# Garage Management Application

Version: 1.0

The objective is to create a mini application allowing the management of a garage.

To return a compressed file (zip, 7z, rar) containing:

- The Visual Studio project containing all the solution and project files.

**First stage :**

Goals :

- Develop an application using advanced concepts of OOP
- Set up the notions of class and abstract method
- Set up interface concepts

We want to develop an application to manage the fleet of vehicles that a garage can offer for sale. The application will have to manage cars as well as trucks and motorcycles.

A car will be characterized by its name, price, brand, engine, options, fiscal horsepower, door number, seat number and trunk size (expressed in m3).

A truck will be characterized by its name, price, brand, engine, options, number of axles, loading weight and loading volume.

A motorcycle will be characterized by its name, price, brand, engine, options and displacement.

For each vehicle, you should be able to perform the following actions: - Display all the information for a vehicle.

- Add an option to the vehicle - Know the make of the vehicle - Know the engine of the vehicle
- Know the vehicle options
- Know the price of the vehicle (excluding options and tax)
- Calculate vehicle tax
- Know the total price of the vehicle

Each vehicle will also have an “Id” which will be automatically generated from 1. You can use a static variable for this to generate IDs with an increment of this variable each time a vehicle is added.

Vehicle tax is calculated as follows:

- For a car, we multiply the power in fiscal horses by 10 €, - For a truck, we take the number of axles and we multiply by 50 €, - For a motorcycle, we multiply the cubic capacity by 0.3€ and we take the whole part.

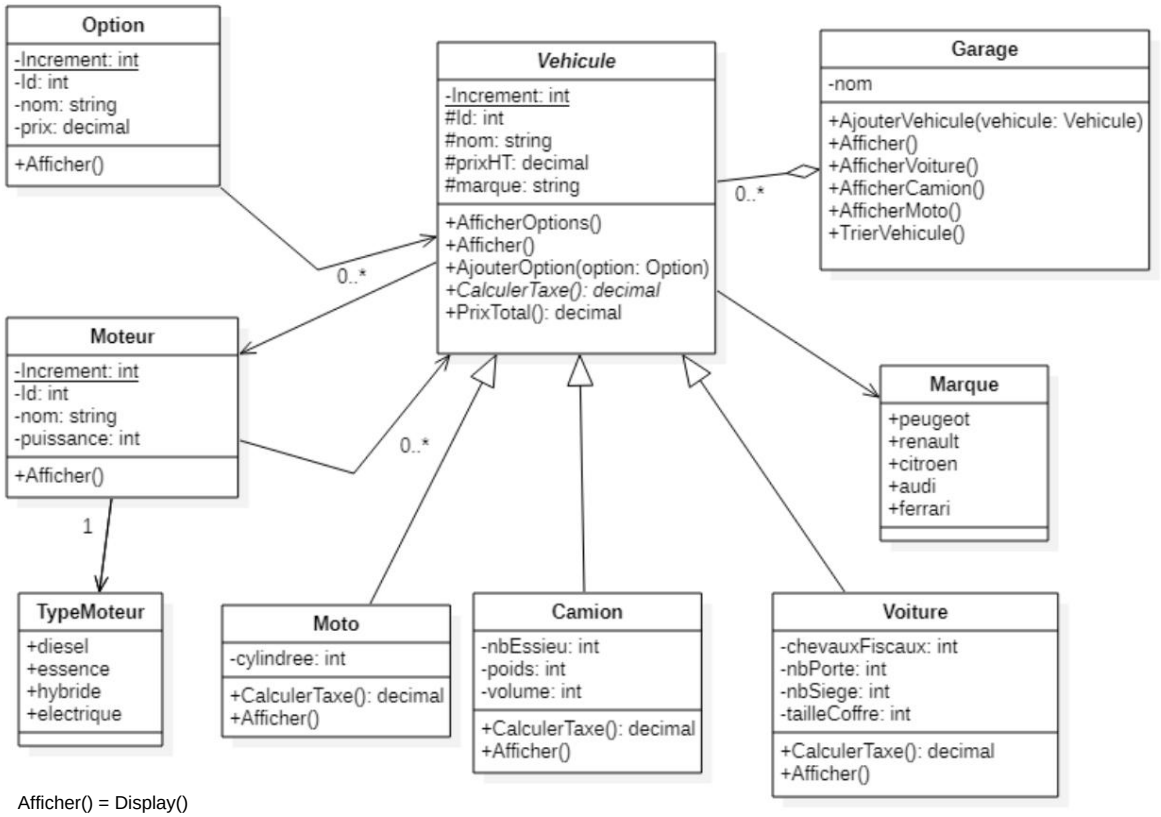
An engine will be characterized by a type (diesel, petrol, hybrid, electric) and a power. We must be able to display all the information of an engine. An engine will also have an id with the same system as for a vehicle.

