REPUBLIQUE DU CAMEROUN
Paix-Travail-Patrie
UNIVERSITE DE YAOUNDE I
FACULTE DES SCIENCES
Département d'informatique
B.P.812 Yaoundé
Tél /Fax : (+237) 22 23 44 96

REPUBLIC OF CAMEROON
Peace-Work-Fatherland
UNIVERSITY OF YAOUNDE I
FACULTY OF SCIENCE
Department of computer science
PO.BOX 812 Yaoundé
Tel /Fax : (+237) 22 23 44 96

Filière : Informatique **Niveau** : M1
Unité d'enseignement (UE) : INF 4067 : Pattern Design & UML

Système de Vente en Ligne de Véhicules

Pattern Design et UML

Rédigé par :

Noms & Prénoms	**Matricule**
MAHAMAT SALEH MAHAMAT	21T2423
TCHAMI TAMEN SORELLE	20U2855
TSAKEU NGUEMO MARILYN FLORA	21T2627

Supervisé par :

Dr Valéry MONTHE
Enseignant, Chercheur à l'Université de Yaoundé I

Année universitaire : 2024 - 2025

Sommaire

1 Introduction

L'objectif de ce projet est de concevoir et de mettre en œuvre une application web de vente en ligne de véhicules en appliquant divers **patrons de conception** étudiés dans le cadre du cours **INF 4067**.
Ce système monolithique, développé avec **Spring Boot** pour le backend et **React.js** pour le frontend, doit intégrer des fonctionnalités essentielles telles que :

- La gestion d'un catalogue de véhicules ;
- La prise en charge des commandes et le suivi de leur statut ;
- La personnalisation des options pour les véhicules ;
- La génération de documents liés aux transactions (factures, certificats d'immatriculation) ;
- La gestion des clients et de leurs interactions avec le système ;
- L'authentification et la gestion des accès utilisateurs.

Ce projet permet d'appliquer des concepts avancés d'architecture logicielle et d'ingénierie logicielle, en mettant en œuvre plusieurs patrons de conception pour garantir **la modularité**, **la réutilisabilité** et **la maintenabilité** du système. L'approche suivie repose sur une modélisation UML détaillée avant l'implémentation technique, assurant une conception structurée et évolutive.

2 Architecture Technique et Logicielle

Le système repose sur une **architecture monolithique**,structurée en plusieurs couches distinctes. Elle est développée en **Spring Boot** pour le backend et **React.js** pour le frontend, avec une base de données **H2**.

2.1 Architecture Technique

L'application est construite avec les technologies suivantes :

- **Backend** : implémenté en **Spring Boot**, exposant des API REST sécurisées.
- **Frontend** : développé avec **React.js** et utilisant **Axios** pour les requêtes API.
- **Base de données** : gérée avec **H2**.
- **Authentification** : basée sur **Spring Security** avec gestion des sessions HTTPS.
- **Stockage des images** : fichiers stockés sur le serveur et accessibles via une API.

2.2 Architecture Logicielle

L'application est organisée en couches logicielles, chacune ayant une responsabilité spécifique :

2.2.1 Couche Présentation (Frontend)

- Interface utilisateur développée avec **React.js**.
- Communication avec le backend via des **requêtes Axios**.
- Gestion des états avec **useState** et **useEffect**.

2.2.2 Couche Métier (Backend)

- Gestion des utilisateurs, des commandes, des paiements et du catalogue de véhicules.
- Application des **design patterns** (Factory, Observer, State, Strategy, etc.).
- Sécurisation avec **Spring Security** et gestion des sessions.

2.2.3 Couche Persistance (Base de Données)

- Stockage des données avec **H2**.
- Gestion des entités avec JPA/Hibernate.

2.2.4 Communication et Sécurité

- API REST avec **Spring Web**.
- Sécurisation des routes avec **Spring Security**.
- Gestion des sessions HTTPS pour l'authentification.

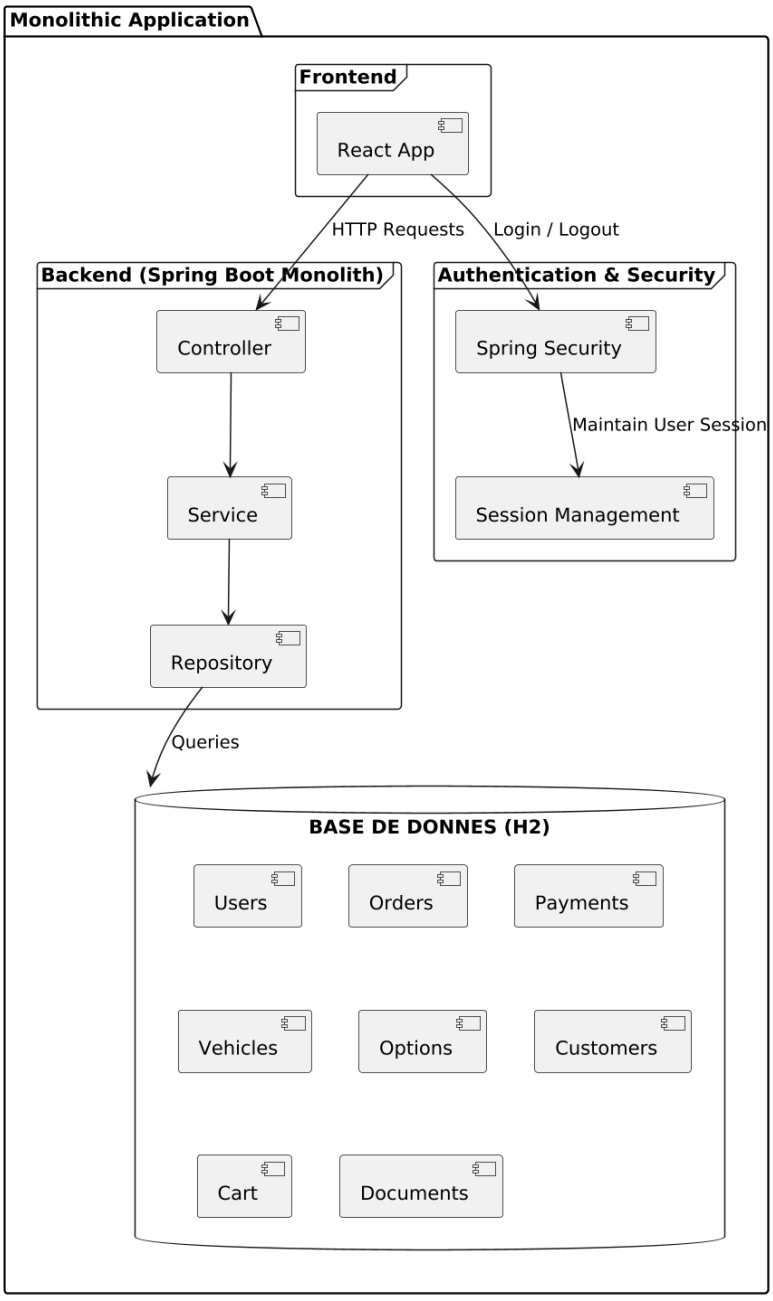
2.3 Gestion des Interactions

Toutes les fonctionnalités sont directement intégrées au sein du même code base, ce qui permet de garantir une exécution optimisée des traitements :

- Lorsqu'un utilisateur passe une commande, l'application enregistre l'opération et génère automatiquement les documents nécessaires.
- Les paiements sont traités directement via une intégration avec des passerelles.
- Les notifications et mises à jour des commandes sont gérées en interne via des mécanismes de gestion d'événements.

