M1 INFO
KURDYK Louis
KALOUSDIAN Leah

Projet de Programmation Orientée Objet

en Java

Professeur
Khaddouja ZELLAMA

1. Choix effectués

1.1. Héritage

Les objets représentant le contenu de la base de données héritent d'une classe permettant d'identifier le type d'objet considéré que ça soit un client ou bien un véhicule.
Ainsi, les voitures et les scooters sont des classes filles d'une classe véhicule. Il en va de même pour les particuliers et les entreprises qui sont représentés par des classes filles d'une classe client.
Ce choix permet aux différents contrôleurs d'interface graphique de JavaFX de ne pas différencier les classes filles au moment de l'affichage, ainsi que de faciliter les ajouts dans la base de données via la création de nouveaux véhicules par une seule méthode.

1.2. Polymorphisme

Du polymorphisme est utilisé pour différencier l'affichage d'une voiture de celui d'un scooter, idem pour les particuliers et les entreprises. Cela concerne la méthode toString() réécrite dans chacune des classes filles. L'existence d'une méthode toString() dans les classes mères a permis d'effectuer des tests dans les premières journées de développement.

1.5. Design pattern : patron monteur

Pour construire des objets complexes avec plusieurs configurations possibles, nous avons utilisé la notion de design pattern, en particulier le patron monteur.
Dans cet exemple, le patron monteur permet de construire différents modèles de document pdf.

Interface

Nous avons la classe **Director**, qui avec ses méthodes de construction, contrôle l'ordre de celle-ci. Le directeur sait quelles méthodes appeler pour produire un modèle de document précis. Il manipule les monteurs uniquement à travers leur interface commune appelée **PdfBuilder**, ce qui permet de passer différents types de monteurs au directeur.

Classe abstraite

Dans notre cas, nous avons 3 documents à produire : un certificat d'immatriculation, un certificat de cession et un bon de commande. En regardant les modèles de certificat réels, nous avons remarqué que les certificats d'immatriculation et de cession étaient similaires, donc avons décidé d'implémenter une classe commune appelée **Certificate**. Cette classe hérite de la classe abstraite **PdfDocument** qui a trois attributs : une commande (**Order**), un chemin d'accès relatif (path), et un titre.

Objets monteurs

La classe **Certificate** est accompagnée de son objet monteur, **CertificateBuilder**. Le bon de commande contient les mêmes éléments que les certificats, mais dans un style différent, donc aura son propre patron monteur : **PurchaseBuilder**.

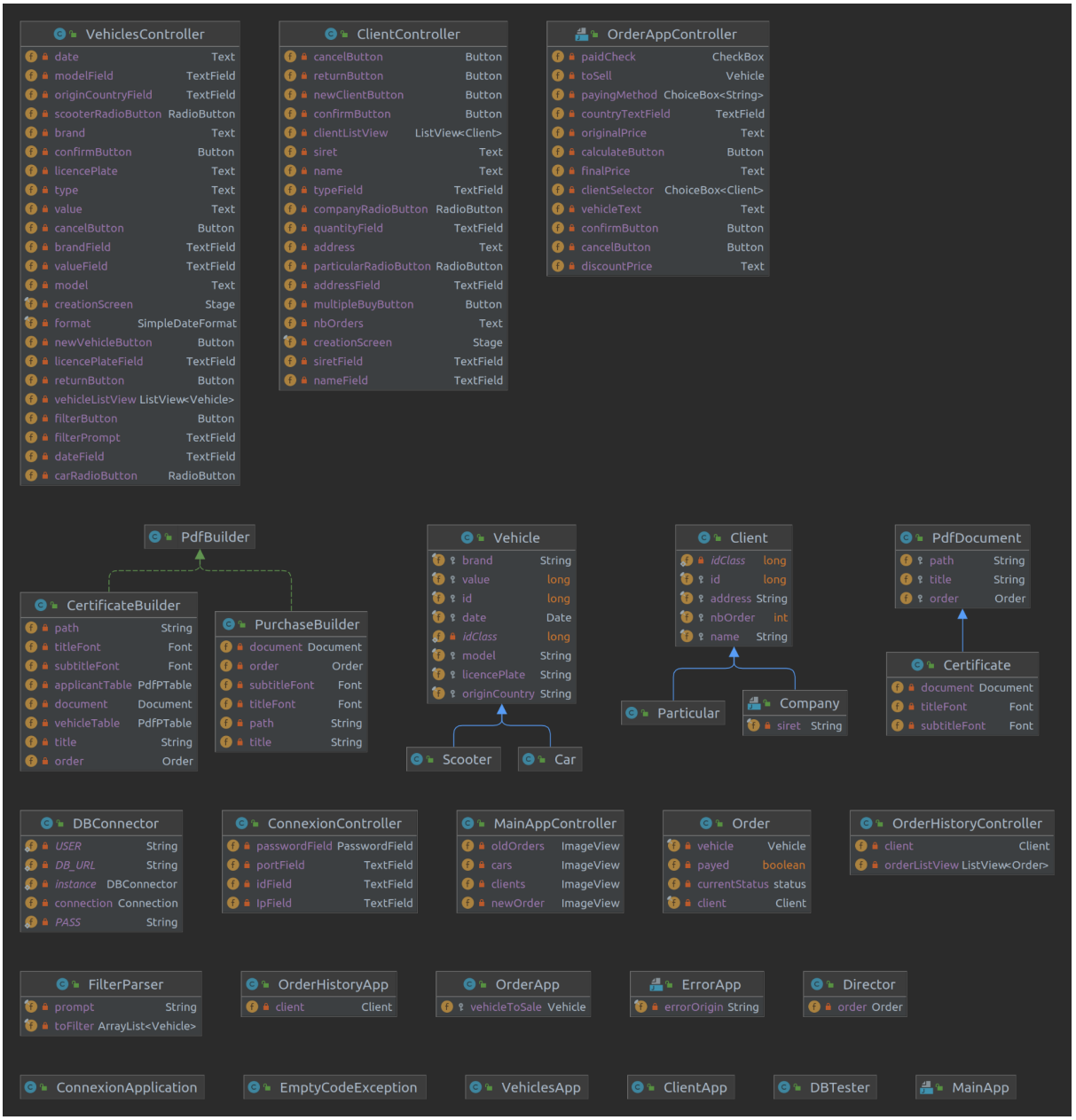
Les objets monteurs, ici **PurchaseBuilder** et **CertificateBuilder**, récupèrent le résultat final, car le directeur ne connaît pas le type du produit construit. Seul l'objet monteur sait exactement ce qui est en train d'être monté.

