Rapport du Projet : Application de Gestion de Parc Automobile

1. Contexte et Objectif du Projet

Ce projet a pour objectif de développer une application console permettant à une entreprise de location de gérer efficacement son parc de véhicules. L'application doit fournir une interface de gestion pour ajouter des véhicules, gérer les clients, louer et retourner des véhicules, et afficher les véhicules disponibles. Le projet s'appuie sur les concepts de la programmation orientée objet (POO) : héritage, polymorphisme, encapsulation et gestion des exceptions.

2. Architecture de l'Application

L'architecture de l'application repose sur une structure modulaire et extensible, conçue pour séparer les responsabilités de chaque entité du système. Les principales entités sont les suivantes :

- Véhicule : Classe abstraite représentant les attributs de base d'un véhicule.
- Voiture et Camion : Sous-classes de `Véhicule`, ajoutant des attributs spécifiques aux véhicules respectifs.
- Client : Classe pour stocker les informations d'un client et les locations en cours.
- ParcAutomobile : Classe centrale de gestion du parc de véhicules.
- Louable : Interface définissant les actions de location et de retour pour les véhicules.
- Exceptions Personnalisées : Gestion des cas d'erreurs spécifiques pour assurer la fiabilité du système.

Cette structure vise à faciliter l'extension de l'application, par exemple en ajoutant de nouveaux types de véhicules, en modifiant les règles de gestion des locations, ou en ajoutant des fonctionnalités supplémentaires sans affecter la base du code.

3. Concepts POO Utilisés dans l'Application

3.1 Encapsulation

L'encapsulation est utilisée pour protéger les données internes des classes et empêcher les modifications accidentelles. Les attributs des classes sont définis en privé (`private`) ou

protégé (`protected`), et ne sont accessibles qu'à travers des méthodes spécifiques, appelées **getters** et **setters**.

- Par exemple, dans la classe `Véhicule`, l'attribut `estLoue` est modifié exclusivement via les méthodes `louer()` et `retourner()`, ce qui garantit que le statut du véhicule est toujours cohérent et maintenu à jour.

3.2 Héritage et Abstraction

L'héritage et l'abstraction permettent de structurer le code pour que les classes spécialisées puissent réutiliser les attributs et méthodes de classes plus génériques.

- La classe `Vehicule` est abstraite et contient les attributs communs à tous les véhicules (immatriculation, marque, modèle, etc.), tandis que les classes dérivées **Voiture** et **Camion** ajoutent des attributs spécifiques (comme `nombrePlaces` pour les voitures et `capaciteChargement` pour les camions).

- La méthode abstraite `calculerPrixLocation()` dans `Vehicule` force les sous-classes à définir leur propre manière de calculer le prix de location, selon leurs spécificités.

3.3 Polymorphisme avec l'Interface `Louable`

Le polymorphisme est implémenté par l'interface **Louable**, qui déclare les méthodes `louer()` et `retourner()`. Cela permet de traiter les instances de `Voiture` et `Camion` de manière uniforme lorsqu'elles sont référencées par l'interface `Louable`.

- En appliquant le polymorphisme, l'application peut gérer différents types de véhicules de manière homogène dans les fonctions de gestion des locations. Cela simplifie le code et permet de modifier le comportement de chaque type de véhicule sans impacter les autres parties du programme.

4. Gestion des Exceptions et Validation

4.1 Exceptions Personnalisées

Des exceptions personnalisées sont créées pour gérer des situations d'erreurs spécifiques à la location de véhicules, offrant une gestion fine des erreurs :

- VehiculeIndisponibleException : Levée si un véhicule est déjà loué, empêchant ainsi une location multiple et incohérente.

- ClientNonAutoriseException : Levée si un client ne dispose pas des autorisations nécessaires pour louer un certain type de véhicule, comme un camion nécessitant un permis spécifique.

Ces exceptions améliorent la lisibilité et la robustesse du code, permettant de signaler des erreurs précises et de garantir une meilleure gestion des processus de location.

4.2 Validation des Données Utilisateur

Les données saisies par l'utilisateur sont vérifiées avant d'être traitées par le système pour garantir la cohérence des informations. Par exemple :

- Avant de louer un véhicule, le programme vérifie que le véhicule est disponible et que le client possède un permis valide pour le type de véhicule demandé.

- Les informations du client, telles que le numéro de téléphone et le numéro de permis, sont vérifiées pour minimiser les erreurs de saisie.

Ces validations renforcent l’intégrité des données et permettent de réduire les erreurs d'utilisation.

5. Menu Interactif pour la Gestion du Parc Automobile

L'application dispose d'un menu interactif accessible par la console, permettant à l'utilisateur de réaliser plusieurs actions de gestion du parc :

- Ajout de véhicules et de clients : L’utilisateur peut ajouter de nouveaux véhicules dans le parc ou créer de nouveaux clients avec leurs informations personnelles.

- Location et retour de véhicules : Les véhicules peuvent être loués et retournés facilement, le statut de chaque véhicule étant mis à jour en conséquence.

- Affichage des véhicules disponibles : L'utilisateur peut afficher une liste de tous les véhicules disponibles ou loués pour une gestion plus intuitive du parc.

Le menu assure une navigation claire et permet une utilisation simple et fluide de l'application.

6. Bonus : Gestion des Contrats de Location (Optionnel)

Une extension potentielle de l'application serait d’ajouter une **classe ContratLocation** pour gérer les locations en incluant des informations comme :

- Dates de début et de fin de location : Suivi précis de la période de location.

- Calcul du coût total : Le coût serait calculé automatiquement en fonction de la durée de location et du type de véhicule.

Cette fonctionnalité améliorerait le suivi des locations et permettrait à l'entreprise de mieux gérer les coûts et les retours de véhicules.

7. Analyse des Structures de Données Utilisées

Le choix des structures de données est essentiel pour garantir la performance et la flexibilité de l’application :

- ArrayList: Utilisé pour stocker les collections de véhicules et de clients, offrant une gestion simple des données et un accès rapide aux éléments.

  - Par exemple, la classe `ParcAutomobile` utilise un `ArrayList` pour stocker les véhicules et propose des méthodes de recherche et de filtrage des véhicules (loués ou disponibles).

Cette structure permet une croissance dynamique du nombre d'éléments, ce qui est idéal pour un parc de véhicules dont la taille peut évoluer avec le temps.

8. Conclusion et Perspectives

Ce projet illustre l'utilisation de la programmation orientée objet dans un contexte pratique, combinant une structure de classe modulaire, des interfaces, et une gestion fine des erreurs. L’application permet de gérer un parc de véhicules de manière efficace et extensible.

Points d'amélioration

- Gestion des contrats de location : Enrichir le modèle pour inclure une classe de contrat et un système de suivi de la facturation.

- Persistance des données : Ajouter une base de données ou un système de stockage de fichier pour permettre de conserver les données de véhicules, de clients et de locations entre les exécutions de l'application.

- Interface utilisateur : Évoluer vers une interface graphique pour faciliter l’usage de l’application par les utilisateurs non techniques.

Grâce à cette application, l'entreprise de location pourrait améliorer l'efficacité de sa gestion de parc tout en maintenant une traçabilité et une organisation rigoureuse.