DJERIAN NOEMIE
DUGOURD CLEMENT
GIL KILLIAN

SPECIFICATION: PROJET DE QUALITÉ DE DEV

DECEMBRE 2022

NOM DU PROJET:

GARAGE DE L'IUT

UNE APPLICATION EN:

JAVA

ETUDE DE CONCEPTION

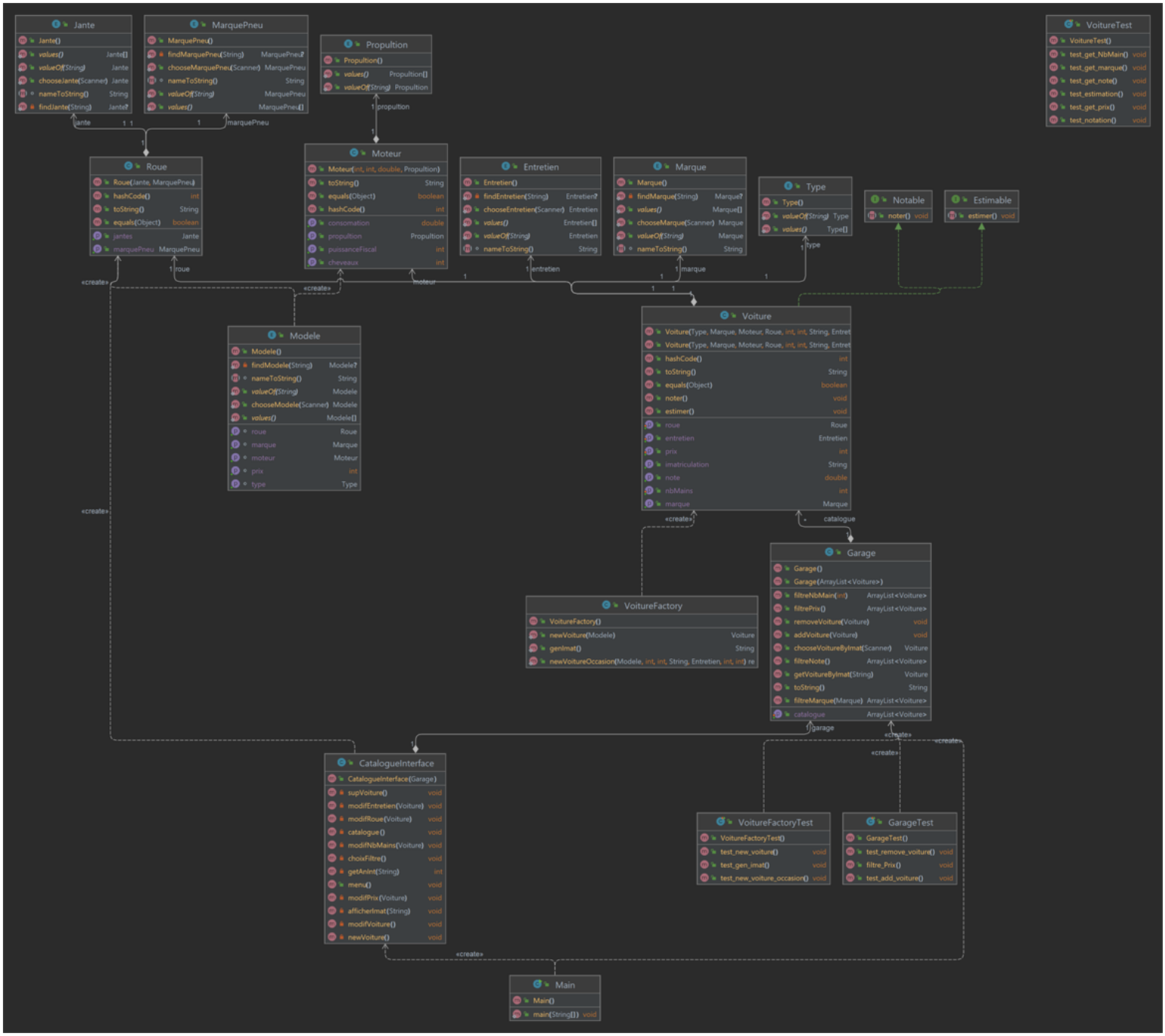
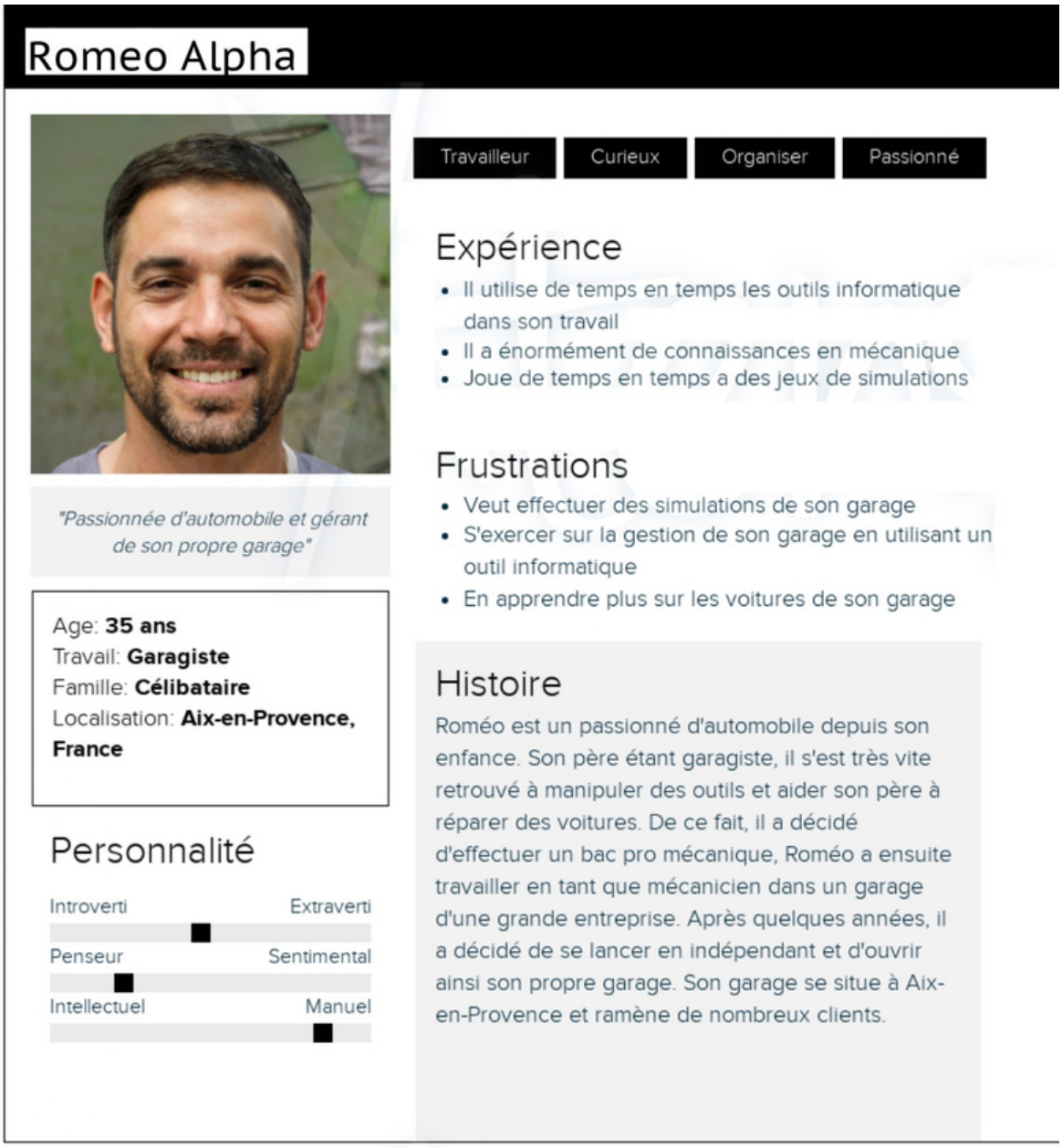


