Projet Informatique
Location de Véhicules

Rapport

Auteurs :
Ilyann Aragon
Paul Renaud

Date de rendu : 10 mai 2025

ENSTA
Année universitaire 2024–2025

Table des matières

Description générale du problème

Introduction

Ce projet vise à développer un système de réservation de véhicules permettant à des clients de formuler des demandes selon des critères précis (type, volume, nombre de places, etc.) et d'obtenir une location adaptée, ou un véhicule de catégorie supérieure ou égales en cas d'indisponibilité. Le système proposera également une interface dédiée au client pour effectuer ses recherches et réservations de manière autonome.

Côté loueur, l'application permettra de gérer le parc de véhicules, suivre la disponibilité, calculer les coûts de possession, éditer des factures PDF, et analyser les performances annuelles afin d'optimiser le parc. Une interface graphique permettra au loueur de traiter les demandes, suivre l'activité, et consulter un indicateur de rentabilité (chiffre d'affaire,etc).

Le projet sera réalisé en deux phases : une première centrée sur la logique métier, et une seconde sur les interfaces utilisateur (IHM) pour le client et le loueur.

Discussion sur les différentes pistes envisagées

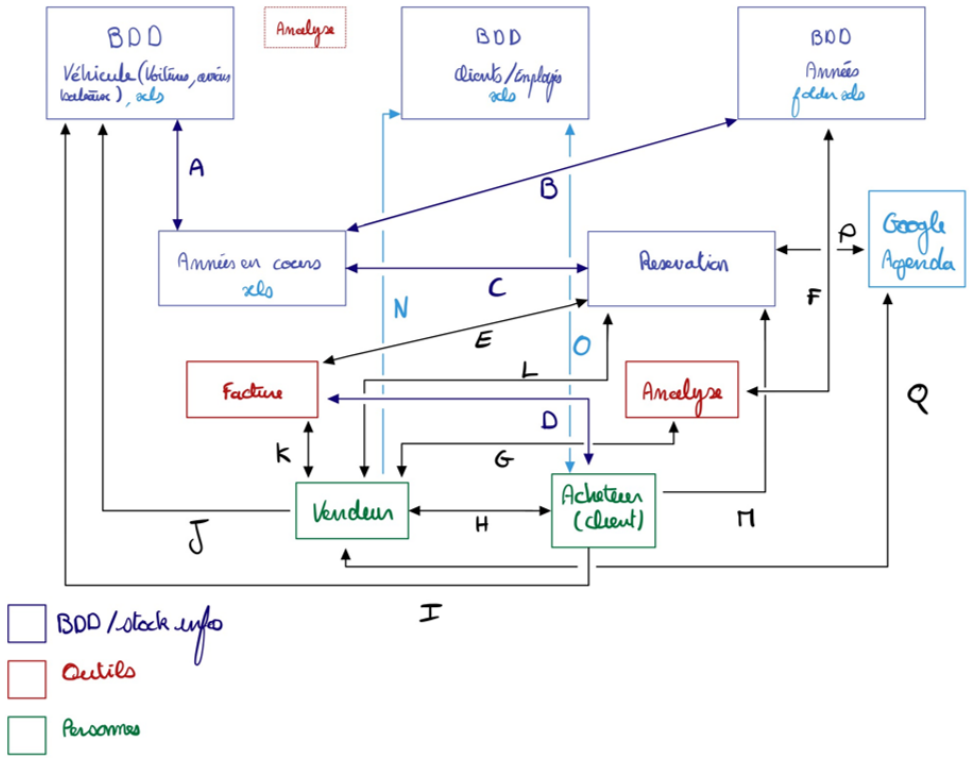

FIGURE 1 – Architecture simplifiée du système de location de véhicules

Résumé des fonctionnalités du système

Le système de location développé couvre l'ensemble du cycle de gestion, depuis la réservation jusqu'à l'analyse historique des données. Les principales fonctionnalités sont détaillées ci-dessous.

Disponibilité des véhicules et utilisation
Le client peut consulter les véhicules disponibles selon ses dates. Le système vérifie les disponibilités en temps réel.