L'ensemble du système fonctionne de manière unifiée, garantissant ainsi une meilleure fiabilité et une exécution fluide des différentes opérations.

L'architecture globale peut être illustrée par le diagramme de composants ci-dessous :



Ce diagramme de composants illustre l'organisation structurelle du système de vente en ligne de véhicules basé sur une architecture monolithique en Spring Boot et React.js.

3 Diagrammes UML

La conception du système a été réalisée en utilisant plusieurs **diagrammes UML** afin de modéliser les interactions, les entités, les processus métier, ainsi que la structure logicielle du système. Ces diagrammes permettent de mieux comprendre le fonctionnement global du système, les relations entre les différentes entités et la communication entre les composants.

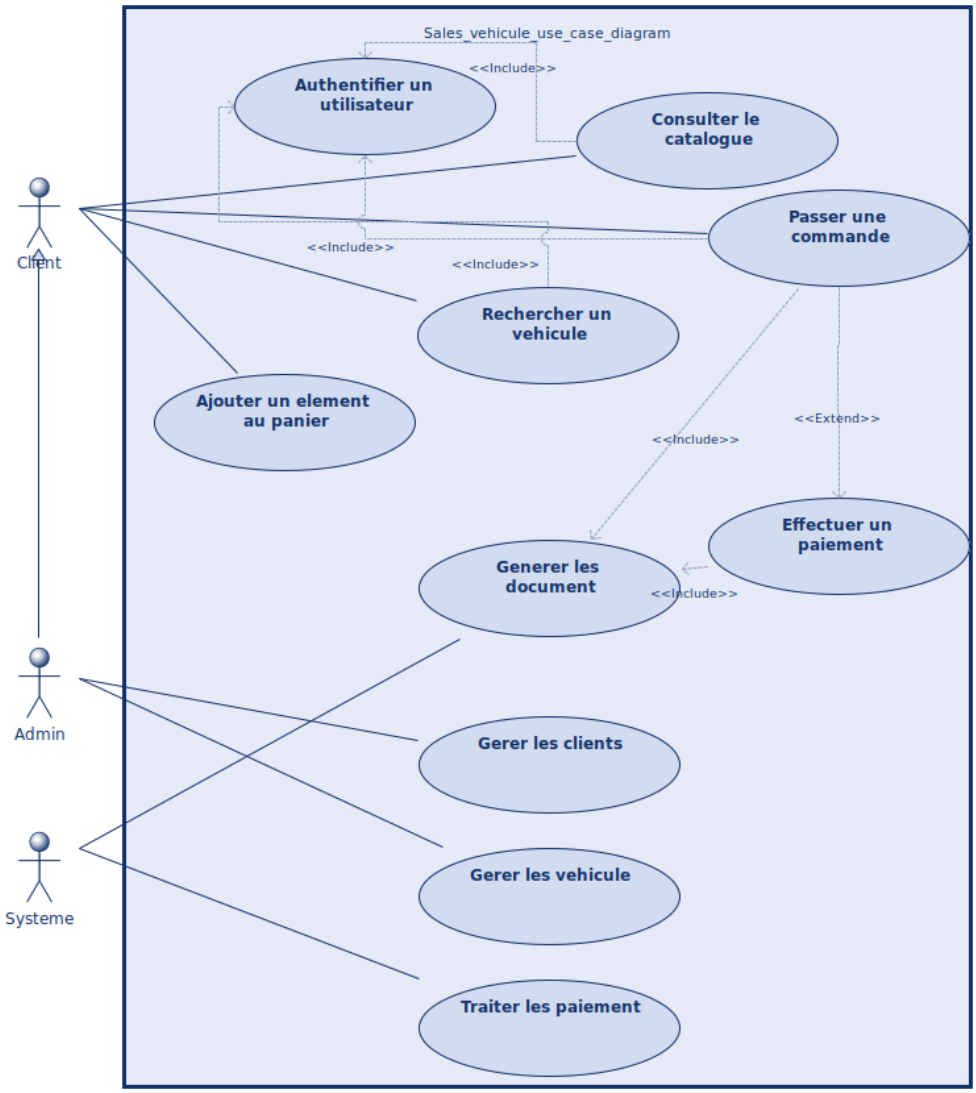
Les diagrammes réalisés sont :

— Diagramme de cas d'utilisation
— Diagramme de classe
— Diagramme de séquence
— Diagramme d'activité
— Diagramme de composants

Chaque diagramme est accompagné d'une explication détaillée afin de clarifier son rôle dans le système.

3.1 Diagramme de cas d'utilisation

Le **diagramme de cas d'utilisation** illustre les interactions entre les différents acteurs (utilisateurs) et les fonctionnalités principales du système.



Description : Ce diagramme représente les interactions entre les acteurs et les cas d'utilisation du système de vente de véhicules en ligne. Il distingue trois acteurs principaux :

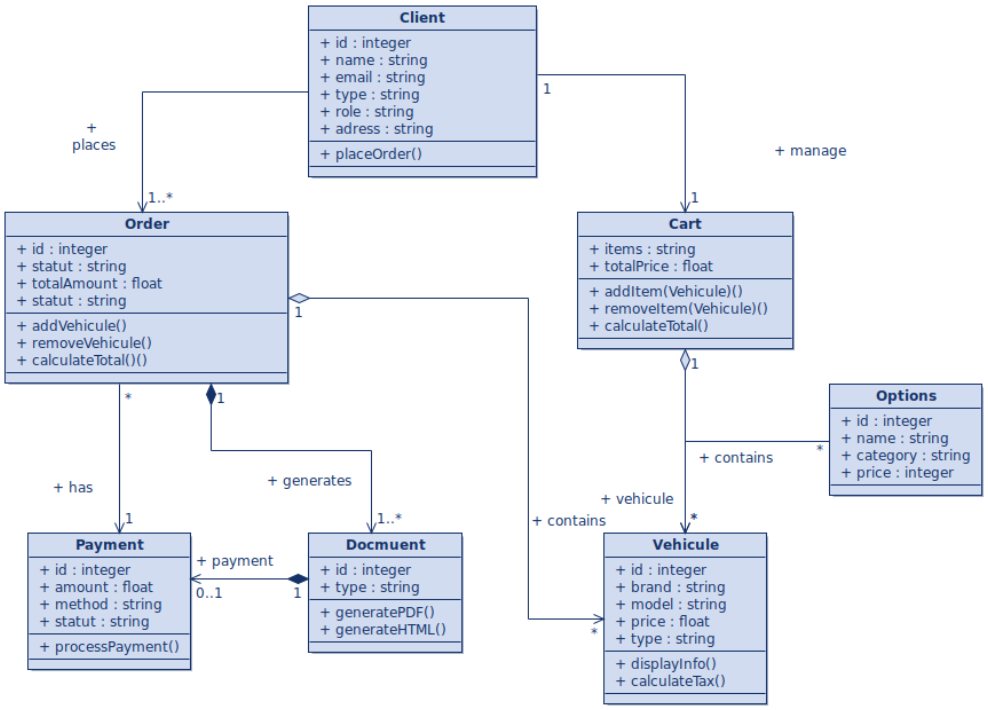
— **Client** : Peut consulter le catalogue et rechercher un véhicule après s'être authentifié, ajouter des véhicules au panier, passer une commande, et effectuer un paiement.