DIAGRAMME DE CLASSE DU PROJET

LE PROJET ENTIER DISPONIBLE SUR GITHUB :
HTTPS://GITHUB.COM/DUGOURDCLEMENT/PROJETQUALITEDEV

ETUDE DE CONCEPTION



UN PERSONA PERMETTANT DE SE METTRE À LA PLACE DU CLIENT AFIN DE MIEUX APPRÉHENDER SES BESOINS

ETUDE DE CONCEPTION

USE CASE

L'APPLICATION

S Y N T H È S E :

Pour le sujet de notre projet de qualité de développement, nous avons décidé d'implémenter une application de simulation de Garage, utilisant les concepts que nous avons pu aborder tout au long des précédents cours durant cette matière. Nous avons respecté la norme de programmation donnée en cours. La partie « graphique » de notre application est minimaliste, l'affichage et en effet en ligne de commande sur le terminal. Nous avons jugé plus optimal de faire comme ceci pour plus nous concentrer sur le code et ne pas perdre de temps avec l'aspect esthétique du JavaFX. Le concept de notre application est simple, gérer un garage : consultation de catalogue, ajout, modification et suppression de voiture.

Pour le projet, nous avons dû réfléchir au préalable à toutes les classes, interfaces, dépendances, enum nécessaires au bon fonctionnement du jeu, soit les spécifications minimales de l'application. Ainsi, il nous a suffi d'implémenter toutes les caractéristiques d'un garage, de quoi est-il composé, comment sont composé les voitures qui y résident, gérer le catalogue comprenant tous les véhicules disponibles ainsi que leurs propriétés, …

- Le garagiste peut consulter le catalogue de voitures disponibles dans le garage. Il a aussi la possibilité de filtrer le catalogue par prix, marque, note et nombre de mains
- Le garagiste a la possibilité de modifier le prix, l'entretien, les roues et le nombre de mains d'une voiture
- Le garagiste a la possibilité de supprimer une voiture

Nous avons rajouté un système de notation des voitures, qui permet d'attribuer automatiquement une note à une voiture selon 3 caractéristiques : l'entretien, le kilométrage et le nombre de mains.
Le garagiste est donc libre d'utiliser l'application pour simuler une réelle gestion de garage !

L'APPLICATION

BILAN TECHNIQUE :

Pour ce projet d'application de simulation garage, il nous a fallu au plus utiliser des concepts techniques vu précédemment lors de nos derniers TP. Il était spécifié dans le sujet que l'utilisation des concepts suivant sont requis : algorithmique, structures de données, particularités, langage Java et modèles de conception. L'application doit également comprendre des classes abstraites, des interfaces, des collections, avec des itérateurs, des exceptions, des threads, de la généricité, et au moins un algorithme de tri.

ALGORITHME :

Le garage comprend bien évidemment de nombreux algorithmes tel que des exécutions conditionnelles, des choix multiples (switch case), des boucles for et des boucles while. On s'en sert notamment pour l'ajout et la suppression de voiture dans le garage, les vérifications de syntaxe lors d'une entrée clavier (ex : immatriculation), l'attribution de note/prix selon les caractéristiques de la voiture, …

STRUCTURE DE DONNÉES:

On utilise des tableaux pour filtrer les voitures selon leurs notes, prix, nombre de mains mais aussi pour notre catalogue. Cette structure de donnée de base s'applique lorsqu'un regroupement de nombreuses données homogènes (soit de même type) à lieu : ici le regroupement de nos voitures.

MODELE DE CREATION :

Factory : La fabrique permet de créer un objet dont le type dépend du contexte. Avec notre classe VoitureFactory, on crée une voiture neuve ou d'occasion, rendant l'instanciation des voitures plus flexible que l'opérateur d'instanciation new.

L'APPLICATION

Builder : Le Monteur est un patron de conception de création qui permet de construire des objets complexes étape par étape. Nous nous en somme par exemple servi lors de la génération des plaques d'immatriculation de nos voitures avec un stringBuilder (permet d'ajouter à cette instance la représentation sous forme de chaîne d'un objet spécifié). En fonction de sa position dans la plaque, le string Builder ajoute un random (lettre ou chiffre) à la String genImat.