Stockage des données dans le dossier BDD des années passées
À la fin de l'année, les réservations sont archivées automatiquement dans une base dédiée à l'historique.

Ajout des réservations à la BDD en cours
Chaque réservation validée est enregistrée immédiatement dans la base de données active.

Facturation et génération de documents
Une facture PDF est générée automatiquement à partir des informations de réservation.

Transformation des réservations en factures complètes
Le système intègre les données clients pour automatiser la production de factures détaillées.

Analyse des données sur plusieurs années
Les performances ou tendances d'utilisation peuvent être comparées entre années via des indicateurs visuels.

Visualisation graphique (Matplotlib)
Les données sont affichées sous forme de graphiques clairs facilitant l'analyse rapide.

Communication client-vendeur (SAV)
Plusieurs canaux sont prévus : email, téléphone, ou contact direct pour le service après-vente.

Consultation du catalogue par le client
Un moteur de recherche avancé permet de filtrer les véhicules selon les critères du client.

Gestion du catalogue par les vendeurs
Les employés peuvent ajouter, supprimer ou modifier un véhicule et son statut.

Factures accessibles au client
Les factures sont téléchargeables dans l'espace personnel du client.

Réservations côté client
Les clients disposent d'un espace dédié pour effectuer eux-mêmes leurs réservations.

Réservations par un employé pour le compte d'un client
Les vendeurs peuvent aussi enregistrer une réservation manuellement pour un client.

Ajout de clients
Un compte peut être créé par le client lui-même ou par un employé, selon le contexte.

Google Agenda (abandonné)
Initialement prévue, la synchronisation automatique avec Google Agenda a été abandonnée au profit d'un système local unique et centralisé pour plus de robustesse.

Consultation du planning de réservation
Une vue globale des réservations est disponible pour les vendeurs via l'interface de gestion.

Dans un souci de simplification et de robustesse, l'intégration de Google Agenda pour la synchronisation automatique des réservations a été abandonnée. Bien que cette fonctionnalité puisse apporter un confort de visualisation, elle impose des contraintes techniques supplémentaires, notamment en termes de gestion d'API, d'authentification OAuth2, et de dépendances extérieures.

Initialement, le système séparait les réservations en deux fichiers : une base pour l'année en cours et une autre pour les années passées. Cette architecture, bien que logique dans une approche archivistique, alourdissait la gestion des accès, notamment pour les recherches ou les comparaisons. Finalement, une base unique a été adoptée. Elle contient l'ensemble des réservations, toutes années confondues, avec un champ supplémentaire indiquant l'année de la réservation.

Hypothèses réductrices

Dans le cadre du développement du projet de location de véhicules, plusieurs hypothèses simplificatrices ont été retenues afin de concentrer les efforts sur les fonctionnalités principales tout en maintenant une cohérence globale du système. Ces hypothèses, bien que réductrices, ont été choisies de manière raisonnée.

Réservation à la journée uniquement
Les réservations sont effectuées par jour complet, sans prise en compte d'horaires précis ou de demi-journées. Ce choix permet de simplifier la gestion des périodes de location et d'éviter les conflits liés aux chevauchements horaires. Cela reflète par ailleurs un fonctionnement courant dans de nombreux services de location classique.

Tarification indépendante de la distance parcourue
Le prix de la location est calculé uniquement sur la base du nombre de jours réservés. Le kilométrage n'est pas pris en compte dans le calcul. Ce modèle tarifaire évite de devoir intégrer un suivi des distances et facilite la logique de facturation, tout en restant réaliste pour un service de location courte durée simplifié.

Coût d'entretien annuel fixe
Chaque véhicule dispose d'un coût d'entretien estimé sur une base annuelle. Cette donnée est utilisée pour estimer la rentabilité globale du parc de véhicules, sans entrer dans le détail des maintenances ponctuelles ou aléatoires. Cela permet de conserver une dimension économique dans le projet, tout en évitant une modélisation trop lourde ou trop incertaine.