1.6 Design pattern : Singleton

La classe permettant l'interaction entre le logiciel et la base de données utilise un patron singleton. Cela permet d'assurer l'existence d'une unique instance de la classe au sein d'une exécution du programme et donc assure que chaque requête est bien effectuée l'une après l'autre en cas de volonté d'ajouter plus de multi-threading.

2. Architecture

Diagramme UML :



Le code est séparé en 3 packages principaux :

- Applications : contient l'ensemble des applications JavaFX ainsi que des contrôleurs nécessaires pour les interactions avec l'utilisateur. Chaque application correspond à une fenêtre lors de l'utilisation est est contrôlée par la classe dans son dossier. Cette architecture a été choisie afin d'avoir un accès simple au fichier .fxml correspondant à chaque fenêtre en utilisant la méthode class.getResource().

- Content : contient l'ensemble des classes permettant d'interagir avec la base de données ainsi que de passer de la représentation de la base de données à une représentation en objet Java pour pouvoir les utiliser avec JavaFX.

- PdfHandler : contient l'ensemble des classes permettant la génération des PDF nécessaires ainsi que leur manipulation.

3. Difficultés rencontrées

3.1 Mise en place du filtre

La mise en place du filtre permettant de sélectionner quels véhicules doivent être affichés selon des caractéristiques données était la partie la plus théorique à implémenter. En effet, comme ce dernier devait pouvoir prendre en compte des disjonctions et des conjonctions logiques, il fallait être en mesure d'établir un ordre de priorité sur les opérateurs "and" et "or". Cela nous a mis sur la piste d'implémenter un semblant de logique propositionnelle avec un système de littéraux et de clauses. Chaque littéral correspond à une expression évaluable en un booléen (ex. Model=305). Les clauses sont des conjonctions ou des disjonctions de ces littéraux (mais pas les deux pour éviter les problèmes de priorité opératoires). Ainsi, le filtre est extrêmement sensible aux demandes n'ayant pas la forme souhaitée (cf. manuel d'utilisation) mais permet de faire des demandes avec un certain niveau de complexité. Ce niveau pourrait être amélioré avec l'implémentation d'une évaluation récursive pour permettre plus de niveau de parenthésage mais cela demanderait un effort de parsing plus important que nous n'avons pas souhaité mettre en place.

3.2 Contrôleurs lié à des objets

L'utilisation de la méthode class.getResource() pour initialiser les FXMLLoader ne permet pas de préciser le fichier du contrôleur du fichier .fxml dans celui-ci, pour les contrôleur ayant besoin d'un argument (comme un véhicule par exemple). Il est nécessaire de retirer cette mention dans le fichier correspondant et de set le contrôleur au lancement de l'application en question dans le code Java. Réaliser ce fait ainsi que trouver la solution décrite nous a pris un certain temps et nous a causé des erreurs d'exécution inattendues.

3.3 Mise en page du document pdf

La librairie itextpdf

Nous avons utilisé la librairie itextpdf pour créer et mettre en page des documents pdf. Il a fallu se familiariser avec la librairie et adapter les fonctionnalités de cette librairie : ses Objets, ses méthodes, la gestion des entrées/sorties et des exceptions à notre cas précis, et tout ceci incorporé dans les différentes classes du patron monteur, en particulier les objets monteurs **CertificateBuilder** et **PurchaseBuilder**.

