

## **Laboratoire de Programmation en C++**

**2<sup>ème</sup> informatique et systèmes :  
option(s) industrielle et réseaux (1<sup>er</sup> quadrimestre)  
et 2<sup>ème</sup> informatique de gestion (1<sup>er</sup> quadrimestre)**

**Année académique 2022-2023**

### ***Gestion d'une concession automobile***

**Anne Léonard  
Patrick Quettier  
Jean-Marc Wagner**

## Introduction

### 1. Informations générales : UE, AA et règles d'évaluation

Cet énoncé de laboratoire concerne les unités d'enseignement (UE) suivantes :

**a) 2<sup>ème</sup> Bach. en informatique de Gestion : « Développement Système et orienté objet »**

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Base de la programmation orientée objet – C++** (45h, Pond. 30/90)
- **Principes fondamentaux des Systèmes d'exploitation** (15h, Pond. 10/90)
- **Système d'exploitation et programmation système UNIX** (75h, Pond. 50/90)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

**b) 2<sup>ème</sup> Bach. en informatique et systèmes : « Développement Système et orienté objet »**

Cette UE comporte les activités d'apprentissage (AA) suivantes :

- **Base de la programmation orientée objet – C++** (45h, Pond. 45/101)
- **Système d'exploitation et programmation système UNIX** (56h, Pond. 56/101)

Ce laboratoire intervient dans la construction de la côte de l'AA « Base de la programmation orientée objet – C++ ».

Quel que soit le bachelier, la cote de l'AA « Base de la programmation orientée objet – C++ » est construite de la même manière :

- ♦ théorie : un examen écrit en janvier 2023 et coté sur 20
- ♦ laboratoire (**cet énoncé**) : une évaluation globale en janvier, accompagnée de « check-points » réguliers pendant tout le quadrimestre. Cette évaluation fournit une note de laboratoire sur 20.
- ♦ note finale : **moyenne géométrique de la note de théorie (50%) et de la note de laboratoire (50%).**

Cette procédure est d'application tant en 1<sup>ère</sup> qu'en 2<sup>ème</sup> session.

1) Chaque étudiant doit être capable d'expliquer et de justifier l'intégralité du travail.

2) En 2<sup>ème</sup> session, un **report de note** est possible pour la théorie ou le laboratoire **pour des notes supérieures ou égales à 10/20**. Les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

3) Les consignes de présentation des travaux de laboratoire sont fournies par les différents professeurs de laboratoire.

## 2. Le contexte : Gestion d'une concession automobile

Les travaux de Programmation Orientée Objets (POO) se déroulent dans le contexte de la gestion d'une concession automobile. Plus particulièrement, il s'agit essentiellement de gérer la relation vendeur/client lors de la conception et de l'achat d'un véhicule automobile (choix du modèle, ajout d'options, ristournes accordées, contrat de vente).

Dans un premier temps, il s'agira de modéliser la notion de modèle et d'options de voiture. Nous aborderons ensuite et modéliseront les intervenants de l'application : les administratifs, les vendeurs et les clients.

L'application finale sera utilisée par 2 types d'utilisateurs (qui s'y connecteront à l'aide d'un couple login/mot de passe) :

- Les administratifs qui pourront ajouter de nouveaux utilisateurs à l'application et gérer en partie les contrats d'achats de voitures.
- Les vendeurs qui pourront concevoir, en compagnie d'un client, un projet de voiture, conclure avec lui un contrat sur base de ce projet.

L'application aura l'aspect visuel suivant :

Application Garage : Wagner Jean-Marc (Vendeur)

Connexion Employés Clients Voiture

Projet en cours :

Nom : 2008\_Vilvens

Modèle : 2008 GT Line e-2008

Puissance (Ch) : 136 Prix de base : 41620,00

☐ Essence ☐ Diesel ☒ Electrique ☐ Hybride

Modèles disponibles : 208 Active 1.2 PureTech 75 (18050,00) Choisir

Options disponibles : Peinture metallisee (400,00) Ajouter

Options choisies :

Code	Prix	Intitulé
VD09	150,00	Vitres laterales arrieres surteintees
UB01	250,00	Detecteur d'obstacles arriere

Supprimer Accorder réduction

Prix avec options : 42020,00

Nouveau Ouvrir Enregistrer

Employés du garage :

Numéro	Nom	Prénom	Fonction
1	ADMIN	ADMIN	Administratif
2	Charlet	Christophe	Vendeur
7	Wagner	Jean-Marc	Vendeur

Clients :

Numéro	Nom	Prénom	GSM
5	Leonard	Anne	0478.56.36.32
3	Quettier	Patrick	0495.36.36.36
6	Vilvens	Claude	0475.36.21.14

Contrats :

Numéro	Vendeur	Client	Voiture
1	Charlet ...	Leonard ...	3008_Quettier
2	Charlet ...	Leonard ...	208_Leonard
3	Wagner ...	Vilvens ...	2008_Vilvens

Nouveau Supprimer Visualiser Voiture

Il s'agit d'une interface graphique basée sur la librairie C++ Qt. Celle-ci vous sera fournie telle quelle. Vous devrez programmer la logique de l'application et non concevoir l'interface graphique.

### **3. Philosophie du laboratoire**

Le laboratoire de programmation C++ sous **Linux** a pour but de vous permettre de faire concrètement vos premiers pas en C++ au début du quadrimestre (septembre-octobre) puis de conforter vos acquis à la fin du quadrimestre (novembre-décembre). Les objectifs sont au nombre de trois :

- mettre en pratique les notions vues au cours de théorie afin de les assimiler complètement,
- créer des "briques de bases" pour les utiliser ensuite dans une application de synthèse,
- vous aider à préparer l'examen de théorie du mois de janvier.

Il s'agit bien d'un laboratoire de C++ sous Linux. Il a également pour but de vous familiariser à un environnement de développement autre que Windows mais surtout en dehors d'un IDE totalement fenêtré. Pour cela, on vous fournira une machine virtuelle (VMWare) dès le premier laboratoire. Pour ceux qui n'auraient pas de PC portable ou qui le souhaitent, une machine physique Linux, identique à la machine virtuelle fournie, est à disposition des étudiants (des comptes seront créées) : **Jupiter (10.59.28.2)** sur le réseau de l'école).

### **4. Méthodologie de développement**

La programmation orientée objet permet une approche modulaire de la programmation. En effet, il est possible de scinder la conception d'une application en 2 phases :

1. La programmation des classes de base de l'application (les briques élémentaires) qui rendent un service propre mais limité et souvent indépendant des autres classes. Ces modules doivent respecter les contraintes imposées par « un chef de projet » qui sait comment ces classes vont interagir entre elles. Cette partie est donc réalisée par « le programmeur créateur de classes ».
2. La programmation de l'application elle-même. Il s'agit d'utiliser les classes développées précédemment pour concevoir l'application finale. Cette partie est donc réalisée par « le programmeur utilisateur des classes ».

Durant la première partie de ce laboratoire (**de la mi-septembre à mi-novembre**), vous vous situez en tant que « programmeur créateur de classes ». On va donc vous fournir une série de jeux de tests (les fichiers **Test1.cpp**, **Test2.cpp**, **Test3.cpp**, ...) qui contiennent une fonction main() et qui vous imposeront le comportement (l'interface) de vos classes. Cette méthodologie de développement est bien connue en entreprise, il s'agit de la méthodologie appelée « **Test-Driven Development (TDD)** » :

Le **Test-Driven Development (TDD)**, ou développement piloté par les tests en français, est une méthode de développement de logiciel qui consiste à concevoir un logiciel par petites étapes, de façon progressive, en écrivant avant chaque partie du code source propre au logiciel les tests correspondants et en remaniant le code continuellement. (wikipedia)

Dans la deuxième partie du laboratoire (**de mi-novembre à fin décembre**), vous vous situerez en tant que « programmeur utilisateur des classes » utilisant les classes que vous aurez développées précédemment. C’est dans cette seconde phase que vous développerez l’application proprement dite.

## 5. Code source de base

Tous les fichiers sources de base fournis peuvent être obtenus par

- Clonage du repository **GitHub** : [https://github.com/hepl-dsoo/LaboCpp2022\\_Enonce](https://github.com/hepl-dsoo/LaboCpp2022_Enonce)
- Dans l’équipe **Teams** « Développement Système et Orienté Objet (2022-2023) » dont vous faites partie (si ce n’est pas le cas, demandez à votre professeur de laboratoire de vous y ajouter).

## 6. Planning et contenu des évaluations

### a) Evaluation 1 (continue) :

Le développement de l’application, depuis la création des briques de base jusqu’à la réalisation de l’application avec ses fonctionnalités a été découpé en une série d’étapes à réaliser dans l’ordre. A chaque nouvelle étape, vous devez rendre compte de l’état d’avancement de votre projet à votre professeur de laboratoire qui validera (ou pas) l’étape.

### b) Evaluation 2 (examen de janvier 2023) :

**Porte sur :**

- la validation des étapes non encore validées le jour de l’évaluation,
- le développement et les tests de l’application finale.
- Vous devez être capable d’expliquer l’entièreté de tout le code développé.

**Date d’évaluation** : jour de votre examen de Laboratoire de C++ (selon horaire d’examens)

**Modalités d’évaluation** : Sur la machine Linux fournie, selon les modalités fixées par le professeur de laboratoire.

## Plan des étapes à réaliser

Etape	Thème	Page
1	Une première classe	7
2	Associations entre classes → agrégations	8
3	Extension des classes existantes : surcharges des opérateurs	10
4	Associations de classes : héritage et virtualité	12
5	Les exceptions	14
6	Les containers et les templates	15
7	Première utilisation des flux	17
8	Un conteneur pour nos conteneurs : La classe Garage	19
9	Mise en place de l'interface graphique : Introduction à Qt	22
10	Importation de modèles et d'options : Fichiers textes	27
11	Ajout/Suppression d'employés et de clients / Gestion des rôles	28
12	Enregistrement sur disque : La classe Garage se sérialise elle-même	32
13	La gestion des contrats d'achat : la classe Contrat	33

**CONTRAINTES** : Tout au long du laboratoire de C++ (évaluation 1 et 2, et seconde session), il vous est interdit, pour des raisons pédagogiques, d'utiliser les **containers génériques template de la STL**.