Ces hypothèses ont été retenues pour équilibrer la rigueur du projet et sa faisabilité dans le cadre pédagogique imposé. Elles permettent d'aboutir à un système fonctionnel, cohérent, et techniquement maîtrisable dans le temps imparti.

Description de l'application

Présentation générale du programme

Notre application est divisé en plusieurs fichiers (modules) pythons. Un premier module objects.py regroupent toutes les classes utilisées dans notre fichier principal (application.py). Les fonctions que nous utilisons en plus pour ouvrir des fichiers ou réaliser d'autres actions sont regroupées dans un fichier fonction.py pour éviter d'alourdir le fichier application. Nous avons aussi décidé de crée une fonction facture dans un fichier séparé nommé facture.py, nous envisageons de transformer cette fonction en méthode de la classe Réservation. Pour lancer notre application il suffit de se placer dans le fichier python application.py et de lancer le code. Pour lancer les test unitaires il suffit de se placer dans le fichier tests_unitaires.py et de lancer le code.

Description des principales classes et méthodes

FIGURE 2 – Diagramme de classes de l'Application.

Classe `Application`

La classe `Application` représente le noyau logique et fonctionnel de notre plateforme de gestion de location de véhicules. Véritable point d'entrée du système, elle centralise toutes les interactions entre l'utilisateur et les différentes entités métier, qu'il s'agisse de

la gestion des utilisateurs (clients et vendeurs), des réservations ou encore du catalogue de véhicules.

Conçue dans une optique de clarté et de modularité, cette classe pilote l'ensemble des fonctionnalités de la plateforme via une série de menus dynamiques, adaptés au rôle de l'utilisateur connecté. Elle gère ainsi les connexions sécurisées, la navigation dans les options disponibles, et l'accès conditionnel aux différentes actions possibles selon le profil (client ou vendeur).

Parmi les fonctionnalités offertes, on peut citer :

— La consultation, l'ajout, la suppression ou la modification de véhicules dans le parc.
— La gestion des comptes clients, y compris la création de nouveaux profils, la modification des informations ou le changement de mot de passe.
— Le processus de réservation complet, incluant la recherche de véhicules disponibles, la réservation effective, l'annulation ou encore le surclassement automatique si applicable.
— L'analyse des ventes, avec un accès à des bilans détaillés et à la génération de rapports tels que les factures.

Grâce à des méthodes bien segmentées et à une logique de navigation intuitive, la classe `Application` garantit une expérience utilisateur fluide et cohérente. Elle intègre par ailleurs des outils avancés comme la recherche multicritères, la consultation des réservations à venir par véhicule, ou encore un système de surclassement intelligent visant à optimiser l'utilisation du parc.

Enfin, sa structure orientée objet permet une maintenance facilitée et une extensibilité naturelle, rendant la plateforme adaptable aux évolutions futures, tant en termes de fonctionnalités que de volume d'utilisation.

