



**ECOLE MAROCAINE DES
SCIENCES DE L'INGENIEUR**
Membre de
HONORIS UNITED UNIVERSITIES



Rapport de projet

3^{ème} année

Ingénierie Informatique et Réseaux

GESTION DE LA FLOTTE DE VEHICULES

Réalisé par :

Abdelkhalek AIT-IDIR & Aymane ADIDI

Encadré par :

Tuteur de l'école : Dr. Mariame AMINE

Introduction

Contexte général

Le projet de gestion de flotte de véhicules vise à optimiser la gestion et l'utilisation des véhicules au sein d'une organisation. Ce système aide à suivre la disponibilité des véhicules, leurs réservations par les employés, et leur kilométrage. De plus, il permet une gestion efficace des véhicules électriques en prenant en compte leur autonomie.

Introduction

Le projet répond au besoin de digitalisation de la gestion des véhicules dans une organisation. Avec une interface simple et conviviale, il permet de réduire les erreurs humaines et d'améliorer l'efficacité.

Objectifs du projet

- Offrir un système pour réserver des véhicules.
- Gérer les détails des véhicules, y compris leur kilométrage et disponibilité.
- Suivre les réservations effectuées par les employés.
- Intégrer la gestion des véhicules électriques.

Technologies utilisées

Le projet est entièrement réalisé en **C++**, avec l'utilisation des bibliothèques standard (par exemple, **iostream**, **vector**, et **string**).

Conclusion

Ce système répond aux besoins organisationnels pour une gestion simplifiée et efficace des ressources véhiculaires.

Analyse des besoins

Introduction

L'analyse des besoins est essentielle pour comprendre et définir les fonctionnalités attendues.

Exigences fonctionnelles

- Permettre l'ajout de nouveaux véhicules à la flotte.
- Enregistrer les employés et leurs réservations de véhicules.
- Mettre à jour les kilométrages des véhicules après usage.
- Afficher les véhicules disponibles et la liste des réservations.

Exigences non fonctionnelles

- **Facilité d'utilisation** : Interface intuitive et simple d'interaction.
- **Performance** : Traitement rapide des réservations et des mises à jour.
- **Maintenabilité** : Code clair, modulaire, et évolutif.

Conclusion

Les exigences fonctionnelles et non fonctionnelles garantissent un système répondant aux attentes des utilisateurs tout en assurant une gestion optimale.

Conception du système

Introduction

La conception inclut des modèles pour chaque composant du système, assurant une implémentation organisée et compréhensible.

Modélisation des classes et objets

Classes principales

1. Vehicule

- **Attributs** : Marque, modèle, kilométrage, disponibilité.
- **Méthodes** : getMarque(), getModele(), setKilometrage(), afficherDetails().

2. VehiculeElectrique (Hérite de Vehicule)

- **Attributs supplémentaires** : Autonomie.
- **Méthodes** : afficherAutonomie().

3. Employe

- **Attributs** : Nom, liste des véhicules réservés.
- **Méthodes** : reserverVehicule(), afficherReservations().

4. Flotte

- **Attributs** : Liste des véhicules et des employés.
- **Méthodes** : ajouterVehicule(), ajouterEmploye(), reserverVehicule().

Schéma relationnel (MCD simplifié)

Employe (ID_Employe, Nom)

Vehicule (ID_Vehicule, Marque, Modele, Kilometrage, Disponible)

VehiculeElectrique (ID_Vehicule [FK], Autonomie)

Flotte (ID_Reservation, Date_Reservation, ID_Employe [FK], ID_Vehicule [FK])

Diagramme des Cas d'Utilisation (UML)

Voici les interactions principales entre les acteurs et les fonctionnalités :

Acteurs :

- **Employé :**
 - Réserver un véhicule.
 - Consulter les véhicules disponibles.
- **Administrateur :**
 - Ajouter un véhicule.
 - Mettre à jour l'état d'un véhicule.
 - Visualiser les réservations.
 - Mettre à jour le kilométrage des véhicules.

Cas d'utilisation :

- Association entre chaque acteur et ses fonctionnalités directes (réserve, ajout, mise à jour).