Façade : Le but est de proposer une interface facilitant la mise en œuvre d'un ensemble de classes généralement regroupées dans un ou plusieurs sous-systèmes. Nous nous servons du modèle Façade dans catalogueInterface. En effet, dans son fonctionnement le fait de modifier une voiture déclenche tout une succession de fonction en arrière-plan. Cela s'apparente au modèle d'une Façade.

Itérateur : est un patron de conception comportemental qui permet de parcourir les éléments d'une collection sans révéler sa représentation interne. On utilise les iterateurs à chaque fois que l'on parcourt nos listes.

TYPES ABSTRAITS:

L'intérêt est que la fonction ne peut être instancié. On s'en sert notamment dans notre classe de type Enum modèle pour les fonctions getMarque, getRoue, getType, getMoteur,… Cette méthode permet de ne pas déclarer le corps de la fonction et de le redéfinir pour chacun de nos enums car ils ont tous un corps qui diffère.

INTERFACE:

L'interface notable qui implémente la fonction noter et Estimable qui implémente la fonction estimer présent dans la classe Voiture. Chaque interface décrit la fonction en fournissant uniquement la signature.

COLLECTIONS :

Dans garage, il y a une collection qui contient toutes les voitures et une qui est modulable (en fonction du filtre choisi par le garagiste). En effet, on utilise une des cinq interfaces de bases des fonctionnalités de collections : List (ici ArrayList).

L'APPLICATION

ITÉRATEURS:

On utilise les itérateurs dans les boucles for each : for (Voiture voiture : garage.getCatalogue())

EXCEPTION :

Nous n'avons pas utiliser les exceptions mais nous avons simulé ces dernières comme par exemple dans la fonction getAndInt qui vérifie que l'entrée clavier soit un entier et qui boucle jusqu'à que ça en soit un en informant à l'utilisateur l'entrée attendu. Pareillement avec le format de la plaque d'immatriculation.

ALGORITHME DE TRI :

On utilise l'algorithme de tri particulièrement lors de la fonction filtreNote présente dans la classe garage. Elle affiche le catalogue de voiture disponible dans le garage de la meilleure note à la moins bonne (ordre décroissant).

PROBLÈMES RENCONTRÉS:

La compréhension de certain concept tel que les enum et leurs nombreuses fonctionnalités.

La pré-implémentation qui consistait à déterminer quel attribut méritait l'attribution d'une classe particulière. Réfléchir sur les dépendances entre chaque classe, …

Lors de l'implémentation dans le terminal, un problème a émergé : lorsque nous rentrons un int alors que le programme attendait un string, le programme planter instantanément. Alors une fonction intermédiaire à été implémenter pour vérifier que les entrées clavier correspondait bien au type attendu.

La recherche des caractéristiques de toutes les voitures à été très longue et as pris beaucoup de temps.

L'APPLICATION

MESURES D'AMEILORATIONS:

Afin que notre jeu soit plus complexe et enrichi, nous avons réfléchi à plusieurs possibilités de d'améliorations. Pour commencer, nous pensons qu'une interface graphique peut rendre le jeu plus attrayant car il est vrai que l'affichage de toutes les voitures avec les toString() dans le catalogue peut sembler un peu désagréable à la lecture. Le JavaFx aurait pu permettre à « Garage de L'IUT » d'avoir un aspect plus dynamique et intuitif. Or, le temps imparti ne nous as pas permis de mettre en œuvre cette possibilité.

Aussi nous avons penser à ajouter d'autre type de véhicule tel que des deux roues. Cela aurait enrichi notre catalogue et n'aurais pas réduit le garagiste qu'à la gestion unique des voitures.

Enfin, la possibilité majeure d'amélioration et celle qui permettrai à l'utilisateur de choisir s'il est garagiste ou bien simplement visiteur du garage. En effet, le garagiste à accès à des options tel que l'ajout modification et suppression, chose qu'un simple client ne pourrait avoir accès.