## Etape 1 : Une première classe (Test1.cpp)

### a) Description des fonctionnalités de la classe

Un des éléments de base de l'application est la notion de modèle de voiture. Par exemple, Peugeot propose le modèle « 208 Active 1.5 BlueHDi 5P » qui présente un certain nombre de caractéristiques de base (moteur, puissance, nombre de chevaux, émission de CO2, ...). Afin de simplifier les choses, nous nous limiterons aux caractéristiques essentielles.



Notre première classe, la classe **Modele**, sera donc caractérisée par :

- Un **nom** : une chaîne de caractères allouée dynamiquement (**char \***) en fonction du texte qui lui est associé (Exemple : « 208 Active 1.5 BlueHDi 5P »).
- Une **puissance** : un entier (**int**) représentant la puissance en chevaux du moteur.
- Un type de **moteur** (**Moteur**) pouvant prendre les valeurs d'énumération Essence, Diesel, Electrique, Hybride. Cette énumération se déclare ainsi :

```
enum Moteur { Essence, Diesel, Electrique, Hybride };
```

- Le **prix de base** du modèle (**float**). Ce prix ne tient donc pas compte d'éventuelles options supplémentaires.

Comme vous l'impose le premier jeu de test (Test1.cpp), on souhaite disposer des trois formes classiques de constructeurs et d'un destructeur, des méthodes classiques getXXX() et setXXX() et une méthode pour afficher les caractéristiques de l'objet. La variable de type chaîne de caractères sera donc un **char\***. **Pour des raisons purement pédagogiques, le type string (de la STL) NE pourra PAS être utilisé dans la classe Modele.** Vous aurez l'occasion d'utiliser la classe string dans la suite du projet.

### b) Méthodologie de développement

Veillez à tracer vos constructeurs et destructeurs pour que vous puissiez vous rendre compte de quelle méthode est appelée et quand elle est appelée.

On vous demande de créer pour la classe Modele (ainsi que pour chaque classe qui suivra) les fichiers **.cpp** et **.h** et donc de travailler en fichiers séparés. Un **makefile** permettra d'automatiser la compilation de votre classe et de l'application de tests.

## Etape 2 (Test2x.cpp) : Associations entre classes → agrégations

Les bibliothèques standards du C++ offrent un moyen simple de gérer les chaînes de caractères. Il s'agit de la classe **string** que vous pouvez utiliser pour la suite de vos développements.

### a) La classe Voiture (Test2a.cpp)

Il s'agit à présent de créer une classe qui permettra au vendeur de réaliser un projet de voiture pour un client. Ce projet de voiture comporte un modèle de base (avec un prix de base) et un ensemble d'options que le vendeur pourra ajouter pour rencontrer les souhaits de l'acheteur (peinture métallisée, air conditionné, vitres teintées arrière, ...). Ce projet sera modélisé par la classe **Voiture** dont voici les variables membres (d'autres suivront pour les options mais une chose à la fois ☺ !) :

- Le **nom** du projet : il s'agit d'une chaîne de caractères (**string**) contenant un nom du genre « Projet\_208GTI\_MrDugenou ».
- Un **modèle** (variable du type **Modele**) : il s'agit du modèle de base du projet.

A nouveau (comme cela est imposé dans **Test2a.cpp**), vous devez programmer les trois types de constructeurs attendus, le destructeur et les méthodes getXXX/setXXX pour les deux variables membres. Bien sûr, cette classe doit posséder ses propres fichiers .cpp et .h.

La classe Voiture contenant une variable membre dont le type est une autre classe, on parle d'agrégation par valeur, l'objet Modele fait partie intégrante de l'objet Voiture.

### b) La classe Option (Test2b.cpp)

Il s'agit à présent de pouvoir ajouter au projet de voiture un certain nombre d'options souhaitées par l'acheteur. Nous allons donc modéliser une option par la classe **Option** qui présente les variables membres suivantes :

- Un **code** : il s'agit d'une chaîne de 4 caractères (**string**) qui identifie de manière unique une option dans le catalogue (Pour simplifier les choses, nous considérerons que n'importe quelle option peut être ajoutée à n'importe quel modèle de voiture). Exemple : « AB0B », « J2U6 », ...
- Un **intitulé** : il s'agit d'une chaîne de caractères (**string**) représentant l'intitulé de l'option. Exemple : « Peinture métallisée », « Air conditionné », « Jantes en alliage léger 15 pouces », ...
- Un **prix** : il s'agit d'un nombre (**float**) représentant le prix de l'option.

A nouveau, vous devez programmer les constructeurs, le destructeur et les getters/setters adéquats. Dans la suite de l'énoncé, nous ne le dirons plus mais il doit être clair que **toute classe doit disposer de constructeurs, d'un destructeur (!!!) et des getters/setters adéquats.**



c) **Agrégation entre les classes Voiture et Option (Test2c.cpp)**

Nous pouvons à présent compléter notre classe **Voiture** par la variable membre suivante :

- la variable **options** : il s'agit de l'ensemble des options que l'on va ajouter au projet. Nous limiterons le nombre d'options à 5. Cette variable sera du type « **tableau de pointeurs d'options** », c'est-à-dire **Option\* options[5]**. Chacune des cases de ce tableau sera initialisée à NULL mais pourra dans la suite pointer vers un objet de la classe Option.

Remarquons cependant que la variable options n'a pas besoin de getter/setter. Il s'agit d'un conteneur d'options dans lequel on ajoutera/retirera des options. On préférera donc ajouter à la classe **Voiture** les méthodes suivantes :

- **AjouteOption(const Option & option)** qui permet d'ajouter une option au projet. Pour cela, vous recherchez une case vide (NULL) dans le tableau options et vous lui assignez un pointeur vers un objet Option que vous aurez alloué dynamiquement.
- **RetireOption(string code)** qui permet de retirer, de la liste des options du projet, l'option dont le code est passé en paramètre à la méthode. Remarquez qu'une fois trouvée, vous devez libérer la mémoire et remettre la case du tableau à NULL...
- **getPrix()** qui permet de retourner le prix de la voiture dans l'état actuel. Il s'agit du prix de base du modèle augmenté du prix de chaque option. Remarquez que la classe Voiture ne contient pas de variable membre prix. La méthode getPrix() calcule le prix à partir du prix du modèle et du prix de chaque option présente dans le tableau options, avant de retourner le résultat de son calcul. De l'extérieur, le « programmeur utilisateur de la classe » ne le voit pas. Vive l'encapsulation ☺ !

Notez que la méthode **Affiche** de la classe **Voiture** doit être mise à jour pour tenir compte des options.

La classe **Voiture** possède un certain nombre de pointeurs vers des objets de la classe Option. N'oubliez pas de mettre à jour le **destructeur** de la classe Voiture afin de libérer correctement l'espace mémoire occupé par les options.

### **Etape 3 (Test3.cpp) :**

#### **Extension des classes existantes : surcharges des opérateurs**

Il s'agit ici, de surcharger un certain nombre d'opérateurs des classes développées ci-dessus afin de faciliter la conception d'un projet de voiture.

#### **a) Surcharge des opérateurs =, + et – de la classe Voiture (Essai1(), Essai2() et Essai3())**

Dans un premier temps, on vous demande de surcharger les opérateurs =, + et - de la classe Voiture, permettant d'exécuter un code du genre :

```
Modele m("208 Access 3P 1.4 HDi",70,Essence,14600.0f);
Voiture v1,v2("208_MrDugenou",m),v3;

v1 = v2 ;

Option op1("0MM0","Peinture metallisee",450.0f) ;
Option op2("ZH75","Jantes alliage 15 pouces",450.0f) ;

v3 = v2 + op1 ; // attention, v2 doit rester inchangé
v3 = op2 + v3 ; // on ajoute l'option op2 à la voiture v3
v3.Affiche() ;
v3 = v3 - op1 ; // on retire l'option op1 de la voiture v3
v3 = v3 - "ZH75" ; // on retire l'option "ZH75" de la voiture v3
v3.Affiche() ;
```

Bien sûr, on portera une attention particulière sur le tableau de pointeurs afin que deux objets de type Voiture ne pointent pas vers les mêmes objets Option...

#### **b) Surcharge des opérateurs de comparaison de la classe Voiture (Essai4())**

Il peut être intéressant pour le client de comparer le prix de deux projets de voiture (avec les options) afin qu'il puisse choisir celui qui lui convient le mieux. On vous demande donc de surcharger les opérateurs <, > et == de la classe Voiture, permettant de comparer les prix de deux projets et d'exécuter un code du genre

```
Voiture v1,v2 ;
...
if (v1 > v2) cout << message ;
else if (v1 == v2) cout << autre message;
else ...
```

**c) Surcharge des opérateurs d'insertion/extraction (Essai5(), Essai6() et Essai7())**

On vous demande à présent de surcharger les opérateurs << et >> des classes Modele et Option, et l'opérateur << de la classe Voiture, ce qui permettra d'exécuter un code du genre :

```
Modele m;
Voiture v1;
Option op1, op2 ;

cin >> m ;
cout << m ;
cin >> op1 ;
cout << op1 ;
...
cout << "Projet de Mr Dugenou :" << v1 << endl;
cout << op1 << op2 << endl ;
```

**d) Surcharge des opérateurs de pré et post-décrément de Option (Essai8())**

Il est possible qu'au cours de la conception d'un projet, le vendeur accepte de baisser le prix d'une option pour faire plaisir au client et s'assurer de la réussite de la vente ☺. On vous demande donc de programmer les opérateurs de post et pré-décrément de la classe Option. Ce qui permettra d'exécuter le code suivant :

```
Option op1("0MM0","Peinture metallisee",450.0f) ;
Option op2("ZH75","Jantes alliage 15 pouces",450.0f) ;

cout << --op1 << endl ;
cout << op2-- << endl ;
cout << op2 << endl ;
```

Une décrément de 50,00 euros.