Surcharge des opérateurs

L'opérateur << a été surchargé pour afficher les détails d'un véhicule de manière concise.

Conclusion

La conception modulaire du système facilite l'implémentation et l'extensibilité future.

Détails d'implémentation

Introduction

L'implémentation traduit les concepts de conception en un code fonctionnel et performant.

Code source

Voici un extrait de chaque classe :

Classe Vehicule

```
// Classe Vehicule
class Vehicule {
private:
    string marque;
    string modele;
    int kilometrage;
    bool disponible;

public:
    Vehicule(const string& marque, const string& modele, int kilometrage, bool disponible)
        : marque(marque), modele(modele), kilometrage(kilometrage), disponible(disponible) {}

    string getMarque() const { return marque; }
    string getModele() const { return modele; }
    int getKilometrage() const { return kilometrage; }
    bool getDisponible() const { return disponible; }

    void setKilometrage(int nouveauKilometrage) { kilometrage = nouveauKilometrage; }
    void setDisponibilite(bool etat) { disponible = etat; }

    void afficherDetails() const {
        cout << "Marque: " << marque << ", Modèle: " << modele
              << ", Kilométrage: " << kilometrage
              << ", Disponible: " << (disponible ? "Oui" : "Non") << endl;
    }

    friend ostream& operator<<(ostream& os, const Vehicule& v) {
        os << "Marque: " << v.marque << ", Modèle: " << v.modele
          << ", Kilométrage: " << v.kilometrage
          << ", Disponible: " << (v.disponible ? "Oui" : "Non");
        return os;
    }
};
```

Classe VehiculeElectrique

```
// Classe VehiculeElectrique
class VehiculeElectrique : public Vehicule {
private:
    int autonomie;

public:
    VehiculeElectrique(const string& marque, const string& modele, int kilometrage, bool disponible, int autonomie)
        : Vehicule(marque, modele, kilometrage, disponible), autonomie(autonomie) {}

    void afficherAutonomie() const {
        cout << "Autonomie: " << autonomie << " km" << endl;
    }
};
```

Classe Employe

```
// Classe Employe
class Employe {
private:
    string nom;
    vector<string> vehiculesReserves;

public:
    Employe(const string& nom) : nom(nom) {}

    string getNom() const { return nom; }
    vector<string> getVehiculesReserves() const { return vehiculesReserves; }

    void reserverVehicule(const string& modele) {
        vehiculesReserves.push_back(modele);
    }

    void afficherReservations() const {
        cout << "Employé: " << nom << ", Réservations: ";
        for (const auto& v : vehiculesReserves) {
            cout << v << " ";
        }
        cout << endl;
    }
};
```

Classe Flotte

```
// Classe Flotte
class Flotte {
private:
    vector<Vehicule> vehicules;
    vector<Employe> employes;

public:
    Flotte() {}

    void ajouterVehicule(const Vehicule& vehicule) {
        vehicules.push_back(vehicule);
    }

    void ajouterEmploye(const Employe& employe) {
        employes.push_back(employe);
    }

    void reserverVehicule(const string& modele, const string& nomEmploye) {
        for (auto& vehicule : vehicules) {
            if (vehicule.getModele() == modele && vehicule.getDisponible()) {
                for (auto& employe : employes) {
                    if (employe.getNom() == nomEmploye) {
                        vehicule.setDisponibilite(false);
                        employe.reserverVehicule(modele);
                        cout << "Réservation réussie: " << modele << " pour " << nomEmploye << endl;
                        return;
                    }
                }
            }
        }
        cout << "Réservation échouée: " << modele << " non disponible ou employé introuvable." << endl;
    }
}
```


Description de l'implémentation

- **Structures de données** : `std::vector` pour gérer dynamiquement les listes de véhicules et d'employés.
- **Encapsulation** : Utilisation d'attributs privés pour garantir la protection des données.
- **Surcharge d'opérateur** : Amélioration de la lisibilité des sorties en formatant les détails des véhicules.