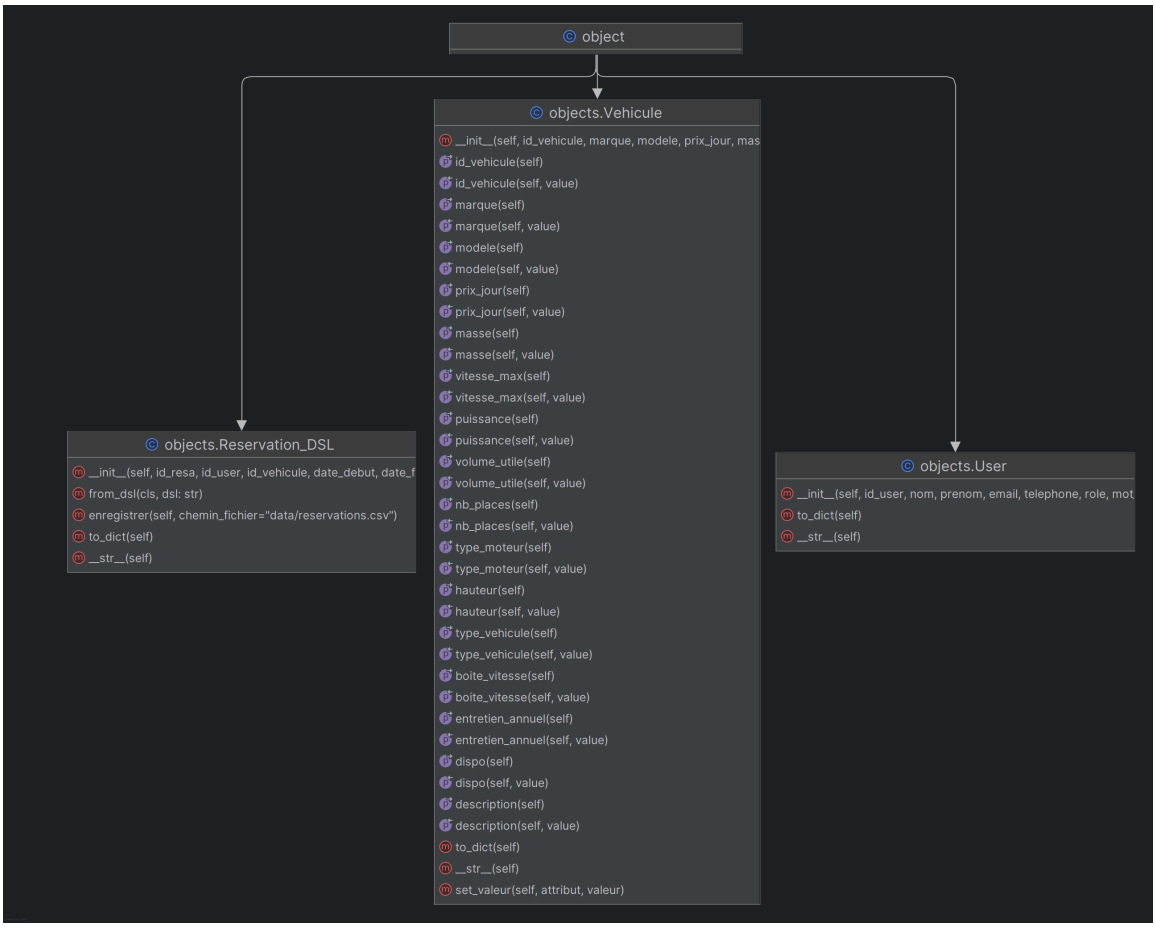

FIGURE 3 – Diagramme de classes des objets : User, Reservation_DSL et Vehicule.

Classe `Reservation_DSL`

La classe `Reservation_DSL` a été développée dans le but de modéliser une réservation de véhicule tout en offrant une interface textuelle flexible grâce à l'utilisation d'un DSL (Domain Specific Language). Inspirée de la classe `Reservation`, elle étend ses fonctionnalités en introduisant une méthode de parsing automatique, permettant de transformer une chaîne de caractères textuelle structurée en une instance de réservation exploitable par le système.

Cette classe remplit deux objectifs principaux :

1. Structurer les données d'une réservation à l'aide d'attributs explicites tels que les identifiants client et véhicule, les dates de réservation, la durée, le prix, ou encore le statut de surclassement.
2. Faciliter l'importation de réservations sous forme de texte grâce à une méthode d'analyse qui interprète un DSL prédéfini.

Par exemple, une réservation peut être exprimée par la chaîne suivante :
`RESERVATION 123456789 CLIENT 987654321 VEHICULE AB-123-CD DU 05-01-2025 AU 05-05-2025 JOURS 3 PRIX 400.00 SURCLASSEMENT True`.
La méthode `from_dsl()` utilise des expressions régulières pour extraire les données pertinentes, puis construit automatiquement un objet `Reservation_DSL`. En cas de format incorrect, une exception informative est levée, renforçant ainsi la robustesse du système.

Outre ce mécanisme de parsing, la classe propose plusieurs méthodes utilitaires :

— `to_dict()` : convertit l'objet en dictionnaire Python pour faciliter son exportation vers un fichier CSV.
— `enregistrer()` : enregistre la réservation dans un fichier CSV, en gérant l'ajout de l'en-tête si besoin.
— `__str__()` : fournit une représentation textuelle claire de la réservation.