**e) Surcharge de l'opérateur [] de la classe Voiture (Essai9())**

Enfin, on vous demande de programmer l'opérateur [] de la classe Voiture retournant un pointeur vers l'option dont l'indice est passé en paramètre à l'opérateur. Ce qui permettra d'exécuter le code suivant :

```
Voiture v(...);
...
for (int i=0 ; i<5 ; i++)
{
    if (v[i] != NULL) cout << *(v[i]) << endl;
    else cout << "---" << endl;
}
```

## Etape 4 (Test4.cpp) :

### Associations de classes : héritage et virtualité

Nous allons à présent aborder la modélisation des personnes intervenant dans notre application. Trois types de personne interviennent :

- Les clients qui ont un nom, un prénom, un numéro de client et, pour faire simple, un numéro de GSM.
- Les vendeurs qui sont des employés de la concession automobile et qui ont un nom, un prénom, un numéro d'employé, et un couple (login-mot de passe) pour pouvoir utiliser l'application.
- Les administratifs qui sont des employés de la concession automobile et qui ont un nom, un prénom, un numéro d'employé, et un couple (login-mot de passe) pour pouvoir utiliser l'application.

#### a) Héritage : une classe de base Personne (Essai1())

On remarque tout de suite que les trois intervenants ont un point commun, ce sont des personnes (on s'en serait douté ☺). L'idée est alors de concevoir une hiérarchie de classes, par héritage, dont la classe de base regroupera les caractéristiques communes aux trois intervenants, à savoir leur nom et prénom. Cette classe de base, qui pourrait à priori être réutilisée dans une autre application, sera la classe **Personne** et contiendra (dans un souci de simplification) les deux variables membres suivantes :

- Un **nom** : chaîne de caractères (**string**).
- Un **prénom** : chaîne de caractères (**string**).

On vous demande créer cette classe avec les constructeurs habituels, les accesseurs et les opérateurs <<, >> et =.

#### b) Héritage : une classe abstraite héritée

Les intervenants de notre garage sont des personnes mais qui disposent en plus d'un numéro d'identification. Cependant, la notion d'intervenant est vague. Dès lors, on vous demande de créer la classe **abstraite Intervenant** qui hérite de la classe **Personne** et qui dispose de

- Un **numéro** : un entier (**int**) qui identifie l'intervenant de manière unique. Bien sûr, on n'oubliera pas les accesseurs correspondants.
- Les **méthodes virtuelles pures Tuple()** et **ToString()** décrites plus loin
- un opérateur =

Etant donné que cette classe est **abstraite**, il est impossible de l'instancier et donc de réaliser des tests actuellement. Elle sert simplement à donner aux futures classes héritées :

- du **code commun et un service** : le numéro de l'employé
- une **interface commune** : toutes les classes héritées auront les méthodes Tuple et ToString

**c) Héritage : les classes dérivées de la hiérarchie (Essai2() et Essai3())**

Un client est un intervenant du garage qui a, en plus, un numéro de GSM. On vous demande donc de programmer la classe **Client**, qui hérite de la classe **Intervenant**, et qui présente, en plus, la variable membre suivante :

- Un **gsm** : une chaîne de caractères (**string**). Par exemple : « 0498/74.25.36 »

Vendeur et administratifs ne sont pas tellement différents par leurs caractéristiques, ce qui les différencie, c'est leur fonction. Donc, on vous demande de programmer la classe **Employe**, qui hérite de la classe **Intervenant**, et qui présente, en plus, les variables membres suivantes :

- Un **login** : chaîne de caractères (**string**).
- Un **mot de passe** : chaîne de caractères (**string \***). La variable membre **motDePasse** sera un **pointeur de type string**. A la création, un employé n'aura pas de mot de passe et la variable membre motDePasse aura la valeur NULL. La méthode void setMotDePasse(string mdp) allouera dynamiquement un objet de type string contenant le mot de passe (mdp) reçu. On vous demande également de programmer la méthode d'instance **void ResetMotDePasse()** qui remet la variable motDePasse à NULL.
- Une **fonction** : chaîne de caractères (**string**) pouvant contenir « Vendeur » ou « Administratif » selon le cas.

On redéfinira bien entendu les méthodes de la classe de base lorsque c'est nécessaire, par exemple les opérateurs d'affectation = ainsi que <<, sans oublier les constructeurs et destructeurs.

**d) Mise en évidence de la virtualité (Essai4() et Essai5())**

Les deux classes dérivées (**Client** et **Employe**) doivent donc redéfinir les méthodes suivantes :

- **string ToString() const** : qui retourne une chaîne de caractères propre à chaque intervenant  
 Client → « [Cx] nom prenom » (Ex : « [C3] Wagner Jean-Marc », 3 est le numéro du client Jean-Marc Wagner)  
 Employe → « [Vx] nom prenom » pour un vendeur et « [Ax] nom prenom » pour un administratif (Exemples : « [V2] Issier Pol » pour le vendeur Pol Issier dont le numéro est 2 et « [A7] Coptere Eli » pour l'administratif Eli Coptere dont le numéro est 7)
- **string Tuple() const** : qui retourne une chaîne de caractères propre à chaque intervenant  
 Client → « numero;nom;prenom;gsm »  
 (Exemple : « 3;Wagner;Jean-Marc;0497/17.25.24 »)  
 Employe → « numero;nom;prenom;fonction »  
 (Exemple : « 2;Issier;Pol;Vendeur »)

Le **caractère de séparation** entre les différents champs d'un tuple doit obligatoirement être un ; pour la suite du projet.

On vous demande de bien comprendre la notion de méthodes virtuelles (Essai4()) et du dynamic-cast du C++ (Essai5()).

e) Variables membres statiques (Essai6())

La variable membre **fonction** de la classe **Employe** ne peut pas prendre n'importe quelle valeur ; elle ne peut prendre que les valeurs « Vendeur » ou « Administratif ». Donc, on vous demande d'ajouter à la classe **Employe** deux **variables membres statiques constantes** du type **chaînes de caractères (string)**. Celles-ci s'appelleront **ADMINSTRATIF** et **VENDEUR** et auront pour valeur respective « Administratif » et « Vendeur ».

## Etape 5 (Test5.cpp) :

### Les exceptions

On demande de mettre en place une structure minimale de gestion des erreurs propres aux classes développées jusqu'ici. On va donc imaginer la hiérarchie de classes d'exception suivante :

- **Exception** : classe d'exception de base ne contenant qu'une seule variable membre **message** de type chaîne de caractères (**string**) destinée à contenir un message d'erreur à l'intention de l'utilisateur. Cette classe va servir de base aux classes d'exceptions suivantes et à toute autre classe d'exception qui pourrait être utile à l'application finale.
- **OptionException** : classe héritée de **Exception** qui ne contient aucune variable membre en plus mais qui en constitue une spécialisation pour les erreurs liées aux options d'une voiture. Cette exception sera lancée par la classe **Option** elle-même lorsque
  - on tente de modifier une option existante (via les setters) ou créer une nouvelle option avec de mauvais paramètres : le code d'une option doit comporter exactement 4 caractères, l'intitulé d'une option ne peut être vide et le prix d'une option doit être positif.
  - on tente de diminuer le prix d'une option (operator--) en vue de rendre le prix négatif.
  - on tente de saisir au clavier (operator>>) une option dont les paramètres encodés ne sont pas valides.

Cette exception est également lancée par la classe **Voiture** lorsque

- on tente d'ajouter une option alors que la voiture contient déjà 6 options.
- on tente d'ajouter une option qui est déjà présente dans la voiture (on se contentera de vérifier le code de l'option)
- on tente de supprimer une option qui n'est pas présente.

- **PasswordException** : classe héritée de **Exception** qui contient en plus une variable membre **code** de type **int** contenant un code d'erreur à l'intention du programmeur qui pourrait agir en conséquence. Ce code d'erreur pourra prendre l'une des 4 valeurs
  - **INVALID\_LENGTH** : lorsque la taille de mot de passe est strictement inférieure à 6.
  - **ALPHA\_MISSING** : lorsque le mot de passe ne contient pas au moins une lettre.
  - **DIGIT\_MISSING** : lorsque le mot de passe ne contient pas au moins un chiffre.
 Ces 3 premiers codes d'erreur sont utilisés par la méthode **setMotDePasse** de **Employe**.
  - **NO\_PASSWORD** : lorsque l'on tente de récupérer le mot de passe d'un employé (via la méthode **getMotDePasse()**) qui ne possède pas encore de mot de passe (pointeur **motDePasse** NULL).

Le fait d'insérer la gestion d'exceptions implique qu'elles soient récupérées et traitées lors des tests effectués en première partie d'année (**il faudra donc compléter le jeu de tests Test5.cpp** → utilisation de **try**, **catch** et **throw**), mais également dans l'application finale.

## Etape 6 (Test6.cpp) : Les containers et les templates

### a) L'utilisation future des containers

On conçoit sans peine que notre future application va utiliser des containers mémoire divers qui permettront par exemple de contenir les clients, les options, ou les employés. Nous allons ici mettre en place une base pour nos containers. Ceux-ci seront construits via une hiérarchie de classes templates.