Défis rencontrés et solutions

1. **Vérification des disponibilités** : Utilisation d'attributs booléens pour indiquer l'état du véhicule.
2. **Gestion des réservations conflictuelles** : Une boucle parcourt les employés pour éviter les doublons.

Conclusion

L'implémentation respecte les bonnes pratiques de codage et répond aux besoins spécifiés.

Tests et validation

Introduction

Les tests vérifient si le système fonctionne correctement dans des cas d'utilisation réels.

Scénarios de test

1. Ajout de véhicules et affichage de la flotte.
2. Réservation d'un véhicule disponible.
3. Mise à jour du kilométrage et vérification.

Résultats des tests

Tous les tests ont été concluants :

- Les véhicules ont été ajoutés correctement.
- Les réservations ont été enregistrées sans conflits.
- La mise à jour du kilométrage fonctionne comme prévu.

Conclusion

Les tests confirment que le programme fonctionne comme attendu.

Conclusion & Perspectives

Bilan du projet

Le projet répond aux besoins de gestion de flotte en offrant des fonctionnalités pratiques et efficaces.

Limites du projet

- Les objets Vehicule sont stockés directement dans un vector, ce qui empêche d'exploiter le polymorphisme.
- les objets sont manipulés statiquement sans différenciation par type
- La classe Flotte ne permet pas de libérer correctement la mémoire allouée pour éviter les fuites.
- les employés sont stockés dans un vector, ce qui nécessite une recherche séquentielle.

Perspectives d'amélioration

- Utilisation des pointeurs pour stocker les véhicules dans la classe Flotte, ce qui permet de gérer des objets polymorphiques (Vehicule et VehiculeElectrique) dans un conteneur unique.
- implémentation de polymorphisme avec la méthode virtuelle afficherDetails et la méthode getType, permettant un comportement spécifique pour les véhicules électriques (VehiculeElectrique) tout en traitant tous les véhicules via des pointeurs de type Vehicule.
- Destructeur : La classe Flotte dispose d'un destructeur qui libère correctement la mémoire allouée pour éviter les fuites.
- Les employés sont stockés dans un map avec leur nom comme clé. Cela permet un accès rapide et efficace aux employés pour gérer leurs réservations.
- Une méthode genererRapport qui fournit un aperçu des statistiques globales de la flotte :
 - Nombre total de véhicules.
 - Nombre de véhicules disponibles.
 - Répartition par type de véhicule.

Annexes

Code source complet

Le code source est fourni dans le fichier attaché.