La méthode permettant de créer le document pdf est setDocument(), et est appelée dans les méthodes de la classe **Director**. Puis, nous procédons à remplir le document avec les objets voulus, et le style voulu, différent pour les types de Pdf.

Pour construire des tableaux contenant les informations voulues sur un objet, comme le véhicule, il a fallu créer un petit algorithme de remplissage qui permettait de remplir un tableau **PdfPTable** d'un certain nombre de colonnes avec nos données.

Redondance évitée

Pour implémenter l'algorithme de remplissage de tableau identique et nécessaire à chaque document pdf, nous avons créé une méthode *default* dans l'interface PdfBuilder. Ainsi, les classes qui implémentent PdfBuilder hériteront de fait de cette méthode.

3.4 Ouverture de plusieurs fichiers simultanément : utilisation de threads

L'ouverture des documents PDF générés nous a demandé quelques expérimentations. En effet, la librairie utilisée nécessite que chaque document soit ouvert dans un thread à part. Le fait de l'ouvrir via un bouton de JavaFX faisait "freeze" l'application et n'affichait aucune exception pour nous permettre de comprendre l'origine du problème. La solution à été trouvée en tâtonnant avec plusieurs solutions avant de comprendre que l'ouverture des documents posait problème au thread de JavaFX et qu'il fallait donc utiliser de nouveaux threads.

3.5 Architecture de la base de données

L'architecture de la base de données présente un défaut lié à un défaut de conception. La table Sales devrait avoir des contraintes de clés étrangères liées à l'acheteur et au véhicule acheté. Cela n'est pas le cas à cause de la séparation entre les tables Cars, Scooters et Particular, Company. Etant donné qu'on ne peut faire une clé étrangère vers Cars ou Scooters, il a été décidé de retirer la contrainte pour ne pas avoir à refaire toute les requêtes SQL, la solution la plus propre étant de fusionner les tables Cars avec Scooters et Particular avec Company et de rajouter un champ pour préciser les types. Prévoir la table de vente plus en amont aurait pu éviter un tel problème.

4. Répartition du travail

La répartie du travail c'est faite par package :

- Applications : Louis Kurdyk
- PdfHandler : Leah Kalousdian
- Content : Louis Kurdyk

5. Fonctionnalités apportées

Les fonctionnalités demandées étaient les suivantes :

1. Concevoir une application de vente de véhicules comme, des automobiles ou des scooters (vous pouvez rajouter d'autres types de véhicules : optionnel). L'application permet d'afficher un catalogue de véhicules proposés à la vente, d'effectuer des recherches au sein de ce catalogue et de passer la commande d'un véhicule.
2. Le système doit gérer les commandes. Il doit être capable de calculer les taxes en fonction du pays de livraison du véhicule. Il doit également gérer les commandes payées au comptant et celles assorties d'une demande de crédit. Le système gère les états de la commande: en cours, validée et livrée.
3. Lors de la commande d'un véhicule, le système construit la liasse des documents nécessaires comme la demande d'immatriculation, le certificat de cession et le bon de commande. Ces documents doivent être disponibles au format PDF ou HTML.
4. Le système permet également de solder les véhicules difficiles à vendre, à savoir ceux qui sont dans le stock depuis longtemps.
5. Il permet également une gestion des clients, en particulier des sociétés possédant des filiales afin de leur proposer, par exemple, l'achat d'une flotte de véhicules.
6. La recherche dans le catalogue peut se faire à l'aide de mots clés ou d'opérateurs logiques (et, ou).

Etat de chaque fonctionnalité avec la partie du manuel d'utilisation correspondante :

1. Implémentée, voir partie III
2. Implémentée, voir partie V et VIII
3. Implémentée, voir partie V et VIII
4. Implémentée, voir partie V
5. Implémentée, voir partie VI
6. Implémentée, voir partie III

A cela s'ajoute la création d'une base de données avec le script fourni, la connexion à une base de données distante (cf. partie I du manuel).

6. Expérimentations

6.1 Algorithme

6.1.1 Filtrage

En ce qui concerne l'élaboration de l'algorithme de filtrage du catalogue, une première version à base d'arbre a été envisagée. Cette-ci aurait permis l'évaluation d'une requête à partir des feuilles (qui aurait été les littéraux) jusqu'à remonter à la racine en passant par les nœuds internes représentant les “and” et “or”. Cette version aurait pu être

plus rapide que celle implémentée à l'aide de thread pour évaluer chaque branche de l'arbre. Cette solution posait problème vis-à-vis de l'instauration d'une priorité opératoire entre "and" et "or" qui soit logique. Ainsi, la solution implémentée était plus simple à implémenter dans le temps restant et présentait moins de risque de comportement inattendu.