# **Documentation and Class Diagrams**

Assigned by:

|  |  |
| --- | --- |
| **Full Name** | **ID** |
| \_Haim Adrian\_ | \_305265514\_ |
| \_Yakir Saadia\_ | \_204357669\_ |

# **GarageLogic**

## **API**

**IVehicle**

Interface that exposes general vehicle properties, such as brand, energy left, license number and collection of tires.

Today, this is the only interface of vehicle type that we expose outside the Logic layer. Though, to make it more convenient for users of the Logic layer, they can use the generic create method of GarageController, and receive specific interface and not only IVehicle. This is what we did in the UnitTests in order to ease our tests with setting specific vehicle properties, such as car color.

Please note that the UnitTests project is the only project that uses the interfaces. In the ConsoleUI we are creating a vehicle, referring it as IVehicle, and then **we fill in specific properties in reflection** – **to make sure we change nothing in case adding a new vehicle type**. The only place to modify is VehicleType, where we list the types in order to use them for creating new instances in reflection.

**IVehicleVisitor**

Interface that supports visiting inheritors of IVehicle, according to the Visitor Design Pattern, which helps us creating a text report (all details) about vehicles, for console today and for UI in the future, leaving the ToString of vehicles shorter for debug / logging purposes.

We can add more visitors in the future and respond to vehicle types easily, without changing the implementation of vehicles at all.

**IElectricVehicle**

Interface that exposes properties and functionality which are common in electric vehicles, such as battery data and charging methods

**IFuelVehicle**

Interface that exposes properties and functionality which are common in fuel vehicles, such as fuel data and refuel/drain methods

**IMotorcycle**

Interface that exposes properties which are relevant to motorcycles, such as engine capacity and license type

**ICar**

Interface that exposes properties which are relevant to cars, such as color and number of doors

**ITruck**

Interface that exposes properties which are relevant to trucks, such as cargo volume and indication of whether a truck having dangerous substances or not

**IElectricMotorcycle**

An electric motorcycle (extends IElectricVehicle and IMotorcycle)

**IFuelMotorcycle**

A fuel motorcycle (extends IFuelVehicle and IMotorcycle)

**IElectricCar**

An electric car (extends IElectricVehicle and ICar)

**IFuelCar**

A fuel car (extends IFuelVehicle and ICar)

**IElectricTruck**

An electric truck (extends IElectricVehicle and ITruck)

**IFuelTruck**

A fuel truck (extends IFuelVehicle and ITruck)

**IGarage**

Interface to expose functionality of a garage, such as adding a vehicle, collecting all license numbers, getting vehicle report, etc.

**IGarageVehicle**

An interface used to decorate an IVehicle and add owner details (ICustomer: name and phone number), as well as a car status (in garage)

**Tire**

A vehicle’s tire contains properties and methods related to air pressure data and manufacturer name.

This is a class and NOT a struct since it is being held by a collection, as described at IVehicle, and we would like to support updating a tire without being obligated to treat it as an immutable object. A tire is not immutable since we can inflate it and update its air pressure property

**eColor**

Enum that lists available car colors, such as Yellow, White, etc.

**eLicenseType**

Enum that lists motorcycle license type, such as A, A1, etc.

**eFuelType**

Enum that lists fuel types, such as Octan95, Soler, etc.

**eDoorsAmount**

Enum that lists supported number of doors in a car. e.g., Two, Three, etc.

**eVehicleState**

A state of car in the garage. e.g., Repairing, Repaired, etc.

### **Exceptions**

**ValueOutOfRangeException**

Extends Exception with two additional float properties describing minimum and maximum values of some range.

It is being thrown when air pressure of tires is above maximum air pressure, or below minimum air pressure.

It is also being thrown in case battery charging or refueling exceeds maximum capacity.

**VehicleException**

Base class for exceptions related to vehicle, which extends Exception with license number of a vehicle

**VehicleAlreadyExistsException**

Extends VehicleException and being thrown when client adds an already existing vehicle to the garage, thus we can notify the client that a vehicle is already existing

**NoSuchVehicleException**

Extends VehicleException and being thrown when client tries to access or perform any operation on a license number that does not exist in the garage

**AVehicleWrongKindException**

Abstract class that extends VehicleException, created to hold strings that describe expectation and actual types. We use this exception to break illegal operations on vehicle types that do not support them. For example, trying to refuel an electric vehicle

**WrongVehicleTypeException**

Extends AVehicleWrongKindException and being thrown when client tries to perform operations on a car that does not support them. For example, charging a fuel vehicle, or refueling an electric vehicle

**WrongFuelTypeException**

Extends AVehicleWrongKindException and being thrown when client tries to refuel a fuel vehicle with a wrong fuel type. For example, when a vehicle’s fuel type is Octan95 and client tries to refuel this vehicle with Soler

### **Controllers**

Controllers were created to make it clear for the client through which classes it can create the models, without being aware to the concrete models. Thus, we have one location in which we code the construction of the models, and one interface only (IVehicle) that we must keep its structure.