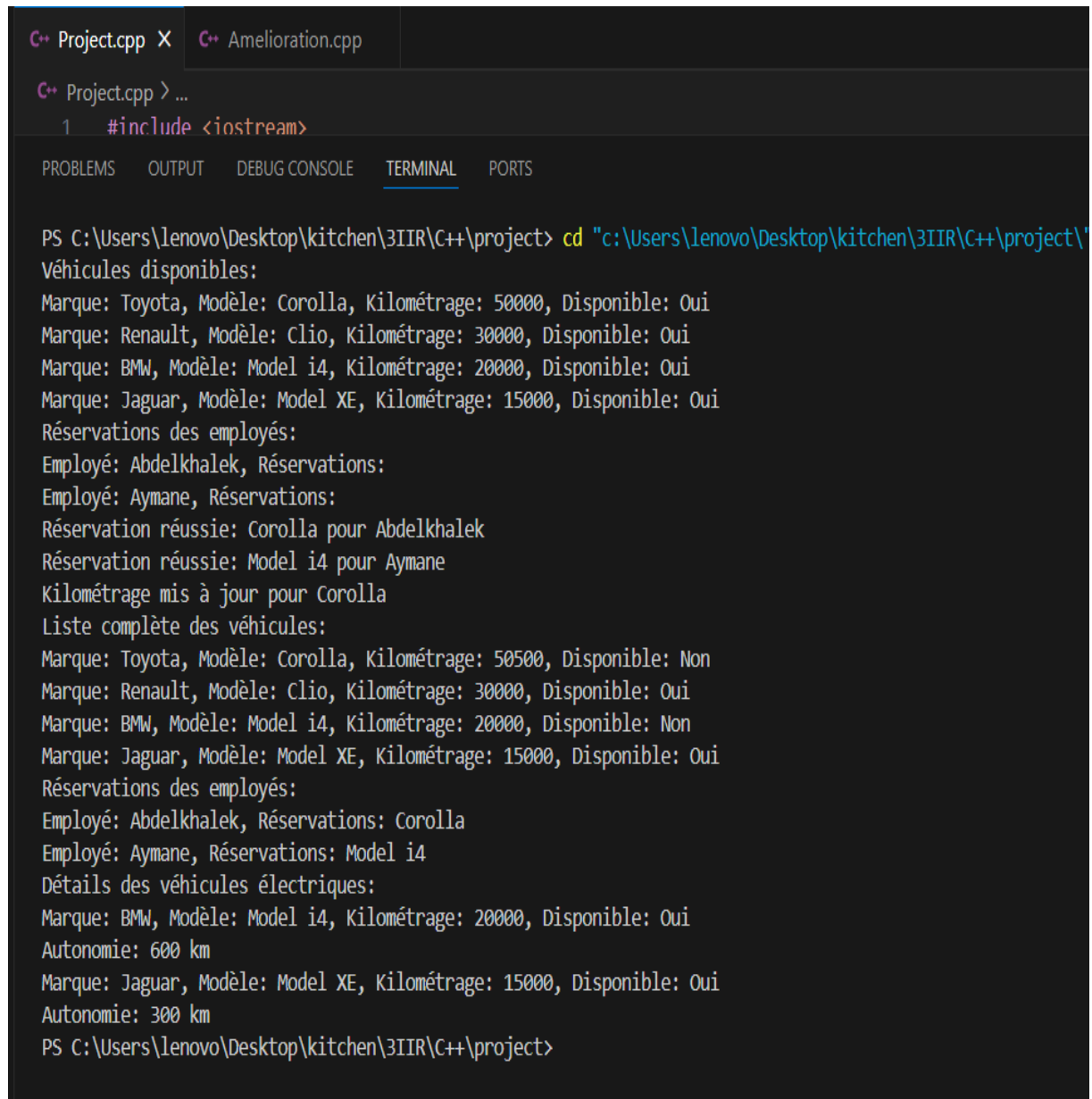
Voici un extrait de la surcharge d'opérateur :

```
friend ostream& operator<<(ostream& os, const Vehicule& v) {  
    os << "Marque: " << v.marque << ", Modèle: " << v.modele  
    << ", Kilométrage: " << v.kilometrage  
    << ", Disponible: " << (v.disponible ? "Oui" : "Non");  
    return os;  
}  
};
```

Voici un extrait de la Mis à jour de kilométrage du véhicule après usage :

```
void miseAJourKilometrage(const string& modele, int nouveauKilometrage) {  
    for (auto& vehicule : vehicules) {  
        if (vehicule.getModele() == modele) {  
            vehicule.setKilometrage(nouveauKilometrage);  
            cout << "Kilométrage mis à jour pour " << modele << endl;  
            return;  
        }  
    }  
    cout << "Véhicule introuvable pour mise à jour." << endl;  
}
```