— **Admin** : Peut gérer les clients, les véhicules,les commandes et traiter les paiements.
— **Système (Automatisé)** : Génère les documents nécessaires à la confirmation de la commande et de paiement.

Ce diagramme illustre ainsi le fonctionnement global du système en mettant en avant les principales fonctionnalités et les rôles des utilisateurs.

3.2 Diagramme de classe

Le **diagramme de classe** présente les principales entités du système ainsi que leurs relations. Il permet de visualiser la structure statique du système en identifiant les classes, leurs attributs, leurs méthodes, et les associations entre elles.



Description : Le diagramme de classe comprend les entités principales suivantes :

— **Client** : Représente un utilisateur du système, qui peut passer des commandes et gérer un panier.
— **Order** : Représente une commande passée par un client. Une commande contient plusieurs véhicules.
— **Vehicle** : Représente un véhicule proposé à la vente (voiture ou scooter).

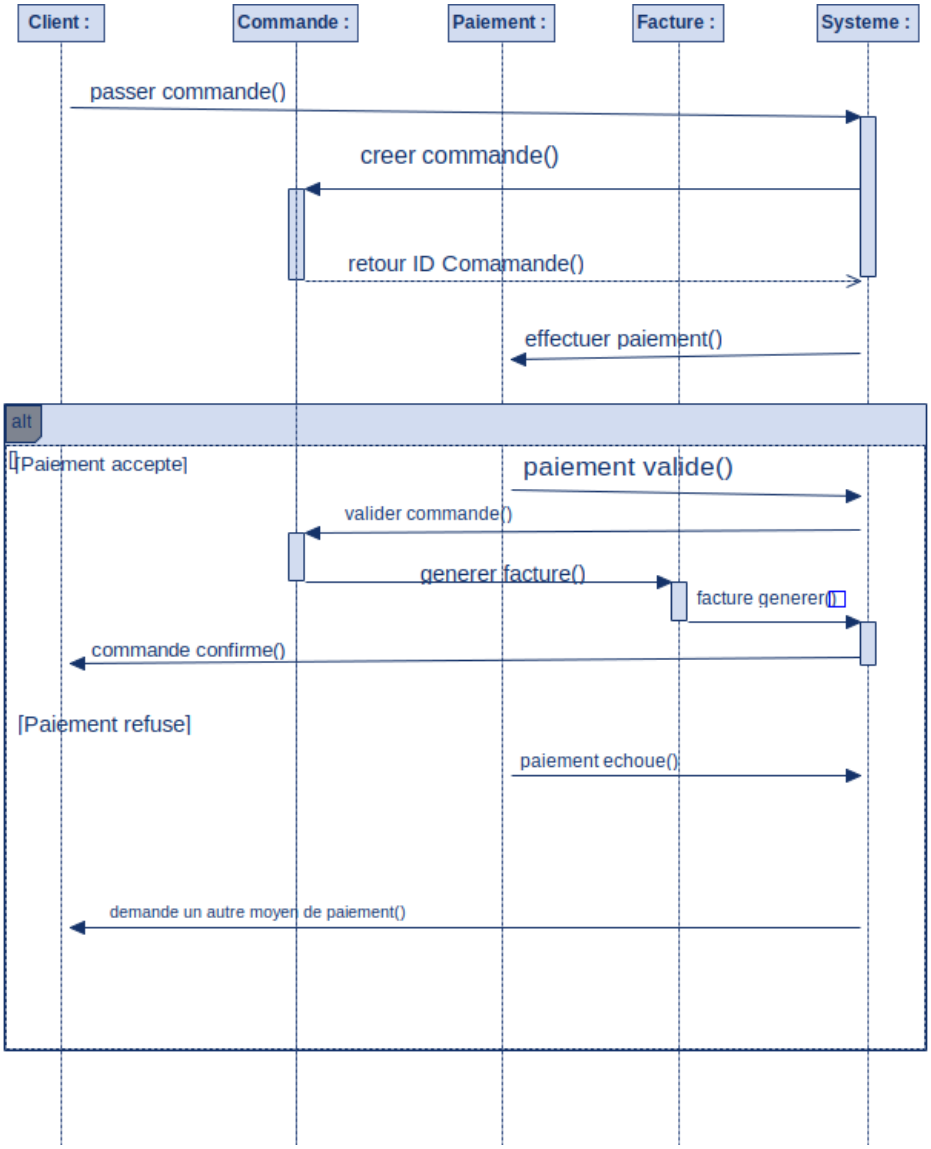
— **Cart** : Représente le panier d'achat d'un client.
— **Payment** : Représente le paiement d'une commande, pouvant être effectué au comptant ou à crédit.
— **Document** : Représente les documents générés lors de la commande, tels que le bon de commande et le certificat d'immatriculation.

Relations principales :
— Un **Client** peut passer plusieurs **Order**.
— Un **Order** contient plusieurs **Vehicle**.
— Un **Cart** est géré par un **Client** et contient plusieurs **Vehicle**.
— Chaque **Order** est associé à un **Payment**.
— Un **Order** génère plusieurs **Document**.

Ce diagramme permet de comprendre la structure des données du système et leurs relations.

3.3 Diagramme de séquence

Le **diagramme de séquence** montre le déroulement des interactions entre les différents objets du système lors du processus de commande. Il permet de visualiser la chronologie des échanges de messages et les appels de méthodes.



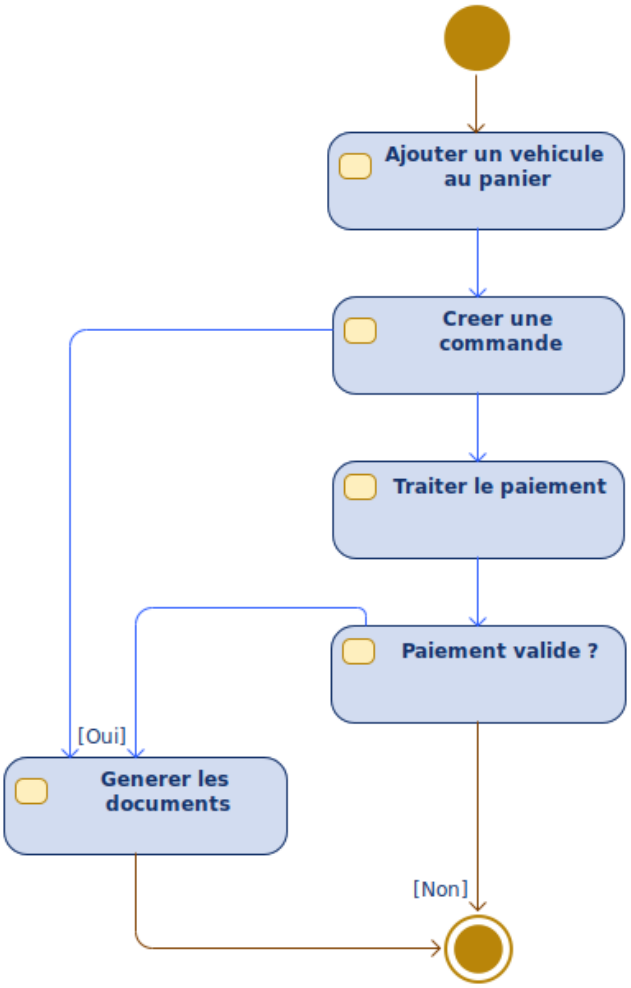
Description : Le processus de commande suit les étapes suivantes :