An option will be characterized by a name and a price. We must be able to display all the information of an option. An option will also have an id with the same system as for a vehicle.

We must be able to add vehicles to the garages, sort the vehicles according to their price, and display all the information concerning the vehicles in the garage.

Sorting will be implemented with the implementation of the IComparable class of the .Net Framework. You will find an example implementation of this class in the “Advanced C# Object-Oriented Programming” course material.

**UML modeling of the Garage:**




Afficher() = Display()
CalculerTaxe() = CalculateTax

**Work to be done:**

1. Create a console application project, and a class library project.
2. Link the class library to the console application.
3. Develop all the classes and functionalities of the application in the library of class.
4. Create a test set of vehicles, engines, options in the console application, in order to add them to the garage and then display all the vehicles in this garage. Also test sorting and display vehicles once sorted.

**Second step :**

From the first version of the garage management application, we are going to add additional functionalities allowing to dynamically add vehicles via a menu, and to save this information in a file.

Goals :

- Use exceptions
- Create our own exceptions
- Process entered data
- Perform operations on lists
- Use the serialization mechanism to save information to a file

In order to better manage the garage and to be able to save its information in a file, we are going to create a menu allowing the following commands to be carried out:

1. View Vehicles 2. Add Vehicle
3. Delete Vehicle 4. Select Vehicle 5. View Vehicle Options
6. Add Vehicle Options 7. Remove Vehicle Options 8. View Options 9. Show Marks 10. Show Engine Types 11. Load Garage 12. Save Garage 13. Exit Application

Create a Menu class in the console application which will set up the display of the menu and all the commands.

In order to manage the garage this class will have as constructor:

```
public Menu(Garage garage) {

        this. garage = garage;
}
```

This will be the garage created in the main program.

In order to also store motors and options, you will need to add a list of motors, a list of options and the AddMotor and AddOption methods to your garage.

A Start() method will launch the menu from the program. This method will display the menu, and will loop until the user has selected choice 13 to exit the application.

To deal with the exceptions of the application, we will put a global try … catch block around the menu which will allow to catch the Exceptions thrown by the commands of the menu and to display an error message to the user.

In this Start() method, a switch case will test the values entered by the user. This value being recovered in the form of string it will be necessary to convert this value into int in order to test it in the switch case.

In the Menu class create a GetChoice() method which will allow:

- Retrieve the choice entered by the user
- Perform the conversion to int
- Return result converted to int

This method will have to implement a first exception handling to handle the case where the user does not enter a number. The exception thrown during the conversion is the FormatException. In this method, you will have to catch the exception then raise the FormatException again with the error message: "The choice entered is not a number".

The message will be displayed by the try … catch block of the Start method.

It will also display a message if the user enters a choice that does not exist in the menu.

For this we will create an Exception specific to our application.
Create this Exception in the Menu.cs file and name the class MenuException. You will find how to create a new custom exception in the file "C-Sharp Object-Oriented Programming Exceptions.pdf". Use the exception constructor to specify a message which could be "The choice is not between 1 and 13".

Create a GetChoiceMenu() method which will use the GetChoice() method to retrieve the entered number and which will check if the number is between 1 and 13. If this is not the case, the MenuException exception must be raised.

The message will be displayed by the try … catch block of the Start method.

For each operation of the menu create a method, which will be responsible for performing all the necessary operations of the command.

For example for the command "1. Display vehicles", we will create a method:

```
public void ShowVehicles() {


}
```

Which will be called in the switch case:

box 1:

```
ShowVehicles(); break;
```

Create all the methods corresponding to each function provided in the menu.

Code all the functionalities by trying to put in the class Garage the basic functions and in the class Menu the interface with the user.

Also add new exceptions to handle, for example, cases where the user selects a vehicle that does not exist, a brand that does not exist, an engine that does not exist, etc.

**Last step :**

Modify the different classes Vehicle, Car, Truck, … in order to add the serialization mechanism.

Just add the attribute:

[Serializable]

To each class.

Add, inspired by the support “The serialization of objects.pdf” in your Garage class, the two methods:

Save: Allows you to save an object.
Load: Allows you to load an object.

Add in your menu management and in your program the function of saving and loading a garage.