L'exécution du code source



```
C++ Project.cpp X C++ Amelioration.cpp
C++ Project.cpp > ...
1 #include <iostream>

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\lenovo\Desktop\kitchen\3IIR\C++\project> cd "c:\Users\lenovo\Desktop\kitchen\3IIR\C++\project\"
Véhicules disponibles:
Marque: Toyota, Modèle: Corolla, Kilométrage: 50000, Disponible: Oui
Marque: Renault, Modèle: Clio, Kilométrage: 30000, Disponible: Oui
Marque: BMW, Modèle: Model i4, Kilométrage: 20000, Disponible: Oui
Marque: Jaguar, Modèle: Model XE, Kilométrage: 15000, Disponible: Oui
Réservations des employés:
Employé: Abdelkhalek, Réservations:
Employé: Aymane, Réservations:
Réservation réussie: Corolla pour Abdelkhalek
Réservation réussie: Model i4 pour Aymane
Kilométrage mis à jour pour Corolla
Liste complète des véhicules:
Marque: Toyota, Modèle: Corolla, Kilométrage: 50500, Disponible: Non
Marque: Renault, Modèle: Clio, Kilométrage: 30000, Disponible: Oui
Marque: BMW, Modèle: Model i4, Kilométrage: 20000, Disponible: Non
Marque: Jaguar, Modèle: Model XE, Kilométrage: 15000, Disponible: Oui
Réservations des employés:
Employé: Abdelkhalek, Réservations: Corolla
Employé: Aymane, Réservations: Model i4
Détails des véhicules électriques:
Marque: BMW, Modèle: Model i4, Kilométrage: 20000, Disponible: Oui
Autonomie: 600 km
Marque: Jaguar, Modèle: Model XE, Kilométrage: 15000, Disponible: Oui
Autonomie: 300 km
PS C:\Users\lenovo\Desktop\kitchen\3IIR\C++\project>
```

Code source après l'amélioration

Le code source après l'amélioration est fourni dans le fichier attaché.

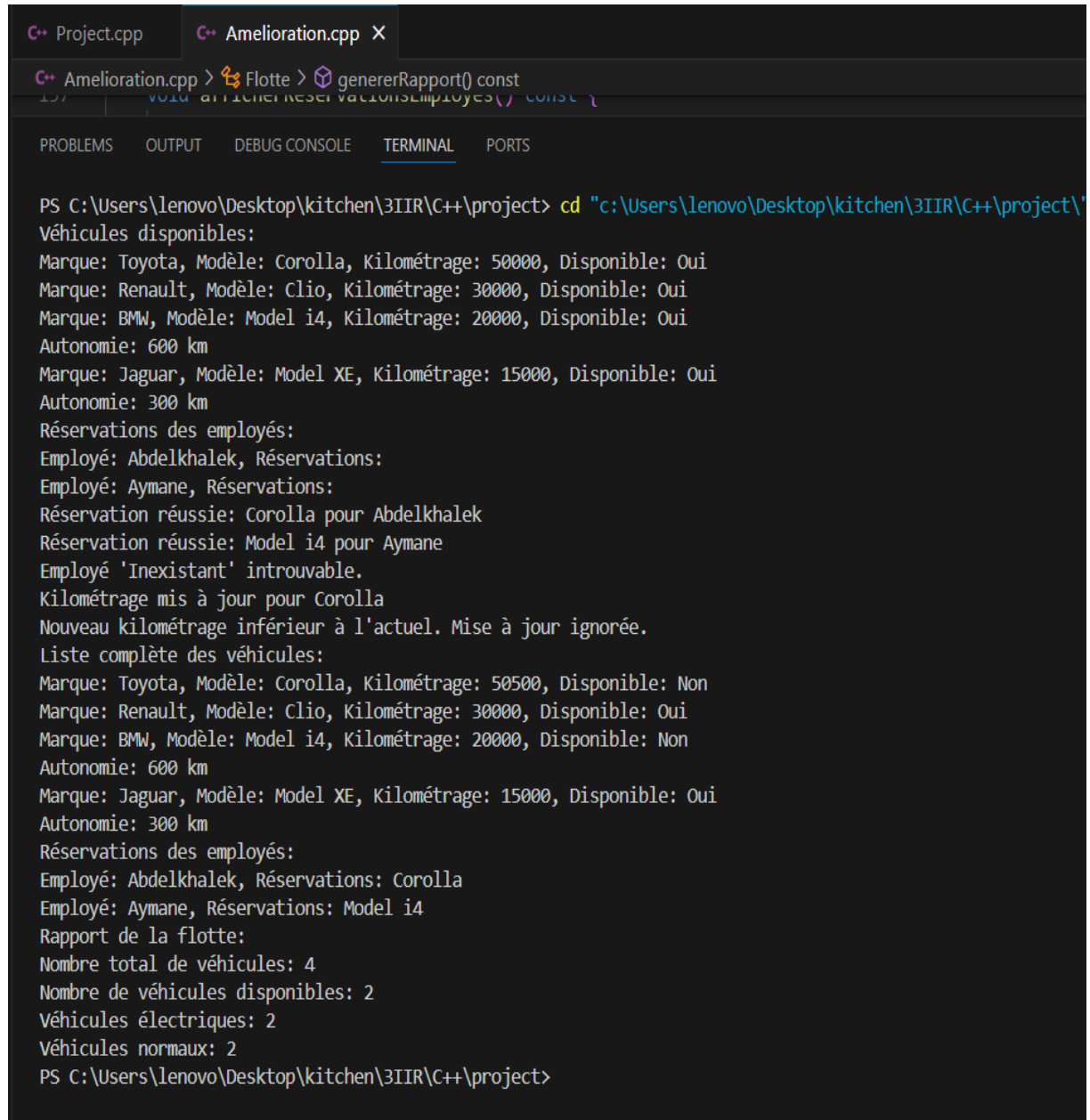
Voici un extrait de la méthode genererRapport:

```
void genererRapport() const {
    int totalVehicules = vehicules.size();
    int vehiculesDisponibles = 0;
    int vehiculesElectriques = 0;
    int vehiculesNormaux = 0;

    for (const auto& vehicule : vehicules) {
        if (vehicule->getDisponible()) {
            vehiculesDisponibles++;
        }
        if (vehicule->getType() == "VehiculeElectrique") {
            vehiculesElectriques++;
        } else {
            vehiculesNormaux++;
        }
    }

    cout << "Rapport de la flotte:" << endl;
    cout << "Nombre total de véhicules: " << totalVehicules << endl;
    cout << "Nombre de véhicules disponibles: " << vehiculesDisponibles << endl;
    cout << "Véhicules électriques: " << vehiculesElectriques << endl;
    cout << "Véhicules normaux: " << vehiculesNormaux << endl;
}
};
```

