qui concerne ce qu’il est capable de transporter. Les drones du type I sont en effet en mesure d’effectuer des livraisons de colis de colis dont le poids est inférieur à 1 Kg tandis que ceux du deuxième type avec une meilleure capacité, peuvent transporter au maximum des colis de 5 Kg. Chaque déplacement du drone prend un cycle (livraison incluse) et le nombre de drones est limité et il faut en tenir compte. Une contrainte apparait, il s’agit d’effectuer les livraisons en utilisant le moins de cycles possibles et en envoyant le moins de drones possibles.

Nous recevons des requêtes (de type source, destination, poids) et nous devons valider les requêtes en s’assurant de la validité des adresses entrées et de vérifier si le poids du colis est supportable par les drones mis à notre disposition.

Aussi, il sera utile de produire une interface utilisateur, un « menu », à partir duquel l’usager pourra choisir de construire l’automate, de traiter les requêtes et d’imprimer les statistiques. Cependant s’il choisit un index non valide, nous devons afficher une erreur et retourner au menu sans quitter.

# Présentation des travaux : Traitement des requêtes

La première étape du travail est de lire un fichier et d’enregistrer les informations lues. En fait, l’utilisateur nous fournit un fichier contenant l’ensemble des codes postaux de la ville dans laquelle on travaille. A partir de ce fichier, on construit un automate qui est capable de reconnaître tous les codes postaux contenus dans le fichier. Cet automate sera réutilisé plus tard pour valider les requêtes qui seront reçues. La construction de cet automate a nécessité la création des classes Etat et Arbre. La classe Etat représente un état de l’automate. Un état possède un parent, des enfants et une sortie. La classe Arbre représente en soi l’automate qui possède comme attribut un état racine qui est une chaîne vide. A partir de la racine, on peut accéder aux états suivants qui seraient ses enfants. Si on prend par exemple la chaîne « H3T », l’enfant immédiat de la racine serait un état avec la valeur H. C’est cette classe qui permet de sauvegarder tous les codes postaux. Elle est caractérisée par une méthode ajouterCodePostal() qui permet comme son nom l’indique d’ajouter un nouveau code postal à l’automate.

Lors de la lecture du fichier, la méthode creerArbreAdresses() permet de créer l’automate en soi. Dans cette méthode, on rappelle la méthode ajouterCodePostal qui rajoute chaque code postal lu par le fichier

serai la plus appropriée, car c’est effectivement dans un graphe que nous allons sauvegarder les nœuds. Une seconde classe apparaît alors en tandem, la classe Nœud, ou Node en anglais. La classe Graphe comportera des nœuds (agrégation), nous commencerons donc par celle-ci.

Nous savons qu’un nœud a besoin d’un numéro, soit un entier, d’une indication pour la station de recharge (celle-ci ne peut qu’exister ou non, nous choisissons donc un booléen) et potentiellement d’un nom (une string). À ces attributs nous devons ajouter des « getters » et des « setters » pour obtenir et changer les attributs. Une fois la classe nœud créée, nous passons à Graphe. De même nous savons qu’elle a besoin d’un tableau de nœuds, une matrice de poids (soit d’entiers) et un chemin parcouru (un autre tableau de nœud). Concernant les méthodes, on peut imaginer que les tableaux seront de tailles variables et il nous faudra donc allouer de la mémoire, grâce à une méthode allocate(). Ensuite le cœur de la classe, une méthode creerGraphe() qui comme son nom l’indique permettra de faire un graphe à partir d’un nom de fichier passé en paramètre, attention cependant, **ce fichier doit être placé dans le même dossier que celui contenant les fichiers .Java.** De plus afin de faciliter l’affichage par la suite il serai aviser de produire une méthode lireGraphe() qui affiche le graphe tel qu’il est présenté à l’annexe 2.a et elle utiliserai la méthode afficherVoisin() qui comme son nom l’indique affiche les voisins de chaque nœud.

Enfin nous pouvons créer la classe Dijkstra, qui a pour but d’exécuter l’algorithme du même nom. Celle-ci contiendra une matrice contenant les poids d’origine (et elle demeurera inchangée : matriceSource), une seconde matrice qui va évoluer au cour de l’exécution :

matriceDijkstra, un nœud de début et un nœud de fin et enfin le parcours suivit. Pour exécuter l’algorithme nous allons avoir besoin d’initialiser (ce qui donne la méthode du même nom). Ensuite il faudra trouver la distance minimale et doTnc le nœud suivant, mettre à jour la matrice et enfin réaliser le tout jusqu’à atteindre l’arrivée. Pour s’assurer du bon fonctionnement nous sommes d’abords passé au travers du cours puis d’un pseudo code écrit à la main.