1. Le **Client** ajoute un véhicule au panier (*addVehicle()*).
2. Le **Cart** crée une commande à partir des véhicules ajoutés (*createOrder()*).
3. Le **Module Order** traite la commande et demande le paiement via le **Module Payment** (*processPayment()*).
4. Une fois le paiement validé, le **Module Order** demande au **Module Document** de générer les documents nécessaires (*generateDocuments()*).
5. Le **Module Document** renvoie les documents au **Module Order**.
6. Le **Module Order** confirme la commande au **Client** (*orderConfirmed()*).

Ce diagramme met en évidence les interactions clés entre les objets pour effectuer une commande, en assurant la gestion des paiements et la génération des documents nécessaires.

3.4 Diagramme d'activité

Le **diagramme d'activité** décrit les étapes du processus de commande, depuis l'ajout d'un véhicule au panier jusqu'à la validation du paiement et la génération des documents.



Description : Le diagramme d'activité met en évidence les étapes principales du flux de travail :

1. Le processus commence par l'ajout d'un véhicule au panier.

2. Une commande est créée à partir des véhicules présents dans le panier.
3. Le paiement est ensuite traité.
4. Une décision est prise : le paiement est-il validé ?
   — **Oui** : Les documents nécessaires (bon de commande, certificat) sont générés.
   — **Non** : Le processus se termine sans succès.
5. Le processus se termine par la confirmation de la commande au client.

Ce diagramme permet de visualiser les différents chemins possibles dans le processus de commande, en incluant les décisions critiques (comme la validation du paiement).

4 Patrons de conception utilisés

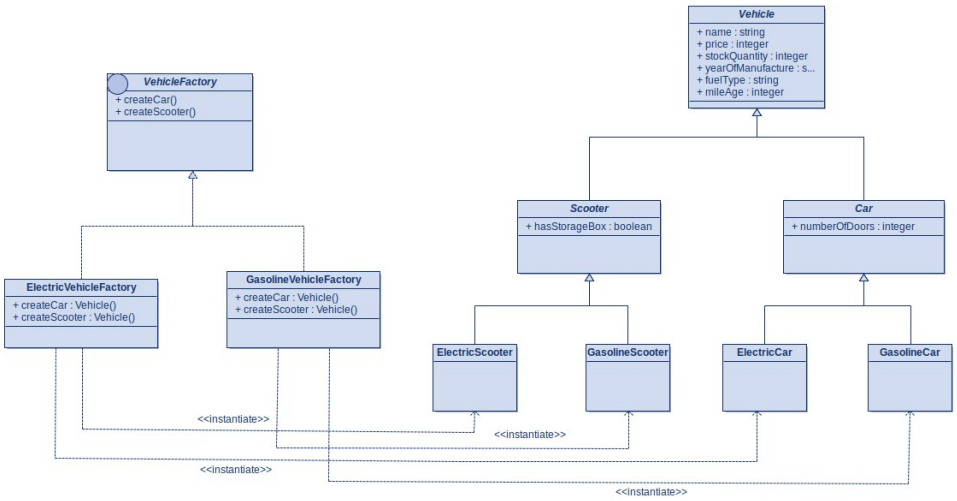
Dans le cadre de ce projet, plusieurs **patrons de conception** ont été appliqués afin d'assurer la flexibilité, la réutilisabilité et la modularité du système. Chaque module utilise un ou plusieurs patrons de conception adaptés à ses fonctionnalités spécifiques.

Les patrons de conception utilisés sont décrits ci-dessous, avec des diagrammes explicatifs pour chaque cas.

4.1 Gestion du catalogue de véhicules

4.1.1 Abstract Factory

Le **patron Abstract Factory** est utilisé dans le **module de gestion du catalogue** pour gérer la création de différents types de véhicules (par exemple, des voitures et des scooters) sans spécifier leurs classes concrètes.



Description :

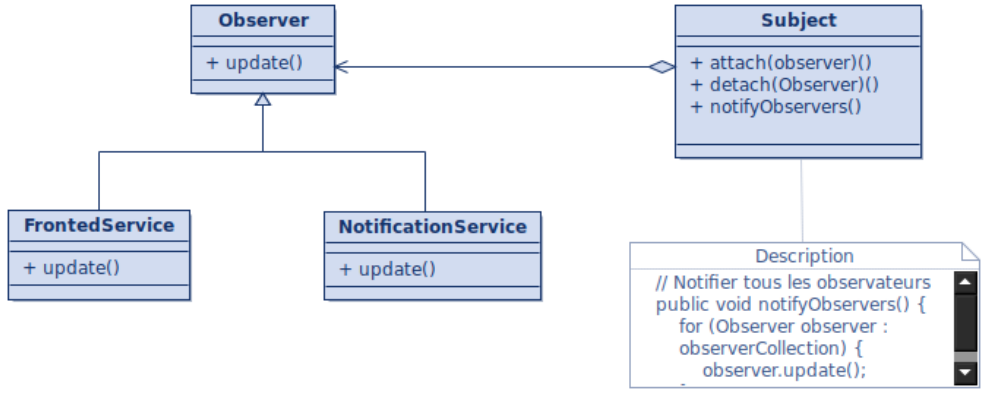
1. VehicleFactory : Déclare une interface pour créer des objets de type Vehicule (voitures et scooters) sans spécifier leur classe concrète.
2. Vehicle : Classe abstraite des différents types des véhicules.
3. ElectricVehicleFactory : Implémente l'interface VehicleFactory pour créer des véhicules électriques (voiture électrique et scooter électrique).
4. GasolineVehicleFactory : Implémente l'interface VehicleFactory pour créer des véhicules à essence (voiture à essence et scooter à essence).
5. Car : Définit la classe abstraite pour les objets de type voiture independemment de de leur famille (electrique , essence)
6. Scooter : Définit la classe abstraite pour les objets de type scooter independemment de leur famille (electrique , essence)
7. ElectricCar : Étend la class Car pour représenter une voiture électrique.
8. GasolineCar : Étend la classe Car pour représenter une voiture à essence.
9. ElectricScooter : Étend la classe Scooter pour représenter un scooter électrique.
10. GasolineScooter : Étend la classe Scooter pour représenter un scooter à essence.

Ce patron permet d'ajouter facilement de nouveaux types de véhicules sans modifier le code existant. Par exemple, on pourrait ajouter une nouvelle factory pour créer des camions ou des motos.