### b) Un container typique : le vecteur (Essai1() et Essai2())

Le cœur de notre hiérarchie va être un vecteur « dynamique ». Ce vecteur sera alloué une fois pour toute lors de l'instanciation du conteneur. Ce vecteur va être encapsulé dans une **classe Vecteur template** contenant comme variables membres

- le pointeur vers le début du vecteur : la variable **v** de type **T\***.
- la capacité maximale du vecteur : variable **\_sizeMax** de type **int**.
- le nombre d'éléments présents dans le vecteur : variable **\_size** de type **int**.

La classe **Vecteur** aura donc la structure de base suivante :

```
template<class T> class Vecteur
{
protected :
    T * v;
    int _sizeMax;
    int _size;
    ...
};
```

La classe **Vecteur** devra disposer des méthodes suivantes :

- Un **constructeur par défaut** permettant d'allouer dynamiquement le vecteur **v** à une taille par défaut **\_sizeMax = 10**. La variable **\_size** est alors initialisée à 0.
- Un **constructeur d'initialisation** prenant en paramètre un entier n et permettant d'allouer dynamiquement **v** à une taille **\_sizeMax = n**. La variable **\_size** est initialisée à 0.
- Un **constructeur de copie**.
- Un **destructeur** permettant de libérer correctement la mémoire.
- La méthode **size()** retournant le nombre d'éléments présents dans le vecteur.
- La méthode **sizeMax()** retournant la capacité du vecteur (**\_sizeMax**).
- La méthode **Affiche()** permettant de parcourir le vecteur et d'afficher chaque élément de celui-ci.
- La méthode **void insere(const T & val)** permettant d'insérer un nouvel élément **à la fin du vecteur**, c'est-à-dire à la première case libre.
- La méthode **T retire(int ind)** qui permet de supprimer et de retourner l'élément situé à l'indice **ind** dans le vecteur. Les valeurs possibles pour ind sont 0, ..., size()-1. Après suppression d'un élément, tous les éléments situés à droite de celui-ci doivent être décalés vers la gauche, et la variable **\_size** doit être mise à jour.
- Un **opérateur =** permettant de réaliser l'opération « vecteur1 = vecteur2 ; » sans altérer le vecteur2 et de telle sorte que si le vecteur1 est modifié, le vecteur2 ne l'est pas et réciproquement.
- Un **opérateur [ ]** permettant de retourner (par référence constante) l'élément situé à l'indice passé en paramètre.

Dans un premier temps, vous testerez votre classe Vecteur avec des **entiers** (**Essai1()**), puis ensuite avec des objets de la classe **Client** (**Essai2()**).

Bien sûr, on travaillera, comme d'habitude, en fichiers séparés afin de maîtriser le problème de l'instanciation des templates.

### c) Parcourir, récupérer les éléments d'un vecteur : l'itérateur de vecteur (Essai3() et Essai4())

Dans l'état actuel des choses, nous pouvons ajouter/supprimer des éléments à un vecteur mais nous n'avons aucun moyen simple et standardisé de le parcourir, élément par élément. La notion d'itérateur va nous permettre de réaliser cette opération.

On vous demande donc de créer la classe **Iterateur** qui sera un **itérateur** de la classe **Vecteur**, et qui comporte, au minimum, les méthodes et opérateurs suivants :

- **reset()** qui réinitialise l'itérateur au début du vecteur.
- **end()** qui retourne le booléen true si l'itérateur est situé au bout du vecteur.
- **Opérateur ++** qui déplace l'itérateur vers la droite.
- **Opérateur de casting ()** qui retourne (par valeur) l'élément « pointé » par l'itérateur.



On vous demande donc d'utiliser la classe `Iterateur` afin de vous faciliter l'accès aux containers. Son usage correct sera vérifié lors de l'évaluation finale.

**d) Le vecteur trié : BONUS (Essai5(), Essai6(), Essai7(), Essai8())**

On vous demande à présent de programmer la classe template VecteurTrie qui hérite de la classe Vecteur et qui redéfinit la méthode `insere` de telle sorte que l'élément ajouté au vecteur soit inséré au bon endroit dans le vecteur, c'est-à-dire en respectant l'ordre défini par les opérateurs de comparaison de la classe template.

Dans un premier temps, vous testerez votre classe `VecteurTrie` avec des **entiers**, puis ensuite avec des objets de la classe **Client**. Ceux-ci devront être triés par ordre alphabétique.

### Etape 7 (Test7.cpp) :

#### Première utilisation des flux

Il s'agit ici d'une première utilisation des flux en distinguant les flux caractères (manipulés avec les opérateurs `<<` et `>>`) et **les flux bytes (méthodes `write` et `read`)**. Dans cette première utilisation, nous ne traiterons que des flux bytes.

**a) La classe Voiture se sérialise elle-même**

On demande de compléter la classe **Voiture** avec les deux méthodes suivantes :

- ♦ **Save()** permettant d'enregistrer dans un fichier toutes les données de la voiture (nom du projet, modèle et options) et cela champ par champ. Ce fichier sera un fichier binaire (utilisation des méthodes `write` et `read`) dont le nom sera obtenu par la concaténation du nom du projet avec l'extension « .car ». Exemple : « Projet208\_MrDugenou.car ».
- ♦ **Load(string nomFichier)** permettant de charger toutes les données relatives à une voiture enregistrée dans le fichier dont le nom est passé en paramètre.

Afin de vous aider dans le développement, on vous demande d'utiliser l'encapsulation, c'est-à-dire de laisser chaque classe gérer sa propre sérialisation. En d'autres termes, on vous demande d'ajouter aux classes **Modele** et **Option** les méthodes suivantes :

- **void Save(ofstream & fichier) const** : méthode permettant à un objet de s'écrire lui-même sur le flux fichier qu'il a reçu en paramètre.
- **void Load(ifstream & fichier)** : méthode permettant à un objet de se lire lui-même sur le flux fichier qu'il a reçu en paramètre.

Ces méthodes seront appelées par les méthodes Save et Load de la classe Voiture lorsqu'elle devra enregistrer ou lire sa variable membre de type Modele et/ou les différentes options.

### **Important**

Tous les enregistrements seront de taille variable. Pour l'enregistrement d'une **chaîne de caractères** « chaîne » (de type **char \*** ; classe **Modele**), on enregistrera tout d'abord le **nombre de caractères** de la chaîne (strlen(chaine)) puis ensuite la **chaîne elle-même**. Ainsi, lors de la lecture dans le fichier, on lit tout d'abord la taille de la chaîne et on sait directement combien de caractères il faut allouer et lire ensuite.

Pour les **chaînes de caractères** de type **string** (classes **Option** et **Voiture**), il faut également faire attention car un objet de type string a une taille pouvant varier et un code du genre

```
string chaine = « Je suis la chaine a enregistrer »;
fichierOut.write((char*)&chaine,sizeof(string));
...
string chaineLue;
fichierIn.read((char*)&chaineLue,sizeof(string));
```

ne fonctionnera pas. Il est nécessaire, comme pour les **char\*** d'enregistrer d'abord le nombre de caractères de la chaîne puis la chaîne elle-même :

```
int taille = chaine.size();
fichierOut.write((char*)&taille,sizeof(int));
fichierOut.write((char*)chaine.data(),taille*sizeof(char));
...
int t;
fichierIn.read((char*)&t,sizeof(int));
chaine2.resize(t); // preuve que le type chaine est de taille variable...
fichierIn.read((char*)chaine2.data(),t*sizeof(char));
```

## Etape 8 (Test8a.cpp et Test8b.cpp) :

### Un conteneur pour nos conteneurs : **La classe Garage**

#### a) Mise en place de la classe Garage (Test8a.cpp)

La concession automobile que nous voulons gérer regroupe des employés, des clients, des modèles et des options de voiture. On vous demande donc de regrouper toutes ces données au sein de la même classe appelée **Garage** :

```
class Garage
{
    private:
        VecteurTrie<Employe> employes;    // ou Vecteur<Employe>
        VecteurTrie<Client> clients;      // ou Vecteur<Client>
        Vecteur<Modele>      modeles;
        Vecteur<Option>      options;

    public:
        Garage();
        void ajouteModele(const Modele & m);
        void afficheModelesDisponibles() const;
        Modele getModele(int indice);

        void ajouteOption(const Option & o);
        void afficheOptionsDisponibles() const;
        Option getOption(int indice);

        void ajouteClient(string nom, string prenom, string gsm);
        void afficheClients() const;
        void supprimeClientParIndice(int ind);
        void supprimeClientParNumero(int num);

        void ajouteEmploye(string nom, string prenom, string login, string fonction);
        void afficheEmployes() const;
        void supprimeEmployeParIndice(int ind);
        void supprimeEmployeParNumero(int num);
};
```

Cette classe comporte dans un premier temps :

- Les **vecteurs triés** d'employés (**ordre alphabétique**), et de clients (**ordre alphabétique**), dans le cas où le bonus de l'étape 6 a été réalisé. On se contentera de simples **vecteurs** dans le cas contraire.
- Les **vecteurs** de modèles et d'options.