Un attribut spécifique de type booléen permet également d'indiquer si un surclassement a été appliqué à la réservation.

En résumé, la classe `Reservation_DSL` allie la rigueur d'une structure orientée objet à la souplesse de l'interprétation textuelle, ce qui la rend particulièrement adaptée pour les interfaces en ligne de commande, les tests automatisés ou l'intégration avec des systèmes externes.

Classe `User` et héritage

La classe `User` représente une entité utilisateur du système, qu'il s'agisse d'un client ou d'un vendeur. Elle centralise les informations essentielles à la gestion des utilisateurs : nom, prénom, identifiant, mot de passe, rôle et coordonnées de contact. Cette structure permet de gérer efficacement l'authentification et la personnalisation des interactions sur la plateforme.

Deux méthodes principales sont proposées :
— `to_dict()` : transforme l'objet utilisateur en dictionnaire Python, facilitant ainsi son enregistrement dans un fichier CSV ou son export vers d'autres systèmes.
— `__str__()` : fournit une représentation lisible de l'utilisateur, sous la forme `"Prénom Nom - Rôle"`, utile pour les interfaces et le débogage.

La classe `User` est conçue pour servir de classe mère à deux sous-classes spécialisées : `Client` et `Vendeur`. Cette structure par héritage permet de mieux séparer les responsabilités et d'introduire des comportements ou attributs spécifiques à chaque type d'utilisateur :
— La classe `Client` pourra intégrer un historique de réservations, des préférences de véhicule, ou encore des statistiques personnelles.
— La classe `Vendeur` gérera quant à elle le suivi du parc automobile, les performances commerciales ou les retours clients.

Grâce à cette hiérarchisation, les méthodes génériques comme `to_dict()` ou `__str__()` sont mutualisées, évitant toute redondance de code et facilitant la maintenance. Cette organisation orientée objet prépare ainsi le système à une évolutivité propre et maîtrisée.

Classe `Vehicule`

La classe `Vehicule` constitue l'un des éléments fondamentaux du système de gestion de location automobile. Elle a été conçue pour modéliser de manière complète et précise chaque véhicule présent dans le parc, en encapsulant l'ensemble des données nécessaires à sa gestion, sa réservation, et son affichage pour les clients.

Chaque instance de la classe représente un véhicule unique, identifié par une plaque d'immatriculation au format `AA-123-AA`, utilisée comme identifiant principal dans les fichiers ou bases de données. Outre cette clé d'identification, de nombreux attributs viennent décrire le véhicule de manière détaillée :
— Informations commerciales : marque, modèle, prix journalier.
— Caractéristiques techniques : masse, puissance, vitesse maximale, volume utile, nombre de places.

— Spécificités mécaniques : type de motorisation (essence, diesel, hybride, électrique...), type de boîte de vitesses (manuelle ou automatique).
— Informations complémentaires : dimensions, coût d'entretien annuel, description textuelle, disponibilité.

La classe propose un constructeur complet ainsi qu'une méthode d'affichage conviviale pour présenter les informations à l'utilisateur. Elle inclut également un ensemble de getters et setters robustes, intégrant des mécanismes de validation afin de garantir la cohérence et l'intégrité des données manipulées.

Parmi les fonctionnalités avancées, on trouve :
— `to_dict()` : une méthode de conversion automatique de l'objet en dictionnaire Python, facilitant son enregistrement dans un fichier CSV ou son intégration à une base de données.
— `set_valeur()` : une méthode générique permettant de modifier dynamiquement n'importe quel attribut de l'objet, tout en effectuant les vérifications nécessaires à la validité de la modification.

Grâce à cette structuration rigoureuse, la classe `Vehicule` offre une grande souplesse tout en assurant la fiabilité du système. Elle constitue une brique logicielle centrale dans l'architecture du programme, et joue un rôle clé dans la sélection, la réservation, et la maintenance des véhicules au sein de la plateforme.