4.1.2 Observer Pattern

Le **patron Observer** est utilisé dans le **Module Vehicle** pour permettre à plusieurs module de recevoir des notifications automatiques lorsqu'un changement

est effectué sur un sujet observé (Subject). Ce patron permet de maintenir les différentes parties du système synchronisées sans dépendance forte entre elles.



Description : Le diagramme montre les principales classes impliquées dans le patron Observer :

- **Subject** : Classe représentant le sujet observé. Elle contient la collection des observateurs inscrits et fournit les méthodes pour les attacher ou les détacher. La méthode principale est `notifyObservers()`, qui notifie tous les observateurs inscrits en appelant leur méthode `update()`.
- **Observer** : Interface définissant la méthode `update()`, que chaque observateur doit implémenter.
- **Fronted** : Classe concrète qui implémente l'interface `Observer`. Cette classe est mise à jour lorsqu'une notification est reçue.
- **Module Notification** : Classe concrète qui implémente également l'interface `Observer`. Cette classe est responsable de l'envoi des notifications aux utilisateurs.

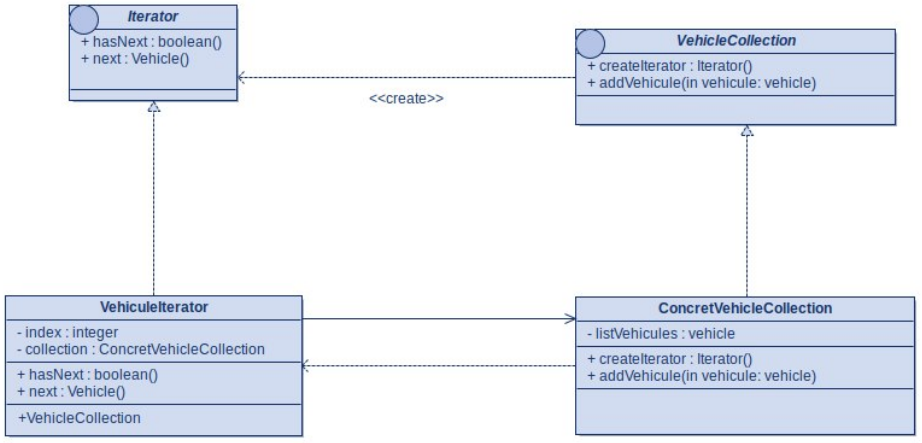
Fonctionnement :

1. Les classes **Frontend** et **Notification** s'inscrivent auprès du **Subject** en tant qu'observateurs.
2. Lorsqu'un changement est effectué dans le **Subject** (par exemple, ajout ou suppression d'un véhicule), la méthode `notifyObservers()` est appelée.
3. Chaque observateur inscrit reçoit une notification et exécute sa méthode `update()` pour se mettre à jour.

Le patron Observer permet d'assurer une communication efficace entre les différentes parties du système, en garantissant que les modules recevant les notifications sont toujours à jour.

4.1.3 Iterator Pattern

Le **patron Iterator** est utilisé dans le **module Vehicle** pour parcourir le catalogue de véhicules sans exposer sa structure interne. Ce patron permet d'accéder aux éléments d'une collection (le catalogue de véhicules) de manière séquentielle, sans que le client ait à connaître les détails de l'implémentation du catalogue.



Description : Le diagramme montre les principales classes impliquées dans le patron Iterator :

1. Iterator : Définit l' interface qui parcoure les éléments d'une collection de véhicules (parvoitures et scooters) .
2. VehicleIterator : est la classe concrète qui implémente l'interface 'Iterator' et effectue l'itération sur la collection de véhicules .
3. VehiculeCollection : Définit l'interface pour la creation d' un objet Iterator. Il représente une collection de véhicules.
4. ConcreteVehiculeCollection : est la classe concrète qui implemente l'interface 'VehiculeCollection' , stocke les véhicules et crée des instances de l'itérateur.

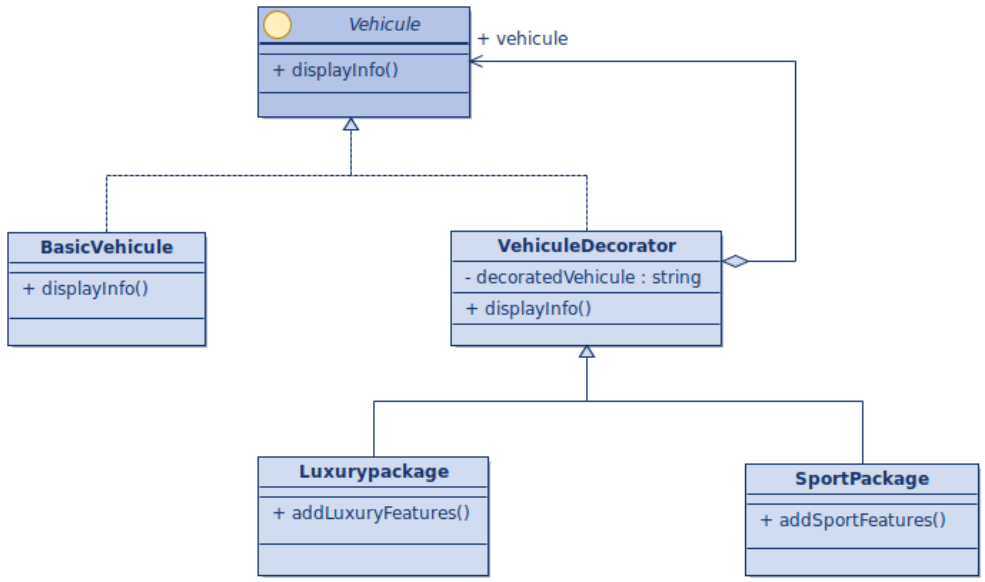
Fonctionnement :

1. Le client demande un itérateur à la classe **VehicleCatalog** en appelant la méthode `createIterator()`.
2. La classe **VehicleCatalog** retourne une instance de **VehicleIterator**.
3. Le client utilise les méthodes `hasNext()` et `next()` de **VehicleIterator** pour parcourir les véhicules disponibles dans le catalogue.

Ce patron permet d'assurer un accès séquentiel aux éléments d'une collection tout en masquant les détails de la structure interne de cette collection.

4.1.4 Decorator Pattern

Le **patron Decorator** est utilisé dans le **Module Vehicle** pour ajouter dynamiquement des fonctionnalités supplémentaires à un véhicule sans modifier sa classe de base. Ce patron permet de personnaliser les véhicules en y ajoutant différents packages (comme un package de luxe ou un package sportif) tout en respectant le principe de responsabilité unique.



Description : Le diagramme montre les principales classes impliquées dans le patron Decorator :

- **Vehicle** : Interface de base ou classe abstraite représentant un véhicule. Elle contient la méthode `displayInfo()`, qui est implémentée par toutes les classes concrètes.
- **BasicVehicle** : Classe concrète représentant un véhicule de base sans aucune personnalisation.
- **VehicleDecorator** : Classe abstraite qui implémente l'interface **Vehicle** et contient un attribut `decoratedVehicle`, permettant de référencer un véhicule existant.
- **LuxuryPackage** : Classe concrète qui hérite de **VehicleDecorator** et ajoute des fonctionnalités de luxe au véhicule (comme le cuir, le toit ouvrant, etc.).
- **SportPackage** : Classe concrète qui hérite de **VehicleDecorator** et ajoute des fonctionnalités sportives au véhicule (comme un moteur turbo, des jantes en alliage, etc.).