- Un **constructeur par défaut**. Inutile de faire un constructeur de copie car il n'existera qu'un seul objet instance de la classe Garage dans l'application.
- Les méthodes **void ajouteModele(...)** et **void ajouteOption(...)** permettent d'ajouter un modèle ou une option dans le bon conteneur.
- Les méthodes **afficheOptionsDisponibles()** et **void afficheModelesDisponibles()** permettent d'afficher le contenu des conteneurs correspondants.
- Les méthodes **getModele(int indice)** et **getOption(int indice)** qui permettent de retourner une copie du modèle ou de l'option situé(e) dans le conteneur correspondant à l'indice passé en paramètre.
- Les méthodes **void ajouteXXX(...)** permettent d'ajouter un XXX (XXX = Client ou Employe) au bon conteneur en recevant en paramètre les paramètres adéquats. Remarquez que le numéro (de client/employé) entier n'est pas passé en paramètre. On vous demande donc d'ajouter à la classe Intervenant une variable membre statique entière appelée numCourant qui sera utilisée et incrémentée de 1 à chaque ajout. Ceci assurera donc l'unicité du numéro des intervenants.
- Les méthodes **void afficheXXX()** (XXX = Clients ou Employes) qui permettent d'afficher le contenu des différents conteneurs. Rappelez-vous que la classe VecteurTrie (ou Vecteur) dispose d'une méthode Affiche()...
- Les méthodes **void supprimeXXXParIndice(int ind)** qui permettent de supprimer le XXX (XXX = Client ou Employe) dont l'indice dans son vecteur est **ind**.
- Les méthodes **void supprimeXXXParNumero(int num)** qui permettent de supprimer le XXX (XXX = Client ou Employe) dont le numéro est **num**. Ceci peut se faire très facilement à l'aide de l'itérateur...

#### b) La classe Garage en tant que singleton (Test8b.cpp)

La classe Garage contenant toutes les données de notre future application, on se rend bien compte qu'elle ne sera instanciée qu'une seule fois. Une telle classe est appelée un « **singleton** ».

« Le **singleton** est un **patron de conception** (« **Design pattern** ») dont l'objectif est de restreindre l'instanciation d'une classe à un seul objet. On fournira un accès global à cet objet. Il est utilisé lorsqu'on a besoin d'exactly un objet pour coordonner les opérations d'un système. » (wikipedia)

En C++ (et donc dans notre cas pour la classe Garage), une manière de faire est

- Rendre le **constructeur par défaut privé** : cela empêchera un programmeur d'instancier un ou plusieurs objets de classe Garage.

- Placer dans la classe Garage une **variable statique privée de type Garage**, que l'on appellera **instance** (il s'agira de l'unique instance de la classe Garage). Cet objet sera instancié par le constructeur par défaut.
- Cet objet étant privé, on lui donnera un accès « global » en écrivant une **méthode statique publique Garage& getInstance()** qui retourne une référence vers l'objet **instance** qui pourra donc être manipulé de n'importe où dans l'application.
- Afin d'éviter qu'un programmeur puisse obtenir une copie de cette instance, on déclarera en **privé** (mais on ne les définira pas dans Garage.cpp !!) un **constructeur de copie** et un **opérateur =** dans Garage.h.

Dans le même ordre d'idée, l'application sera capable de gérer (via l'interface graphique) un seul objet de classe Voiture que l'on appellera **projetEnCours**. On vous demande donc d'ajouter à la classe **Garage** :

- Une **instance statique privée de la classe Voiture**, appelée **projetEnCours**.
- Une **méthode statique publique Voiture& getProjetEnCours()** retournant la référence vers cet objet qui pourra donc être manipulé de n'importe où dans l'application.
- Une **méthode statique publique voidResetProjetEnCours()** permettant de redonner à l'objet **projetEnCours** ses variables membres par défaut et de lui supprimer toutes ses options.

Remarquez que lorsque vous aurez terminé les modifications de la classe Garage afin de la transformer en singleton, le programme Test8a.cpp ne compilera plus. (Pourquoi 😊 ?)

## Etape 9 (utilisation de **InterfaceQt.tar** fourni)

### Mise en place de l'interface graphique : **Introduction à Qt**

#### a) Introduction

Nous allons à présent mettre en place l'interface graphique. Celle-ci sera construite en utilisant la librairie graphique Qt. Sans entrer dans les détails de cette librairie, il faut savoir qu'une **fenêtre** est représentée par une **classe** dont

- les variables membres sont les composants graphiques apparaissant dans la fenêtre : les boutons, les champs de texte, les checkboxs, les tables, ...
- les méthodes publiques permettent d'accéder à ses composants graphiques, soit en récupérant les données qui sont encodées par l'utilisateur, soit en y insérant des données.

L'interface graphique de notre application sera la classe **ApplicGarageWindow** qui a été construite, pour vous, à l'aide de l'IDE QtCreator. Cette classe est fonctionnelle mais il ne s'agit que d'une **coquille vide** qui va permettre de manipuler le seul objet de classe **Garage** de notre application. La classe **ApplicGarageWindow** est fournie par les fichiers

- **applicgaragewindow.h** qui contient la définition de la classe et la déclaration de ses méthodes
- **applicgaragewindows.cpp** qui contient la définition de ses méthodes.

Seuls ces deux fichiers pourront être modifiés par vous. Les autres fichiers fournis (dans **InterfaceQt.tar**) ne pourront en aucun cas être modifiés. Le main de votre application est le fichier **main.cpp** également fourni et ne devra pas être modifié. Un makefile est également fourni pour la compilation. **A vous de combiner votre makefile actuel et le makefile fourni.**

La classe **ApplicGarageWindow** contient déjà un ensemble de méthodes fournies (et que vous ne devez donc pas modifier) afin de vous faciliter l'accès aux différents composants graphiques.

Les méthodes que vous devez modifier contiennent le commentaire **// TO DO**. Elles correspondent aux différents boutons et items de menu de l'application et correspondent toutes à une action demandée par l'utilisateur, comme par exemple « Nouveau modèle », « Nouvelle option », ...

Pour l'affichage de messages dans des **boîtes de dialogue** ou des saisies de int/float/chaînes de caractères via des boîtes de dialogues, vous disposez des méthodes (**que vous ne devez pas modifier mais juste utiliser**) :

- void      dialogueMessage(const char\* titre,const char\* message);
- void      dialogueErreur(const char\* titre,const char\* message);
- string    dialogueDemandeTexte(const char\* titre,const char\* question);
- int       dialogueDemandeInt(const char\* titre,const char\* question);
- float     dialogueDemandeFloat(const char\* titre,const char\* question);

Etant une classe C++ au sens propre du terme, vous pouvez ajouter, à la classe **ApplicGarageWindow**, des variables et des méthodes membres en fonction de vos besoins.

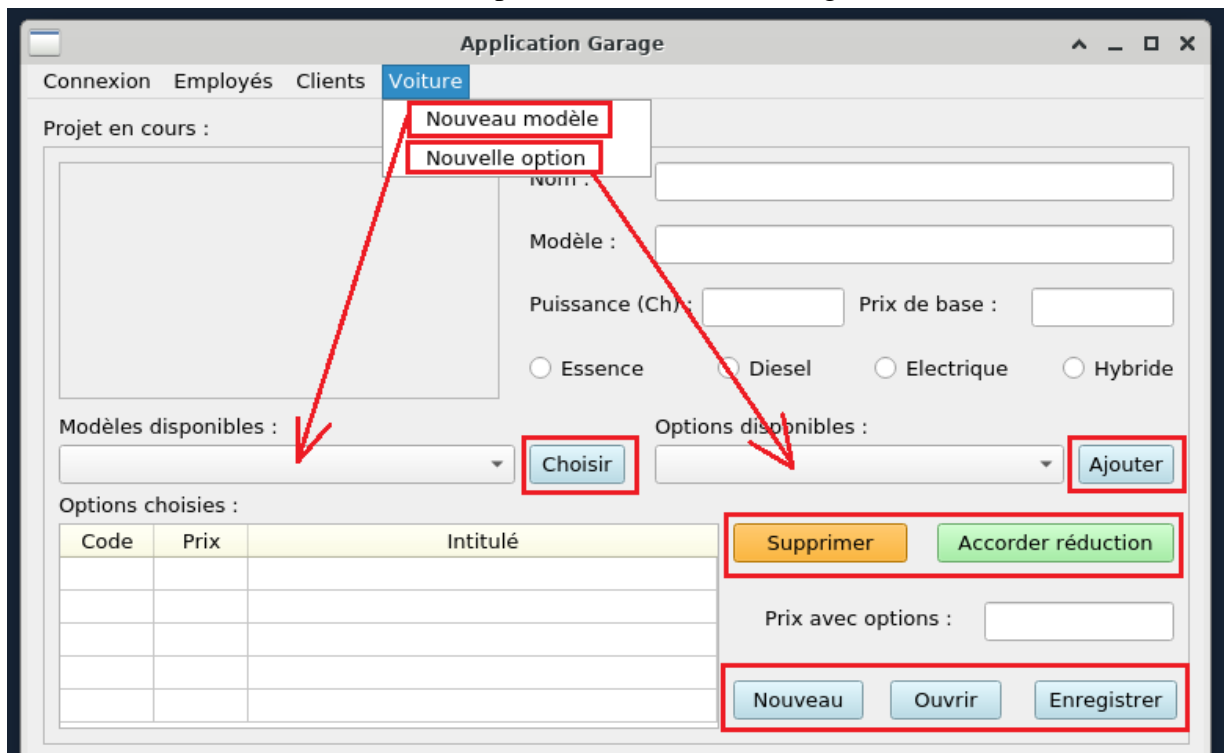
### b) Mise en place des premières fonctionnalités

Les premières fonctionnalités demandées ici correspondent à l'ajout de nouveaux modèles et options disponibles pour la conception de projet de voiture, ainsi qu'à la gestion du projet en cours. Ces fonctionnalités ont déjà été programmées à l'étape précédente dans la classe Garage. Il suffit donc de manipuler l'unique instance statique de cet objet à l'aide de l'interface graphique. Toute la logique de l'application, dite « logique métier », se trouve dans la classe Garage.

Etant des objets statiques, l'unique instance de la classe Garage et le projet (de voiture) en cours seront accessibles partout dans l'application à partir de la classe Garage elle-même :

- **Garage & Garage::getInstance()** : qui retourne la référence vers le singleton de notre application.
- **Voiture & Garage::getProjetEnCours()** : qui retourne la référence vers l'objet Voiture représentant le projet de voiture courant.