L'exécution du code source après l'amélioration

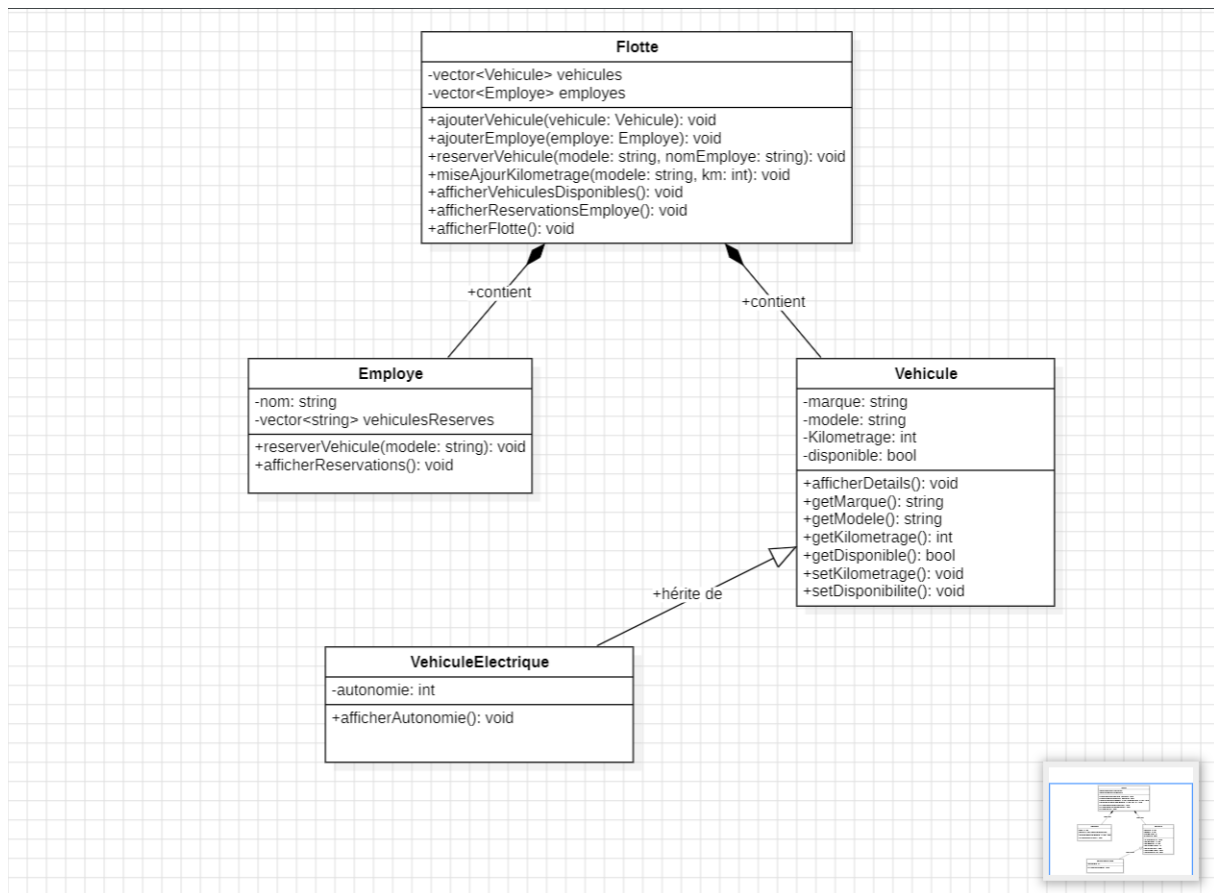


```
C++ Project.cpp C++ Amelioration.cpp X
C++ Amelioration.cpp > Flotte > genererRapport() const
137 // void afficherReservationsEmployes() const {
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\lenovo\Desktop\kitchen\3IIR\C++\project> cd "c:\Users\lenovo\Desktop\kitchen\3IIR\C++\project\"
Véhicules disponibles:
Marque: Toyota, Modèle: Corolla, Kilométrage: 50000, Disponible: Oui
Marque: Renault, Modèle: Clio, Kilométrage: 30000, Disponible: Oui
Marque: BMW, Modèle: Model i4, Kilométrage: 20000, Disponible: Oui
Autonomie: 600 km
Marque: Jaguar, Modèle: Model XE, Kilométrage: 15000, Disponible: Oui
Autonomie: 300 km
Réservations des employés:
Employé: Abdelkhalek, Réservations:
Employé: Aymane, Réservations:
Réservation réussie: Corolla pour Abdelkhalek
Réservation réussie: Model i4 pour Aymane
Employé 'Inexistant' introuvable.
Kilométrage mis à jour pour Corolla
Nouveau kilométrage inférieur à l'actuel. Mise à jour ignorée.
Liste complète des véhicules:
Marque: Toyota, Modèle: Corolla, Kilométrage: 50500, Disponible: Non
Marque: Renault, Modèle: Clio, Kilométrage: 30000, Disponible: Oui
Marque: BMW, Modèle: Model i4, Kilométrage: 20000, Disponible: Non
Autonomie: 600 km
Marque: Jaguar, Modèle: Model XE, Kilométrage: 15000, Disponible: Oui
Autonomie: 300 km
Réservations des employés:
Employé: Abdelkhalek, Réservations: Corolla
Employé: Aymane, Réservations: Model i4
Rapport de la flotte:
Nombre total de véhicules: 4
Nombre de véhicules disponibles: 2
Véhicules électriques: 2
Véhicules normaux: 2
PS C:\Users\lenovo\Desktop\kitchen\3IIR\C++\project>
```

Diagrammes UML

Le diagramme UML représentant les relations entre les classes :



Autres ressources

- Documentation C++ pour les bibliothèques standards utilisées.

Nous avons utilisé les bibliothèques standard de C++ suivantes :

- **iostream** : Pour les entrées/sorties standard, telles que `std::cout` et `std::cin`, permettant l'affichage des informations et la lecture des données saisies.
- **vector** : Une structure de données de la bibliothèque standard utilisée pour gérer dynamiquement des collections, comme la liste des véhicules ou des employés.
- **string** : Fournit la gestion des chaînes de caractères, utilisée pour stocker des informations comme les noms des employés, les marques et modèles de véhicules.