Fonctionnement :

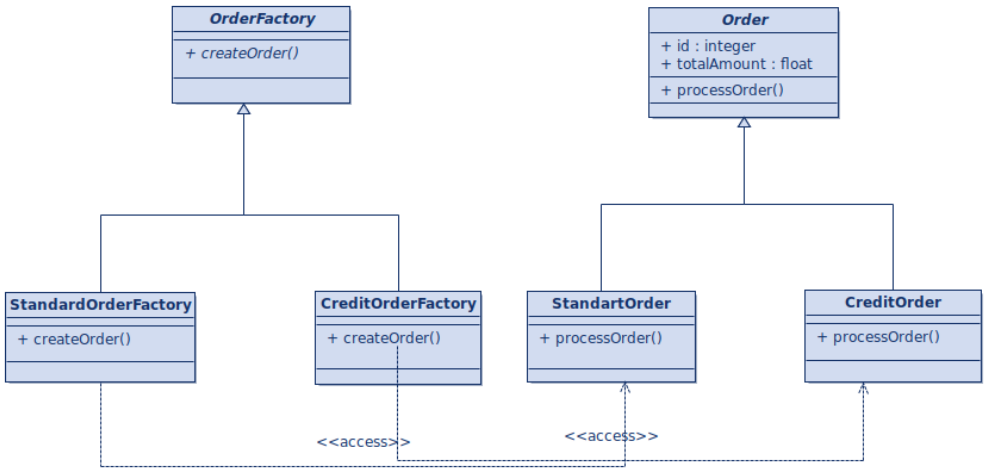
1. Le client commence par créer un **BasicVehicle**.
2. Ensuite, le client peut ajouter dynamiquement des packages supplémentaires au véhicule en utilisant des décorateurs tels que **LuxuryPackage** ou **SportPackage**.
3. Chaque package ajouté conserve la structure de l'objet de base tout en enrichissant ses fonctionnalités.
4. La méthode `displayInfo()` est appelée pour afficher les informations du véhicule avec les packages ajoutés.

Ce patron permet de personnaliser les véhicules en fonction des besoins des clients sans modifier la classe de base **Vehicle**, assurant ainsi une grande flexibilité et réutilisabilité du code.

4.2 Gestion des commandes

4.2.1 Factory Method Pattern

Le **patron Factory Method** est utilisé dans le **module Order** pour créer différents types de commandes sans exposer directement leur classe concrète. Ce patron permet de simplifier la gestion de la création des objets et d'assurer que le bon type de commande est créé en fonction du contexte (commande standard ou commande avec paiement à crédit).



Description : Le diagramme montre les principales classes impliquées dans le patron Factory Method :

— **OrderFactory** : Classe abstraite définissant la méthode `createOrder()`, qui est implémentée par les factories concrètes.

— **StandardOrderFactory** et **CreditOrderFactory** : Classes concrètes qui héritent de **OrderFactory** et implémentent la méthode `createOrder()` pour créer respectivement des commandes standards et des commandes à crédit.
— **Order** : Classe produit abstraite représentant une commande. Elle contient des attributs tels que `id` et `totalAmount`, ainsi qu'une méthode `processOrder()` pour traiter la commande.
— **StandardOrder** et **CreditOrder** : Classes concrètes qui héritent de **Order** et implémentent la méthode `processOrder()` en fonction du type de commande.

Fonctionnement :

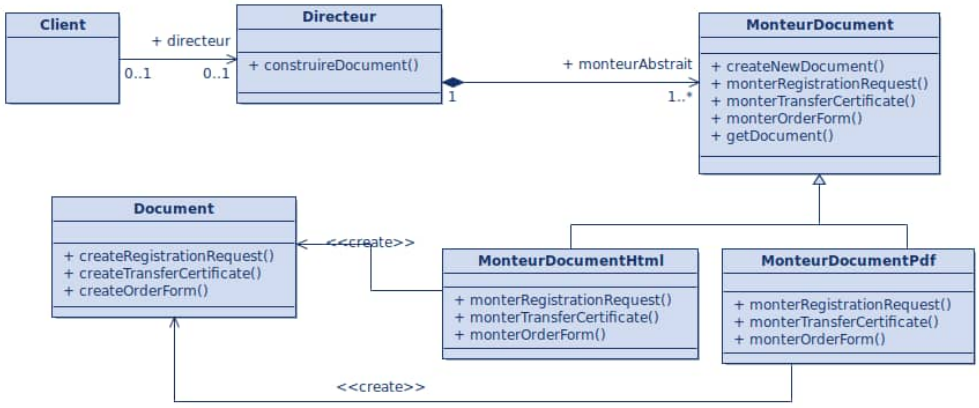
1. Le client utilise une factory concrète (comme **StandardOrderFactory** ou **CreditOrderFactory**) pour créer une commande.
2. La méthode `createOrder()` de la factory concrète retourne une instance de la classe appropriée (**StandardOrder** ou **CreditOrder**).
3. Une fois la commande créée, la méthode `processOrder()` est appelée pour traiter la commande.

Ce patron permet de simplifier la création de différents types de commandes et d'assurer que les commandes sont traitées correctement en fonction de leur type.

4.3 Gestion des documents

4.3.1 Builder Pattern

Le **patron Builder** est utilisé dans le **Module Document** pour faciliter la création de documents complexes tels que les bons de commande, les certificats de transfert et les formulaires d'enregistrement. Ce patron permet de séparer la construction d'un document de sa représentation finale, permettant ainsi de produire des documents dans différents formats (HTML, PDF, etc.).



Description : Le diagramme montre les principales classes impliquées dans le patron Builder :

- **Directeur** : Classe qui dirige le processus de construction du document en utilisant un constructeur abstrait (**MonteurDocument**).
- **MonteurDocument** : Interface définissant les méthodes nécessaires pour construire un document, telles que `createNewDocument()`, `monterRegistrationRequest()`, `monterTransferCertificate()`, et `monterOrderForm()`.
- **MonteurDocumentHtml** : Classe concrète qui implémente l'interface **MonteurDocument** pour produire des documents au format HTML.
- **MonteurDocumentPdf** : Classe concrète qui implémente l'interface **MonteurDocument** pour produire des documents au format PDF.
- **Document** : Classe produit finale représentant le document généré. Elle peut être un bon de commande, un certificat de transfert ou un formulaire d'enregistrement.

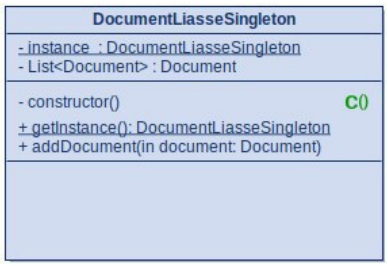
Fonctionnement :

1. Le **Client** crée une instance de **Directeur** et lui fournit un constructeur concret (**MonteurDocumentHtml** ou **MonteurDocumentPdf**).
2. Le **Directeur** dirige le processus de construction du document en appelant les méthodes définies dans l'interface **MonteurDocument**.
3. Une fois le processus terminé, le document final est récupéré via la méthode `getDocument()`.