Les fonctionnalités demandées ici correspondent aux encadrés rouges ci-dessous :



Il s'agit donc de modifier les méthodes suivantes de la classe **ApplicGarageWindow** :

- void on\_actionNouveau\_modele\_triggered();
- void on\_actionNouvelle\_option\_triggered();

pour les items de menu « Voiture », et

- `void on_pushButtonChoisirModele_clicked();`
- `void on_pushButtonAjouterOption_clicked();`
- `void on_pushButtonSupprimerOption_clicked();`
- `void on_pushButtonReduction_clicked();`
- `void on_pushButtonEnregistrer_clicked();`
- `void on_pushButtonOuvrir_clicked();`
- `void on_pushButtonNouveau_clicked();`

pour les boutons.

Notons tout d'abord qu'afin de gérer la petite image du modèle de voiture, on vous demande d'ajouter à la classe **Modele** une variable membre **image** (type **string**) contenant le nom du fichier de l'image (exemple : « 208.jpg » ; le chemin d'accès à ce fichier est codé en dur dans `applicaragewindow.cpp`). Les prototypes des constructeurs de la classe **Modele** doivent rester inchangés afin que les jeux de tests précédents continuent de fonctionner (XXX). Par contre, n'oubliez pas d'adapter le contenu des autres méthodes afin de tenir compte de cette nouvelle variable membre.

Pour **ajouter un nouveau modèle** en cliquant sur l'item de menu « Nouveau modèle » correspondant, vous devez (dans la méthode **on\_actionNouveau\_modele\_triggered**)

1. obtenir le nom, la puissance, le moteur, le prix de base et l'image du modèle par l'intermédiaire de boîtes de dialogues (voir code fourni). Si une des données encodées par l'utilisateur est invalide, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char\* titre, const char\* message)**.
2. instancier un objet de la classe **Modele** avec les paramètres obtenus et l'insérer dans le conteneur adéquat en appelant la méthode **void ajouteModele(const Modele& m)** du singleton de la classe **Garage**.
3. mettre à jour le combobox des modèles disponibles dans l'interface graphique. Pour cela, vous devez utiliser la méthode **void ajouteModeleDisponible(const char \*nom, float prixDeBase)** qui ajoute un nouvel item à la liste déroulante.

L'**ajout de nouvelles options** disponibles doit se faire de manière tout à fait analogue. Remarquez simplement que la classe **Option** est susceptible de lancer **OptionException** qu'il vous faut gérer à l'aide d'un **try... catch**. En cas de paramètre(s) invalide(s), vous devez à nouveau faire apparaître une boîte de dialogue d'erreur.

Le clic sur le bouton « Choisir » permet **d'attribuer au projet en cours le modèle sélectionné** dans la combobox des modèles disponibles. Pour ce faire, vous devez (dans la méthode **void on\_pushButtonChoisirModele\_clicked**) :

1. récupérer l'indice du modèle sélectionné dans la combobox. Pour cela vous disposez de la méthode **int getIndiceModeleSelectionneCombobox()** qui retourne l'indice dans la combobox du modèle sélectionné (il s'agit également de son indice dans le vecteur `modeles` de la classe **Garage**), et -1 si aucun modèle n'est sélectionné. Si aucun modèle n'est sélectionné, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char\* titre, const char\* message)**.
2. Affecter au projet en cours le modèle récupéré via son indice.



- mettre à jour l'interface graphique du projet en cours en utilisant la méthode **void setModele(string nom, int puissance, int moteur, float prixDeBase, string image)** et **void setPrix(float prix)**. Cette dernière méthode permet d'afficher le prix calculé du projet tenant compte des options (méthode **getPrix()** de la classe Voiture).

Le clic sur le **bouton « Ajouter »** permet **d'ajouter au projet en cours l'option sélectionnée** dans la combobox des options disponibles. Pour ce faire, vous devez (dans la méthode **void on\_pushButtonAjouterOption\_clicked**) :

- Récupérer l'indice de l'option sélectionnée dans la combobox. Pour cela vous disposez de la méthode **int getIndiceOptionSelectionneeCombobox()** qui retourne l'indice dans la combobox de l'option sélectionnée (il s'agit également de son indice dans le vecteur options de la classe Garage), et -1 si aucune option n'est sélectionnée. Si aucune option n'est sélectionnée, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char\* titre, const char\* message)**.
- Ajouter au projet en cours l'option récupérée via son indice. Notez à nouveau qu'une **OptionException** peut être générée et qu'il vous faut la traiter.
- Mettre à jour, dans l'interface graphique, la table des options du projet en cours en utilisant la méthode **void setTableOption(int indice, string code = "", string intitule = "", float prix = -1.0)**. Cette méthode utilisée avec ses paramètres par défaut vide la ligne correspondante (qui correspond à l'indice passé en paramètre : 0,1,2,3 ou 4) de la table.
- Mettre à jour l'affichage du prix du projet en cours tenant à présent compte de l'option ajoutée.

Le clic sur le **bouton « Supprimer »** permet **de supprimer du projet en cours l'option sélectionnée** dans la table des options du projet en cours. Pour ce faire, vous devez (dans la méthode **on\_pushButtonSupprimerOption\_clicked**) :

- Récupérer l'indice de l'option sélectionnée dans la table. Pour cela vous disposez de la méthode **int getIndiceOptionSelectionneeTable()** qui retourne l'indice dans la table de l'option à supprimer, et -1 si aucune option n'est sélectionnée. Si aucune option n'est sélectionnée, vous pouvez afficher une boîte de dialogue d'erreur en utilisant la méthode **void dialogueErreur(const char\* titre, const char\* message)**.
- Supprimer du projet en cours l'option sélectionnée. Rappelez-vous que la classe Voiture dispose d'un **opérateur []** vous permettant de récupérer un pointeur vers l'option dont l'indice est passé en paramètre, et d'une méthode **void RetireOption()**.
- Mettre à jour, dans l'interface graphique, la table des options du projet en cours en utilisant la méthode **void setTableOption(int indice, string code = "", string intitule = "", float prix = -1.0)**. Cette méthode utilisée avec ses paramètres par défaut vide la ligne correspondante (qui correspond à l'indice passé en paramètre : 0,1,2,3 ou 4) de la table.
- Mettre à jour l'affichage du prix du projet en cours tenant à présent compte de l'option retirée.

Le clic sur le **bouton « Accorder réduction »** permet **d'accorder une réduction de 50 euros sur l'option sélectionnée** dans la table des options du projet en cours. Pour ce faire, vous devez compléter la méthode **on\_pushButtonReduction\_clicked**. Le principe est exactement le même que lors de la suppression d'une option. Une fois l'option récupérée (via un pointeur), on peut utiliser

dessus l'**opérateur --**. Notez à nouveau qu'une **OptionException** peut être générée et qu'il vous faut la traiter.

Le clic sur le **bouton « Enregistrer »** permet **d'enregistrer le projet en cours sur disque**. Pour ce faire, vous devez (dans la méthode **on\_pushButtonEnregistrer\_clicked**) :

1. Récupérer le nom de projet encodé par l'utilisateur en utilisant la méthode **string getNomProjetEnCours()**
2. Affecter le nom récupéré au projet en cours
3. Utiliser la méthode **Save()** (déjà réalisée à l'étape 7) du projet en cours.

Le clic sur le **bouton « Ouvrir »** permet **de charger à partir du disque le projet en cours** dont le nom est encodé par l'utilisateur. Pour ce faire, vous devez (dans la méthode **on\_pushButtonOuvrir\_clicked**) :

1. Récupérer le nom de projet encodé par l'utilisateur en utilisant la méthode **string getNomProjetEnCours()**
2. Utiliser la méthode **Load(string nomFichier)** (déjà réalisée à l'étape 7) du projet en cours.
3. Mettre à jour l'interface graphique du projet en cours en utilisant les méthodes déjà utilisées plus haut.

Le clic sur le **bouton « Nouveau »** permet **de purger complètement le projet en cours** afin de pouvoir en créer un nouveau. Pour ce faire, vous devez (dans la méthode **on\_pushButtonNouveau\_clicked**) :

1. Utiliser la méthode **resetProjetEnCours()** de la classe Garage
2. Mettre à jour l'interface graphique du projet en cours en utilisant les méthodes déjà utilisées plus haut.

## Etape 10 :

### Importation de modèles et d'options : **Fichiers textes**

Nous allons à présent ajouter à l'application la possibilité d'importer automatiquement au démarrage un ensemble de modèles et d'options sans devoir les encoder manuellement à chaque fois. Ces modèles et options seront fournies sous la forme de **fichiers textes** au format « **csv** » : **Modeles.csv** et **Options.csv** (ces fichiers vous seront fournis au laboratoire et pourraient avoir des noms différents). Ces **fichiers textes** peuvent être créés/lus à l'aide de Excel mais également dans n'importe quel éditeur de texte. Par exemple, pour les modèles (fichiers **Modeles.csv**) nous avons

```
nom;puissance;moteur;image;prixDeBase
208 Active 1.2 PureTech 75;75;essence;208.jpg;18050
208 Active 1.5 BlueHdi 100;100;diesel;208.jpg;20250
208 Allure 1.2 PureTech 130;130;essence;208.jpg;22250
...
```

Chaque ligne représente donc un modèle et le caractère ';' est appelé le **séparateur**. Celui-ci pourrait être ':' ou encore ','. La première ligne est une ligne d'entête précisant l'intitulé de chaque champ.

Le fichier **Options.csv** est structuré de la même manière.