Table des figures imposées et choisies

Figure choisie : Domain Specific Language (DSL) pour les réservations

Nous avons choisi d'implémenter un DSL (Domain Specific Language) pour représenter chaque réservation sous forme de chaîne de caractères structurée. Chaque ligne encode toutes les informations essentielles (identifiants, dates, véhicule, tarif, etc.). Ce choix permet une lecture humaine facilitée (dans le fichier application.py), un traitement automatique via des expressions régulières et une gestion simplifiée dans un fichier texte, sans base de données. De manièrre générale, le DSL permet d'exprimer des concepts métier de manière directe, compréhensible par des non-développeurs ou des experts du domaine.

Figure choisie : Fonction récursive

Nous utilisons une fonction récursive dans notre code : la méthode surclassement dans la classe Application. Cette méthode permet de surclasser un utilisateur en fonction de ses critères entrés pour une recherche de véhicule. La première itération va proposer un surclassement au client avec le même type de véhicule que celui souhaité initialement. Si cela ne lui convient pas, la méthode est appelé mais le client choisit un autre type de véhicule pour le deuxième surclassement. Si le surclassement ne luit convient pas le client peut recommencer un nombre indéfini de fois jusqu'à ce qu'il trouve le véhicule qu'il lui convienne. Le client peut aussi choisir d'arrêter le processus de surclassement à tout moment.

Figure choisie : Gestion des utilisateurs via des comptes dédiés (permissions,...)

Nous avons décidé de créer un compte pour chaque utilisateur car cela est plus simple et cela permet au client de mieux gérer ses réservations (annulations, prise de réservations,etc). Cela permet aussi d'empêcher les clients d'avoir certaines permissions que les vendeurs possèdent (modification des BDD, ajout de véhicule,etc). Cette démarche permet aussi d'optimiser la création de réservation en ne récupérant uniquement les informations du véhicule (celles du client étant déjà renseigné dans son compte).

Tests effectués

Dans le cadre du développement de notre application de gestion de location de véhicules, nous avons mis en place une série de tests unitaires en utilisant le module `unittest` de Python. Ces tests visent à garantir la fiabilité des classes principales et à détecter toute régression lors des évolutions du code.