Ce patron permet de produire des documents dans différents formats sans modifier la logique principale de construction. Cela garantit une grande flexibilité et une séparation claire des préoccupations.

4.3.2 Singleton Pattern

Le **patron Singleton** est utilisé dans le **module Document** pour garantir qu'une seule instance de la classe **DocumentLiasseSingleton** existe et qu'elle est accessible globalement. Ce patron est utile pour éviter la duplication d'instances et assurer une gestion cohérente des modèles de documents utilisés dans le système.



Description : Le diagramme montre la classe principale impliquée dans le patron Singleton :

— **DocumentLiasseSingleton** : Classe singleton qui contient les méthodes pour charger différents types de modèles de documents, tels que les demandes d'enregistrement (`loadRegistrationRequest()`), les certificats de transfert (`loadTransferCertificate()`), et les bons de commande (`loadOrderForm()`).

La classe **DocumentLiasseSingleton** utilise une variable privée `instance` pour stocker la seule instance de la classe. La méthode `getInstance()` permet d'accéder à cette instance unique.

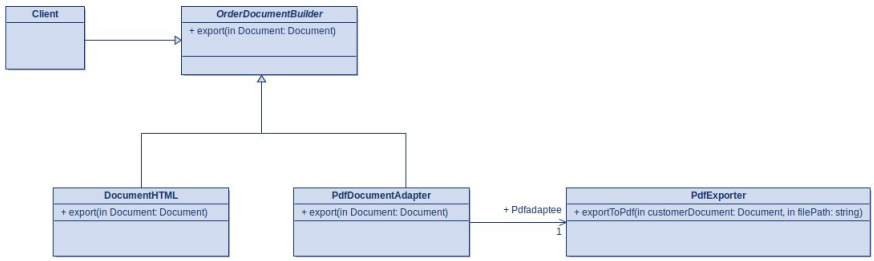
Fonctionnement :

1. Le constructeur de la classe **DocumentLiasseSingleton** est privé, empêchant ainsi la création d'instances multiples à partir de l'extérieur.
2. La méthode `getInstance()` vérifie si une instance de la classe existe déjà :
   — Si oui, elle retourne l'instance existante.
   — Si non, elle crée une nouvelle instance et la retourne.
3. Les méthodes `loadRegistrationRequest()`, `loadTransferCertificate()`, et `loadOrderForm()` sont utilisées pour charger les différents types de modèles de documents.

Ce patron permet de centraliser la gestion des modèles de documents tout en assurant qu'une seule instance de **DocumentLiasseSingleton** est utilisée dans tout le système, réduisant ainsi la consommation de ressources et assurant une cohérence globale.

4.3.3 Adapter Pattern

Le **patron Adapter** est utilisé dans le **OrderDocumentBuilder** pour convertir un document dans différents formats (HTML, PDF, etc.) sans modifier les classes existantes. Ce patron permet d'adapter les documents générés pour répondre aux besoins spécifiques du client ou aux contraintes d'un système externe.



Description : Le diagramme montre les principales classes impliquées dans le patron Adapter :

- **OrderDocumentBuilder** : Classe abstraite représentant le document de base. Elle définit les méthodes principales de création de documents, **export(document :document)**.
- **DocumentHtml** : Sous-classe de **Document** qui génère les documents au format HTML.
- **DocumentPdf** : Sous-classe de **Document** qui génère les documents au format PDF.
- **GeneratePDF** : Classe externe qui fournit une méthode spécifique `generatePDF()` pour générer des documents PDF.

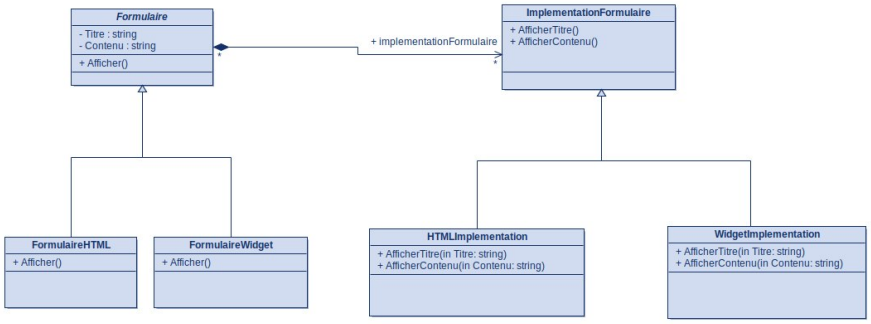
Fonctionnement :

1. Le **Client** utilise un objet de type **Document** (par exemple, **DocumentHtml** ou **DocumentPdf**) pour créer un document.
2. La classe **GeneratePDF** contient une méthode `generatePDF()` pour produire un fichier PDF.
3. Le rôle de l'adaptateur (`pdfAdapte`) est de faire le lien entre la classe **DocumentPdf** et la classe externe **GeneratePDF**, permettant ainsi de convertir un document en PDF sans modifier la classe **Document**.

Ce patron permet d'assurer la compatibilité entre des interfaces qui, autrement, seraient incompatibles. Il facilite également la conversion de documents dans différents formats selon les besoins du système ou du client.

4.3.4 Bridge Pattern

Le **patron Bridge** est utilisé dans le **module Auth** pour séparer l'abstraction de son implémentation, permettant ainsi une grande flexibilité dans la manière d'afficher les formulaires d'authentification. Ce patron permet de gérer différentes implémentations d'affichage (HTML, Widget, etc.) sans modifier la structure principale des formulaires.



Description : Le diagramme montre les principales classes impliquées dans le patron Bridge :

- **Formulaire** : Classe abstraite représentant un formulaire d'authentification. Elle contient les attributs `Titre` et `Contenu`, ainsi que la méthode `Afficher()` pour afficher le formulaire.
- **FormulaireHTML** et **FormulaireWidget** : Classes concrètes qui héritent de **Formulaire**. Elles utilisent une implémentation spécifique pour afficher les formulaires.
- **ImplementationFormulaire** : Interface définissant les méthodes nécessaires à l'affichage des titres et des contenus des formulaires, telles que `AfficherTitre()` et `AfficherContenu()`.
- **HTMLImplementation** et **WidgetImplementation** : Classes concrètes qui implémentent l'interface **ImplementationFormulaire**. Elles fournissent des méthodes spécifiques pour afficher les formulaires au format HTML ou sous forme de widget.

Fonctionnement :

1. La classe abstraite **Formulaire** utilise un objet de type **ImplementationFormulaire** pour afficher le formulaire.
2. Les classes concrètes **FormulaireHTML** et **FormulaireWidget** peuvent utiliser différentes implémentations pour afficher les formulaires.
3. Les classes concrètes **HTMLImplementation** et **WidgetImplementation** fournissent les détails spécifiques pour l'affichage des formulaires respectivement au format HTML ou sous forme de widget.