On vous demande donc d'ajouter à la classe **Garage** les méthodes suivantes :

- **void importeModeles(string nomFichier)** : qui importe les modèles présents dans le fichier csv dont le nom est passé en paramètre. Les modèles lus seront placés dans le conteneur de modèles de la classe **Garage**.
- **void importeOptions(string nomFichier)** : qui importe les options présentes dans le fichier csv dont le nom est passé en paramètre. Les options lues seront placées dans le conteneur d'options de la classe **Garage**.

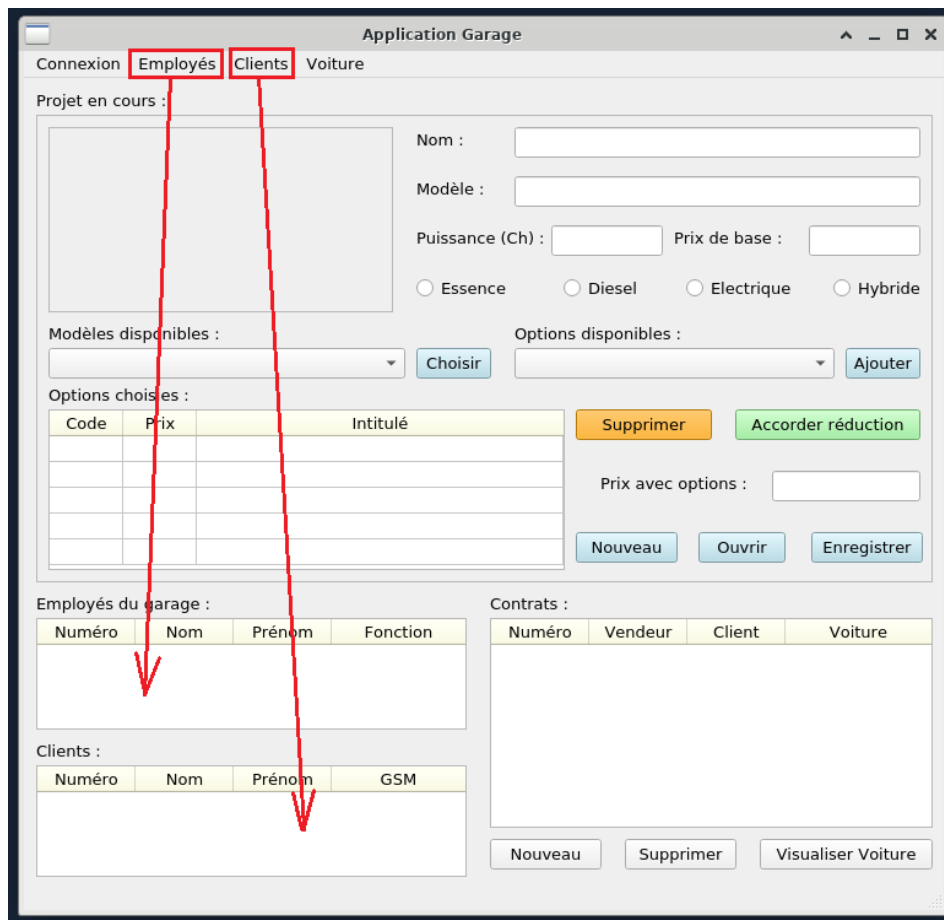
Lors du démarrage de l'application, ces méthodes seront appelées (par exemple dans le constructeur de la classe **ApplicGarageWindow**) afin de lire les deux fichiers csv s'ils existent. Il ne faudra pas oublier d'ajouter aux 2 comboboxs les modèles et options importées (méthodes **ajouteModeleDisponible** et **ajouteOptionDisponible**).

## Etape 11 :

### Ajout/Suppression d'employés et de clients / Gestion des rôles

#### a) Ajout/Suppression d'employés et de clients

Il s'agit ici de gérer l'ajout et la suppression d'employés (c'est-à-dire les utilisateurs de l'application) et de clients. Ces fonctionnalités ont déjà été programmées à l'étape 8a dans la classe **Garage**. Il suffit donc de manipuler notre objet singleton « Garage » à l'aide de l'interface graphique :



Il s'agit donc de modifier les méthodes suivantes de la classe **ApplicGarageWindow** :

- void on\_actionAjouterEmploye\_triggered();
- void on\_actionSupprimerEmploye\_par\_numero\_triggered();
- void on\_actionSupprimerEmploye\_selection\_triggered();

pour les items de menu « Employés », et

- void on\_actionAjouterClient\_triggered();
- void on\_actionSupprimerClient\_par\_numero\_triggered();
- void on\_actionSupprimerClient\_selection\_triggered() ;

pour les items de menu « Clients ».

Pour **ajouter un nouvel employé** en cliquant sur l’item de menu « Ajouter » du menu « Clients », vous devez (dans la méthode `on_actionAjouterEmploye_triggered`)

1. Demander à l’utilisateur le nom, le prénom, le login et la fonction du nouvel employé. Pour cela, vous disposez des méthodes `dialogueDemandeXXX(...)` permettant de demander à l’utilisateur, via une boîte de dialogue, un entier, un réel ou une chaîne de caractères. Si une donnée encodée par l’utilisateur est invalide, vous pouvez afficher une boîte de dialogue d’erreur en utilisant la méthode **`void dialogueErreur(const char* titre, const char* message)`**.
2. Appeler la méthode **`void ajouteEmploye(string nom, string prenom, string login, string fonction)`** de l’objet **garage**.
3. Mettre à jour la table des employés. Pour cela, vous devez
  - a. Vider la table des employés à l’aide de la méthode **`void videTableEmployes()`**.
  - b. Récupérer le vecteur trié des employés. L’**itérateur** vous permet de parcourir ce vecteur et d’en récupérer chaque élément. La fonction **`Tuple()`** de chaque objet `Employe` retournera le tuple qui pourra être inséré dans la table des employés à l’aide de la méthode **`void ajouteTupleTableEmployes(string tuple)`**.

Remarquez qu’un employé ajouté ne dispose d’aucun mot de passe. Il devra en choisir un lors de sa première connexion.

Le clic sur **l’item de menu « Supprimer par numéro » du menu « Employés » supprime un employé dont on demandera le numéro à l’utilisateur**. Pour ce faire, vous devez (dans la méthode `void on_actionSupprimerEmploye_par_numero_triggered`) :

1. Demander à l’utilisateur d’encoder un numéro. Pour cela, vous devez afficher une boîte de dialogue de saisie d’entier en utilisant la méthode **`int dialogueDemandeInt(const char* titre, const char* question)`**.
2. Appeler la méthode **`void supprimeEmployeParNumero(int num)`** de l’objet `garage`.
3. Mettre à jour la table des employés (voir ci-dessus).

Le clic sur le **l’item de menu « Supprimer sélection » du menu « Employés » supprime l’employé sélectionné dans la table**. Pour ce faire, vous devez (dans la méthode `void on_actionSupprimerEmploye_selection_triggered`) :

1. Récupérer l’indice de l’employé sélectionné dans la table. Pour cela vous disposez de la méthode **`int getIndiceEmployeSelectionne()`** qui retourne l’indice dans la table de l’employé sélectionné (il s’agit également de son indice dans le vecteur trié), et -1 si aucun employé n’est sélectionné. Si aucun employé n’est sélectionné, vous pouvez afficher une boîte de dialogue d’erreur en utilisant la méthode **`void dialogueErreur(const char* titre, const char* message)`**.
2. Appeler la méthode **`void supprimeEmployeParIndice(int ind)`** de l’objet `garage`.
3. Mettre à jour la table des employés (voir ci-dessus).

L’ajout/suppression des clients doit se faire de manière tout à fait analogue.

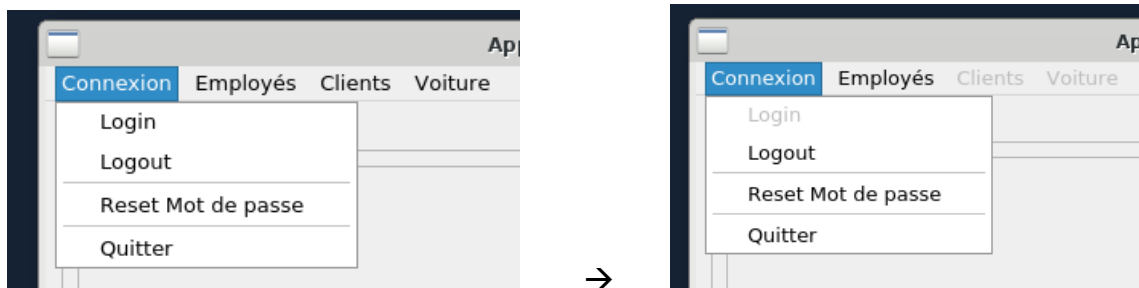
## b) Gestion du login/logout et des rôles des utilisateurs

Actuellement, tous les boutons et items de menu de l'application sont accessibles au démarrage de celle-ci. Dans ce qui vient, les **vendeurs** et les **administratifs** n'auront pas accès aux mêmes fonctionnalités (on parle alors de **rôle de chaque catégorie d'utilisateurs**) et devront se logger (à l'aide de leur login et d'un mot de passe) afin de pouvoir utiliser l'application.

Pour cela, on vous fournit la méthode **void setRole(int val)** de la classe **ApplicGarageWindow** :

- si **val = 0** : aucun utilisateur n'est loggé → tous les items de menu et les boutons sont inactifs mis à part les items « Login » et « Quitter » du menu « Connexion »
- si **val = 1** : un administratif est loggé et cela active les boutons et items de menu auxquels il a accès. Il peut notamment ajouter/supprimer des administratifs mais pas des clients, ni configurer un projet de voiture.
- Si **val = 2** : un vendeur est loggé et cela active les boutons et items de menu auxquels il a accès. Il peut créer un nouveau modèle ou option, configurer un projet de voiture, ajouter/supprimer des clients mais pas des administratifs.
- Si **val = 3** : tous les items de menu et boutons sont actifs (pour la phase de conception → valeur par défaut).