Par la suite nous avons pris conscience qu’il nous fallait établir un chemin optimisé (PlusCourtChemin()) qui doit prendre une décision en fonction des temps de parcours et des drones. Ceci nous ajoute donc une classe Drone qui va donc évidement contenir son type, son autonomie maximum, l’état de sa batterie et son type de colis. Nous allons devoir créer des accesseurs pour cette une nouvelle classe ainsi qu’une méthode de calcul de la batterie en fonction du temps. Temps qui va justement permettre de gérer notre prise de décision, il apparait donc clairement que nous avons besoin de variables de temps, temps maximal sans recharge au cours du parcours, temps total du parcours et temps depuis la dernière recharge.

Finalement nous pouvons obtenir le diagramme de classe suivant :

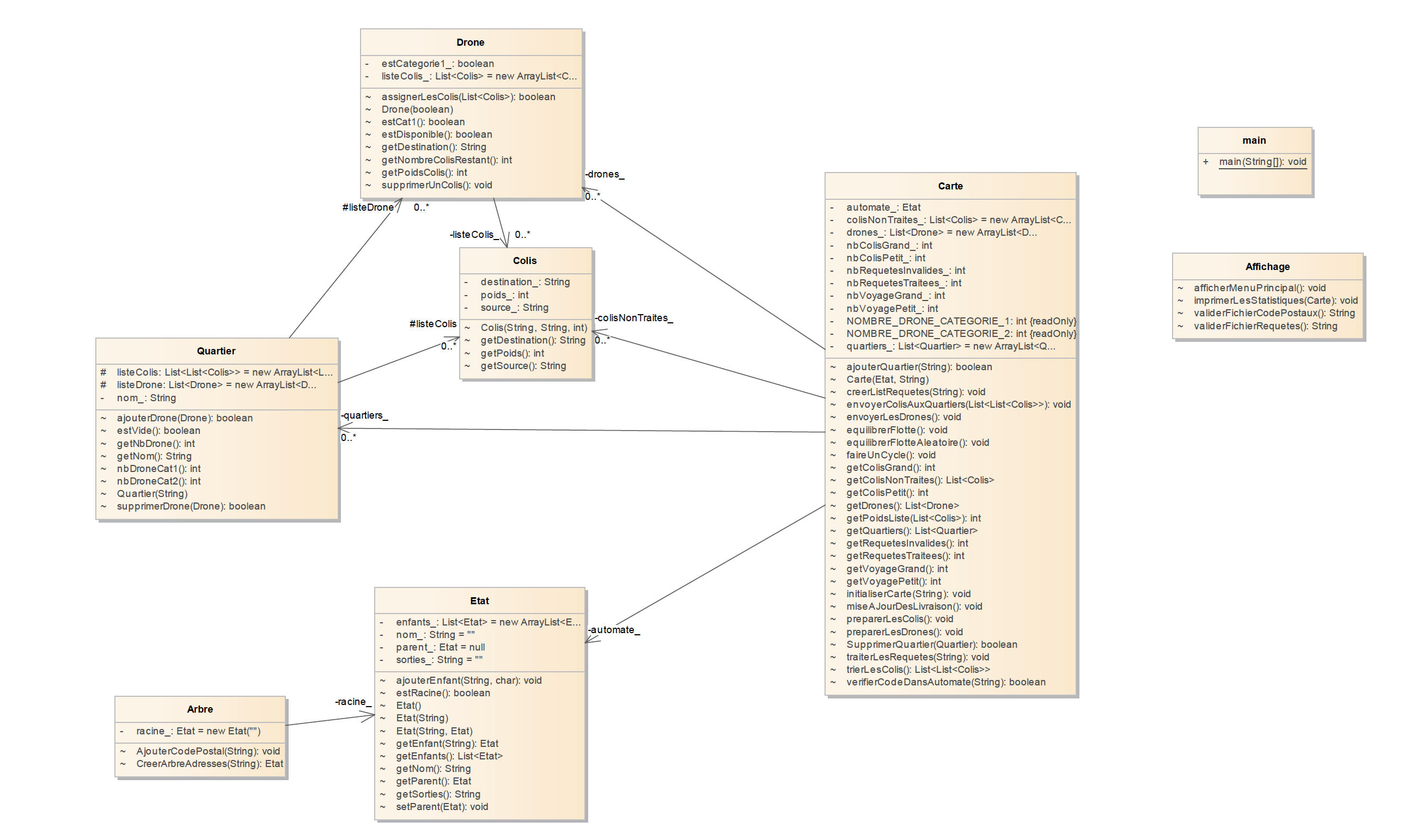


Figure [1] : Diagramme de classe

# Présentation de nos travaux : Déjeuner et dessert

La première étape est de lire et d’enregistrer les données obtenues à partir de la lecture du fichier fourni pour initialiser les attributs. Afin de bien représenter les différents éléments à manipuler, il était nécessaire de créer des classes pour faciliter la gestion et l’encapsulation des données. A cet effet, nous avons créé deux classes : Ingredient et GrapheOriente.

La Classe Ingredient représente un Ingredient qui est caractérisé par deux informations : le nom de l’ingrédient et l’indice qui lui est associé. Quant à elle la classe GrapheOriente représente le Graphe en soit, contenant ainsi la matrice des arcs, les ingrédients utilisés, le nombre de chemins possibles sur le Diagramme de Hasse et la liste de tous les chemins possibles du diagramme de Hasse.