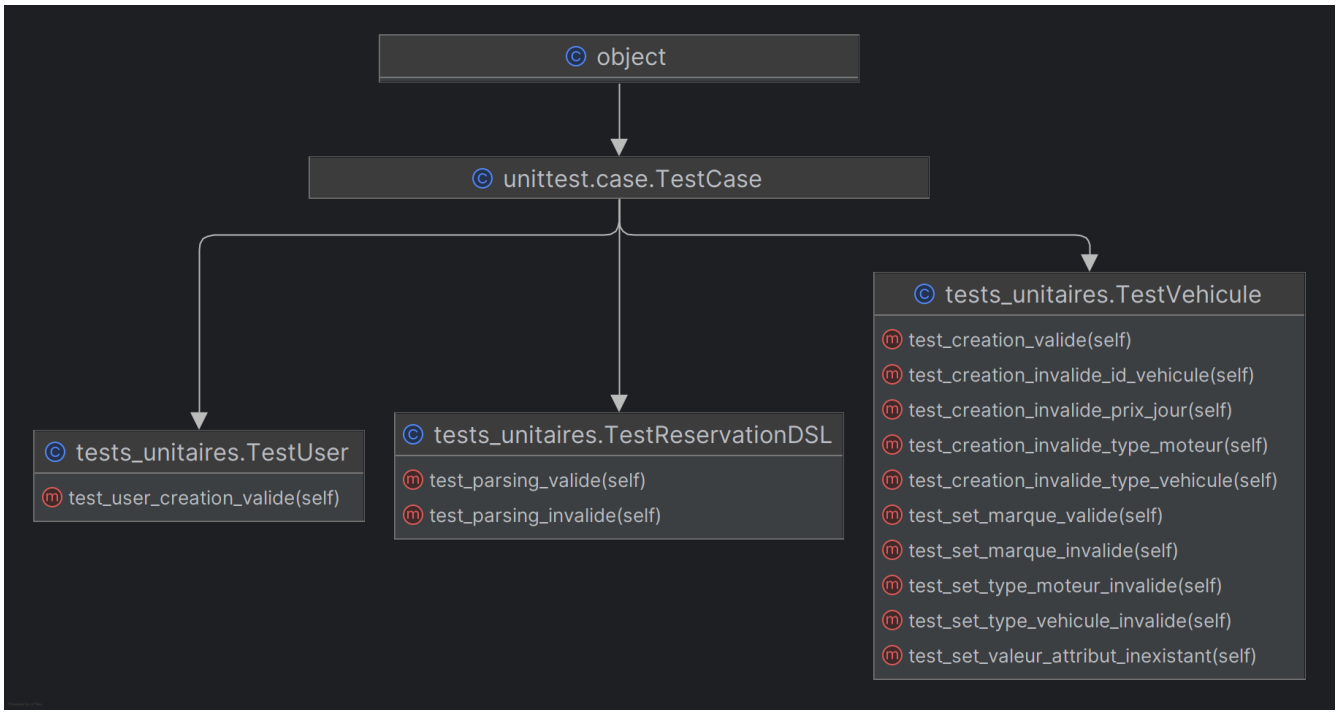

FIGURE 4 – Diagramme de classes des tests unitaires.

0.0.1 Tests de la classe `User`

Un test de création d'utilisateur valide a été implémenté afin de vérifier que les attributs sont correctement initialisés. Un second test de validation d'identifiant incorrect a été commenté pour le moment, mais pourra être réactivé dès que la vérification stricte des identifiants sera intégrée à la classe.

0.0.2 Tests de la classe `Reservation_DSL`

La classe de parsing `Reservation_DSL`, qui permet d'interpréter une ligne de réservation écrite en DSL (Domain-Specific Language), a été testée dans deux cas :

— Un cas valide où toutes les informations sont correctement extraites et converties.

— Un cas invalide où le format ne respecte pas les contraintes du parsing, ce qui déclenche une erreur.

0.0.3 Tests de la classe `Vehicule`

La classe `Vehicule` est la plus testée actuellement, car elle repose sur de nombreuses validations internes :

— Test de création avec des attributs valides.
— Tests de création échouant sur des paramètres incorrects (*id* mal formé, *prix* négatif, *type de moteur* ou *type de véhicule* invalide, etc.).
— Tests de setters permettant de changer dynamiquement certaines valeurs (par exemple la marque ou le type de moteur) tout en validant leur format.
— Utilisation de la méthode `set_valeur` pour tester des cas de modification dynamique générique, y compris les tentatives sur des attributs inexistants.

0.0.4 Perspectives

Les tests couvrent actuellement les principales erreurs attendues lors de la création ou de la manipulation d'objets. Ils permettent de renforcer la robustesse du modèle de données. Cependant, d'autres fonctionnalités restent à tester dans les classes comme `Client` et `Vendeur` quand elle seront codées et fonctionnelles.

Limitations

Apports personnels

Gestion des utilisateurs via des comptes dédiés

Bien que les consignes du projet ne spécifiaient pas la nécessité de gérer des comptes utilisateurs distincts, nous avons choisi d'introduire cette fonctionnalité pour améliorer la modélisation du système. Nous avons ainsi créé des comptes client et comptes vendeur, permettant une gestion plus fine des actions spécifiques à chaque type d'utilisateur.

Chaque utilisateur dispose maintenant d'un compte associé à ses propres attributs et comportements, ce qui permet de mieux séparer les responsabilités. Le compte client est responsable des réservations et de la gestion de ses besoins, tandis que le compte vendeur gère le parc de véhicules, les disponibilités et les réservations effectuées.

Cette approche permet de réduire les conditions imbriquées et d'exploiter le polymorphisme, facilitant l'évolution du système. En outre, elle améliore la modularité et la flexibilité du projet, offrant une meilleure évolutivité en cas d'ajout de nouveaux rôles comme un administrateur.

Ainsi, même si la gestion des utilisateurs n'était pas explicitement demandée dans les consignes, cette amélioration s'inscrit dans une logique de développement de l'application tout en respectant les bonnes pratiques de la programmation orientée objet.

Perspectives et améliorations