Une fois entré en session, on pourra appeler cette méthode afin de mettre à jour l'interface graphique en fonction du type d'employé qui s'est connecté :



Pour **entrer en session** en cliquant sur l'item de menu « **Login** » du menu « **Connexion** », vous devez (dans la méthode `on_actionLogin_triggered`)

1. Demander à l'utilisateur son login en utilisant une boîte de dialogue.
2. Balayer le conteneur d'employés à l'aide de l'**itérateur**. S'il n'existe pas d'employé avec le login encodé, la procédure de login s'arrête ici en affichant un message d'erreur dans une boîte de dialogue.
3. Mémoriser un **pointeur vers l'employé trouvé**. Cela peut se faire en ajoutant une variable membre statique publique de type `Employe*` dans la classe `garage`. Il s'agira d'un pointeur vers l'employé connecté tout au long de la session de l'employé.
4. Tenter de récupérer le mot de passe de l'employé trouvé via la méthode **`getMotDePasse()`** qui, pour rappel, lance une **`PasswordException`** si l'employé n'a pas de mot de passe. Donc 2 possibilités :
  - a. L'employé a déjà un mot de passe : On demande alors à l'utilisateur d'encoder un mot de passe via une boîte de dialogue. Le mot de passe récupéré et le mot de passe

encodé sont alors comparés. S'ils sont identiques, l'entrée en session est réussie. Sinon, un message d'erreur est affiché et la procédure de login se termine ici.

- b. L'employé n'a pas encore de mot de passe : On prévient alors l'employé qu'il n'a pas encore de mot de passe et qu'il va devoir en encoder un nouveau. On lui demande alors d'encoder un nouveau mot de passe. Celui-ci sera alors attribué à l'employé (accessible via le pointeur mémorisé) en utilisant la méthode **setMotDePasse(...)**. De nouveau, pour rappel, cette méthode lance une **PasswordException** si le mot de passe n'est pas un mot de passe valide. Si le mot de passe encodé est valide, on le signale à l'utilisateur et on lui propose de se connecter à nouveau. Sinon, la procédure de login se termine ici.

Une fois la procédure de login réussie, on dispose d'un pointeur vers l'employé connecté. On sait dès lors s'il s'agit d'un vendeur ou d'un administratif et on peut mettre à jour l'interface graphique en conséquence à l'aide de la méthode **setRole(...)** décrite plus haut. Vous pouvez également mettre à jour le titre de la fenêtre en utilisant la méthode **void setTitre(string titre)** de la classe **ApplicGarageWindow** afin d'afficher le nom de l'employé actuellement connecté.

Notez qu'il faut absolument au premier démarrage de l'application qu'il existe au moins un administratif présent dans le conteneur d'employés (administratif qui pourra en créer d'autres ou des vendeurs). On vous demande donc d'ajouter au conteneur d'employés un **administratif « ADMIN »** dans le constructeur par défaut de la classe **Garage**.

Un clic sur l'item de menu « **Logout** » du menu « **Connexion** » fait sortir de session l'employé actuellement connecté. Pour cela, vous devez (dans la méthode `on_actionLogout_triggered`) :

1. Modifier le titre de la fenêtre afin que le nom de l'employé connecté disparaisse.
2. Modifier l'interface graphique en appelant la méthode `setRole(0)`.
3. Remettre à NULL le pointeur vers l'employé actuellement connecté.

Un clic sur l'item de menu « **Reset Mot de passe** » du menu « **Connexion** » permet à l'employé actuellement connecté de « reseter » son mot. Pour cela, vous devez (dans la méthode `on_actionReset_Mot_de_passe_triggered`) :

1. Appeler, par l'intermédiaire du pointeur mémorisé, la méthode **ResetMotDePasse()** sur l'employé actuellement connecté.
2. Prévenir l'utilisateur que son mot de passe a été reseté en affichant une boîte de dialogue.



## Etape 12 :

### Enregistrement sur disque : La classe Garage se sérialise elle-même

Au démarrage de l'application, les **employés** et les **clients** devront être automatiquement lus dans un **fichier binaire** appelé « **Garage.data** ». De même, lorsque l'application se terminera, les employés et les clients devront être enregistrés dans ce même fichier.

Ce **fichier binaire** (utilisation des méthodes write et read) sera structuré de la manière suivante :

1. la variable statique **numCourant** de la classe **Intervenant**. En effet, lorsque nous ouvrirons le fichier, l'ajout futur d'un nouvel intervenant devra générer un numéro non encore utilisé.
2. Le vecteur trié des employés : pour cela, on écrira tout d'abord le **nombre d'employés**, puis ensuite **chacun des objets Employe** présents dans le conteneur employes. Lors de la lecture, on saura alors exactement combien d'objets Employe lire.
3. Le vecteur trié des clients : pour cela, on écrira tout d'abord le **nombre de clients**, puis ensuite **chacun des objets Client** présents dans le conteneur clients. Lors de la lecture, on saura alors exactement combien d'objets Client lire.

On vous demande donc d'ajouter à la classe **Garage** :

- La méthode **Save(ostream & fichier)** qui enregistrera dans le fichier toutes les données des employés et des clients selon la structure décrite ci-dessus.
- La méthode **Load(istream & fichier)** qui lira sur disque toutes les données des employés et des clients.

**Ces deux méthodes devront appeler les méthodes Save et Load de tous les objets** intervenants (Personne, Intervenant, Employe, Client). Chaque objet va donc s'enregistrer lui-même.

Au démarrage de l'application (dans le constructeur de la classe **ApplicGarageWindow**), on tentera d'ouvrir en lecture le fichier « **Garage.data** ». Si celui-ci existe, son contenu sera lu à l'aide de la méthode **Load** de la classe **Garage**. S'il n'existe pas, le fichier sera bidonné avec un seul employé « ADMIN ».

Lors de la fermeture de l'application, c'est-à-dire

- Soit en cliquant sur **l'item de menu « Quitter »** du menu « Connexion » (dans la méthode **on\_actionQuitter\_triggered**)
- Soit en cliquant sur la **croix de la fenêtre** (dans la méthode **void closeEvent(QCloseEvent \*event)**)

vous devrez ouvrir le fichier « **Garage.data** » en écriture et appeler la méthode **Save** de la classe **Garage**.



## Etape 13 :


### La gestion des contrats d'achat : la classe **Contrat**

Il s'agit à présent de gérer les contrats d'achat de voiture entre les vendeurs et les clients. Après avoir confectionné un projet de voiture avec le client, le vendeur connecté peut alors créer un contrat liant le projet, le client et le vendeur. Le contrat créé apparaît alors dans la table des contrats en bas à droite :

**Application Garage** Wagner Jean-Marc (Vendeur)

Connexion Employés Clients Voiture

Projet en cours :



Nom : **2008\_Vilvens**

Modèle : 2008 GT Line e-2008

Puissance (Ch) : 136 Prix de base : 41620,00

☐ Essence ☐ Diesel ☒ Electrique ☐ Hybride

Modèles disponibles : 208 Active 1.2 PureTech 75 (18050,00) Choisir

Options disponibles : Peinture metallisee (400,00) Ajouter

Options choisies :

Code	Prix	Intitulé
VD09	150,00	Vitres laterales arrieres surteintees
UB01	250,00	Detecteur d'obstacles arriere

Supprimer Accorder réduction

Prix avec options : 42020,00

Nouveau Ouvrir Enregistrer

Employés du garage :

Numéro	Nom	Prénom	Fonction
1	ADMIN	ADMIN	Administratif
2	Charlet	Christophe	Vendeur
7	Wagner	Jean-Marc	Vendeur

Contrats :

Numéro	Vendeur	Client	Voiture
1	Charlet ...	Leonard ...	3008_Quettier
2	Charlet ...	Leonard ...	208_Leonard
3	Wagner ...	Vilvens ...	2008_Vilvens

Clients :

Numéro	Nom	Prénom	GSM
5	Leonard	Anne	0478.56.36.32
3	Quettier	Patrick	0495.36.36.36
6	Vilvens	Claude	0475.36.21.14

Nouveau Supprimer Visualiser Voiture

On vous demande ici d'être autonome et imaginatif afin de respecter les consignes suivantes :

- Ajout d'une classe **Contrat** à votre projet. Celle-ci doit contenir un **numéro**, une **référence vers le vendeur**, une **référence vers le client** et le **nom du projet de voiture**.
- Ajouter à la classe **Garage** un **conteneur (vecteur simple)** de **contrats** → ce vecteur devra être enregistré et lu sur disque (méthode **Save** et **Load** de garage) au même titre que les conteneurs d'employés et de clients. Lors de la relecture, toutes les références doivent être rétablies correctement.

- Ajouter à la classe **Garage** les méthodes associées aux contrats comme par exemple **void ajouteContrat(...)**, etc...
- Voici les fonctionnalités des 3 boutons (en bas à droite) associés aux contrats :
  - Un clic sur le bouton « **Nouveau** » crée un nouveau contrat associant le vendeur actuellement connecté, le client sélectionné dans la table des clients et le projet de voiture apparaissant dans la partie supérieure de l'interface graphique. Le contrat apparaît alors dans la table des contrats.
  - Un clic sur le bouton « **Supprimer** » supprime le contrat sélectionné.
  - Un clic sur le bouton « **Visualiser voiture** » fait apparaître dans la partie supérieure de l'interface graphique (« Projet en cours ») le projet lié au contrat sélectionné dans la table des contrats.

Pour manipuler la table des contrats, vous disposez à nouveau des méthodes suivantes de la classe **ApplicGarageWindow** :

- void ajouteTupleTableContrats(string tuple)
- void videTableContrats()
- int getIndiceContratSelectionne()

Bon travail !