En effet, la matrice des arcs est en deux dimensions permet d’initialiser toutes les données. On enregistre donc tous les arcs entre les Nœuds. Si un arc existe entre deux nœuds (de i vers j), alors on insère à la position [i][j] la valeur 1 et -1 à la valeur [j][i] afin de respecter l’orientation des arcs. Cette matrice est la base pour réaliser toutes les tâches suivantes à savoir l’affichage du graphe Orienté et la génération du diagramme de Hasse.

Pour l’affichage du Graphe orienté, l’idée est assez simple. Il suffit de :

* Parcourir le tableau des ingrédients
* Récupérer chaque ingrédient
* Aller vérifier les ingrédients avec lesquels il est en relation et
* Afficher ces ingrédients dans le format demandé

La méthode CreerGrapheOriente de la classe GrapheOriente se base sur cet algorithme pour afficher le graphe Oriente des Ingredients.

En ce qui concerne l’affichage du diagramme du Hasse (méthode genererHasse( ) ) l’algorithme utilisé est le suivant :

* Identifier tous les minimaux du Diagramme du Hasse
* Parcourir chaque minimal et chercher les nœuds qui sont juste après lui. Cette partie est faite de façon récursive afin de réduire la longueur du code. Ainsi, sur chacun des nœuds suivants identifiés, on reprend la procédure jusqu’au maximaux tout en tenant compte des différentes règles du diagramme de Hasse
* Enregistrer au fur et à mesure qu’on parcourt les différents nœuds la liste des chemins
* Afficher les différents chemins obtenus au format demandé

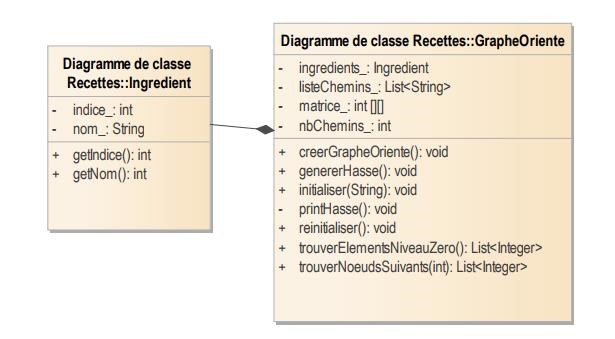


Figure [3] : Diagramme de classe de la partie Déjeuner et Desserts

# Difficultés et solutions : Livraison de colis par drone

## • Difficultés rencontrées

* Nous avions mal anticipé la fin de cette partie, principalement la prise de décision
* Remanier les calculs de temps et de batterie
* Gêne de tableaux dynamiques avec allocation
* Problème de lecture de fichiers

## • Solutions potentielles

* Remanier la prise de décisions par des cas dans la classe graphe plutôt que dans Dijkstra
* Utiliser la batterie plutôt que le temps, mais finalement nous avons décomposé en plusieurs variables avec des noms très descriptifs pour ne pas se perdre.
* Nous avons réalisé trop tard qu’en allouant en multipliant la capacité par deux posait des problèmes de calculs, donc nous avons optés pour une allocation de type capacité +1 à chaque fois mais nous aurions alors plus simplement utiliser des vecteurs.
* Remplacer les chemins absolus par des chemins relatifs

# Difficultés et solutions : Déjeuner et dessert

## • Difficultés rencontrées

Pour la réalisation du diagramme de Hasse :

* Intégration des règles de conception du diagramme de Hasse dans le code
* Difficulté de débogage avec les fonctions récursives

## • Solutions potentielles

* Créer de nouvelles méthodes pour limiter le travail effectué par la fonction récursive
* Intégration progressive des règles du diagramme de Hasse au code (une fonction traite juste la réflexivité et une autre la transitivité)

# Présentation de nos travaux : Déjeuner et dessert

Au cours de ce projet nous avons pût mettre en place les notions de LOG2810 telles que l’algorithme de Dijkstra, diagramme de Hasse et les graphes mais aussi des notions de LOG1000 et de LOG2410. En effet pour la première fois nous partions de rien, nous n’avions pas de projet et de programmes fournis. C’était donc très satisfaisant de voir fonctionner notre projet alors qu’il est partit d’une feuille blanche.

De plus cela nous a permis de nous rappeler quelques notions de SSH, notamment celles de la gestion du temps et de la répartition du travail, quelques heures de préparations qui se sont avérées très utiles par la suite.

Concernant le temps alloué pour ce laboratoire, nous estimons avoir passé une soixantaine d’heures de travail en groupe accompagnées d’une dizaine d’heure personnelles. Ceci semble raisonnable pour une échéance de 3 trois semaines pour un groupe organisé.

Nos attentes concernant le prochain travail pratique sont une liberté de projet équivalente et un rapport temps disponible et temps de travail équivalent.