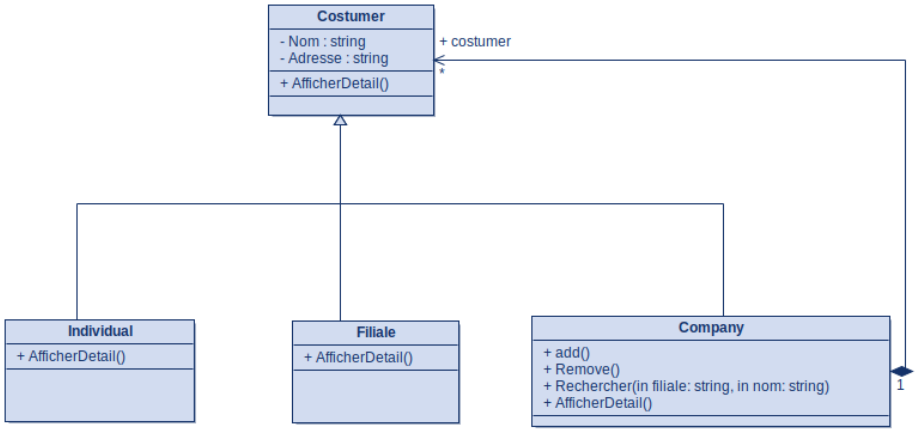
Avantages du Bridge Pattern :

— Sépare l'abstraction (formulaire) de son implémentation (affichage), facilitant ainsi la maintenance et l'évolution du code.
— Permet d'ajouter facilement de nouvelles implémentations d'affichage sans modifier la structure des formulaires.
— Réduit le couplage entre l'interface utilisateur et les implémentations spécifiques.

4.4 Gestion des clients

4.4.1 Composite Pattern

Le **patron Composite** est utilisé pour structurer des objets en arbre de manière à ce que les clients puissent traiter les objets individuels et les groupes d'objets de manière uniforme. Il favorise la **récursivité** et l'**abstraction**, permettant ainsi aux clients d'interagir avec des objets simples ou composites sans se soucier des différences.



Description : Le diagramme montre les principales classes impliquées dans le patron Composite :

— **Component (Costumer)**
  — Définit l'interface commune à tous les objets (simples ou composites).
  — Contient des attributs communs (`Nom`, `Adresse`) et une méthode `AfficherDetail()`.
— **Leaf (Individual, Filiale)**
  — Représente des éléments unitaires (individus ou filiales).
  — Implémente `AfficherDetail()`, mais ne contient pas de sous-éléments.
— **Composite (Company)**
  — Contient d'autres composants (objets de type `Filiale`).

— Gère les sous-composants avec des méthodes comme `add()`, `remove()`, et `Rechercher()`.
— Implémente également `AfficherDetail()`, qui peut appeler récursivement `AfficherDetail()` sur ses sous-composants.

Fonctionnement :

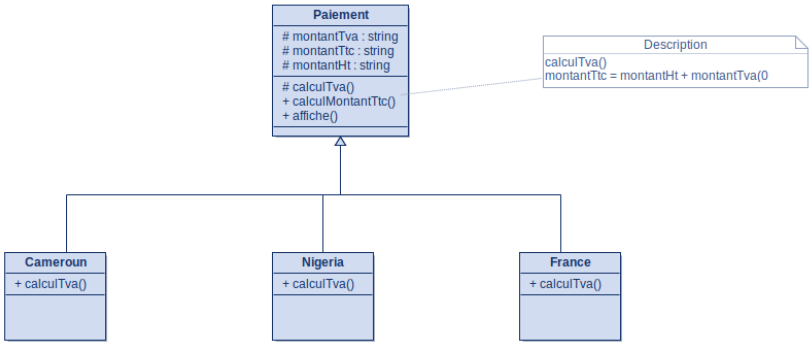
1. Une instance de **Company** peut contenir plusieurs instances de **Filiale** ou d'autres **Company**.
2. La méthode `AfficherDetail()` peut être appelée sur n'importe quelle entité de la hiérarchie pour afficher ses informations.
3. La méthode `add()` permet d'ajouter dynamiquement des filiales à une entreprise.
4. La méthode `remove()` supprime une filiale d'une entreprise.
5. La méthode `Rechercher(filiale, nom)` permet de rechercher une filiale ou un client spécifique dans la hiérarchie.

Avantages du Composite Pattern :
— Permet de gérer des structures hiérarchiques complexes de manière simple et uniforme.
— Facilite l'ajout de nouveaux types de sociétés sans modifier le code existant.
— Réduit le couplage entre les différentes classes du système.

4.5 Gestion des paiements

4.5.1 Template Method Pattern

Le **patron Template Method** est utilisé dans le **module de paiement** pour définir le processus général de traitement d'un paiement, tout en permettant aux sous-classes de personnaliser certaines étapes spécifiques. Ce modèle de conception structure les étapes communes à tous les paiements, tout en offrant la flexibilité nécessaire pour que chaque sous-classe adapte les détails en fonction du mode de paiement utilisé.



Description :

Le diagramme montre les principales classes impliquées dans le **Template Method Pattern** appliqué à la gestion des paiements :

- **Paiement** (*Classe abstraite*) : Définit le processus général de calcul du montant total. - Contient la méthode `calculMontantTtc()`, qui applique la formule `montantTtc = montantHt + montantTva`. - Contient une méthode abstraite `calculTva()`, qui est laissée aux sous-classes pour définir le taux de TVA spécifique à chaque pays.
- **Cameroun** (*Classe concrète qui hérite de* `Paiement|`) :
  - Implémente la méthode `calculTva()` en appliquant une TVA de 19%.
- **Nigeria** (*Classe concrète qui hérite de* `Paiement|`) :
  - Implémente la méthode `calculTva()` en appliquant une TVA de 7.5%.
- **France** (*Classe concrète qui hérite de* `Paiement|`) :
  - Implémente la méthode `calculTva()` en appliquant une TVA de 20%.

Fonctionnement :

1. La classe abstraite `Paiement` définit la méthode `calculMontantTtc()`, qui applique la formule de calcul du montant total.
2. Chaque sous-classe (`Cameroun`, `Nigeria`, `France`) implémente sa propre version de la méthode `calculTva()` en fonction des taux spécifiques à chaque pays.
3. Lorsqu'un objet de type `Paiement` est utilisé pour un pays donné, la méthode `calculMontantTtc()` applique le montant de la TVA calculé par la sous-classe correspondante.

Ce patron permet de s'assurer que tous les paiements suivent un processus cohérent tout en permettant des variations dans certaines étapes spécifiques.

5 Conclusion

Ce projet a permis d'explorer **les concepts UML et les Design Patterns** dans un contexte réel d'application de vente de véhicules. Nous avons conçu une architecture **monolithique** avec une gestion claire des responsabilités et une implémentation efficace des patrons de conception.

Grâce à UML, nous avons modélisé les différents aspects du système (cas d'utilisation, diagrammes de classes, séquences, etc.), facilitant ainsi la compréhension et l'implémentation. Les Design Patterns ont renforcé la flexibilité et la maintenabilité du code, notamment avec Factory, Decorator, Observer, et Strategy.

En somme, cette étude de cas nous a permis de structurer une application robuste en suivant les meilleures pratiques du génie logiciel.