**VehicleController**

Singleton, which is used to create various vehicle types, such as electric motorcycle, fuel motorcycle, electric car, fuel car, etc.

VehicleController uses VehicleFactory, where we create vehicle models using Reflection – which means no changes in the creation in the future. We can implement a new vehicle type, and just add it to VehicleType struct, to connect between the new class and its type, so we will be able to create it using Reflection

**GarageController**

Singleton, which holds a reference to our garage, and expose the functionality requested by this exercise. For example, inserting a car to the garage, or fully inflating tires of a vehicle

**VehicleFactory**

A static class used to construct various vehicle types, generically, using Reflection

**VehicleType**

A struct we use to list all vehicle types and map them to their C# Type, so VehicleFactory will be able to create those types using Reflection.  
In our implementation, this is the only location we need to modify once we add a new vehicle type, and everything will work the same

## **Implementation**

**AVehicle<TEngineType>**

Abstract class that implements IVehicle to hold data members which are common to all vehicles.

This class holds the engine as generic type, because we have an Engine for electric models, and FuelEngine for fuel models

**Engine**

This class contains energy data (current and maximum) for electric and fuel vehicles.

We use this class to avoid of duplicating the code responsible for range validation when updating the amount of energy (fuel/battery)

**FuelEngine**

This class extends Engine to add fuel type details

**AElectricVehicle**

Abstract class that extends AVehicle<Engine> and implements IElectricVehicle, to expose data and functionality related to electric vehicles, through an instance of Engine, thus reducing code duplicates

**AFuelVehicle**

Abstract class that extends AVehicle<FuelEngine> and implements IFuelVehicle, to expose data and functionality related to fuel vehicles, through an instance of FuelEngine

**MotorcycleSpecific**

A class that holds engine capacity and license type for motorcycle, so we can compose it in electric motorcycle and fuel motorcycle, without duplicating motorcycle functionality and properties, and being able to extend AElectricVehicle and AFuelVehicle accordingly

**CarSpecific**

A class that holds color and number of doors for car, so we can compose it in electric car and fuel car, without duplicating car functionality and properties, and being able to extend AElectricVehicle and AFuelVehicle accordingly

**TruckSpecific**

A class that holds cargo volume and flag if having dangerous substances for truck, so we can compose it in electric truck and fuel truck, without duplicating truck functionality and properties, and being able to extend AElectricVehicle and AFuelVehicle accordingly

**ElectricMotorcycle**

Extends AElectricVehicle and compose MotorcycleSpecific, to gain all functionality relevant to an electric vehicle, and expose motorcycle specific functionality as well

**FuelMotorcycle**

Extends AFuelVehicle and compose MotorcycleSpecific, to gain all functionality relevant to a fuel vehicle, and expose motorcycle specific functionality as well

**ElectricCar**

Extends AElectricVehicle and compose CarSpecific, to gain all functionality relevant to an electric vehicle, and expose car specific functionality as well

**FuelCar**

Extends AFuelVehicle and compose CarSpecific, to gain all functionality relevant to a fuel vehicle, and expose car specific functionality as well

**FuelTruck**

Extends AFuelVehicle and compose TruckSpecific, to gain all functionality relevant to a fuel vehicle, and expose truck specific functionality as well

**GarageImpl**

A concrete implementation of IGarage. See IGarage

**GarageVehicle**

A concrete implementation of IGarageVehicle. See IGarageVehicle

**VehicleToReportVisitor**

An implementation of IVehicleVisitor which visits each vehicle type in order to create a string report for a vehicle, ready to be printed

# **ConsoleUI**

**MenuItem<T>**

A menu item has a generic item type, text to display (in console/ui), and a delegate (event handler) to be notified when the menu item is selected.

**MenuItemGroup<T>**

Implements ICollection<MenuItem<T>> in order to group several menu items.

We also maintain a Dictionary inside this class, to make sure each menu item is added once, and to support indexOf and Get operations to take O(1) instead of O(n) of searching in a list. (We map between a menu item to its index in the list of menu items)

The list is needed so we will keep the order to adding the menu items.

**ConsoleMenuManager<TMenuItem>**

This class created so we can print a menu item group to console and ask user to select a menu item.

This way we support multiple menu item groups so we can display them in “different” screens.

**AMenu<T>**

Base class (abstract) for a menu in the UI layer.

A menu consists of menu item group and uses ConsoleMenuManager to print the menu item group.

Each “page” of our application derives from AMenu and initializes its own menu item group.

**AddVehicleMenu**

Maybe the most complex class in the exercise, as it uses ReflectionUtils in order to support adding new vehicles to the system, in such a way that we will not have to modify this class in case adding new vehicle type.

We achieve this by scanning, in reflection, all the vehicle types from VehicleType struct. Then we show them in our menu so user can select what vehicle type he’s like to enter. Once there will be a new vehicle type added, we will show it as well because we look for ALL the vehicle types in reflection.

Then, once we have created a vehicle, we scan its type in reflection to detect public instance properties that are writeable. For each property that we detect, we ask the user to enter a value (with validation based on the property type) and set it to the vehicle. You may find this logic in **ReflectionUtils** class.

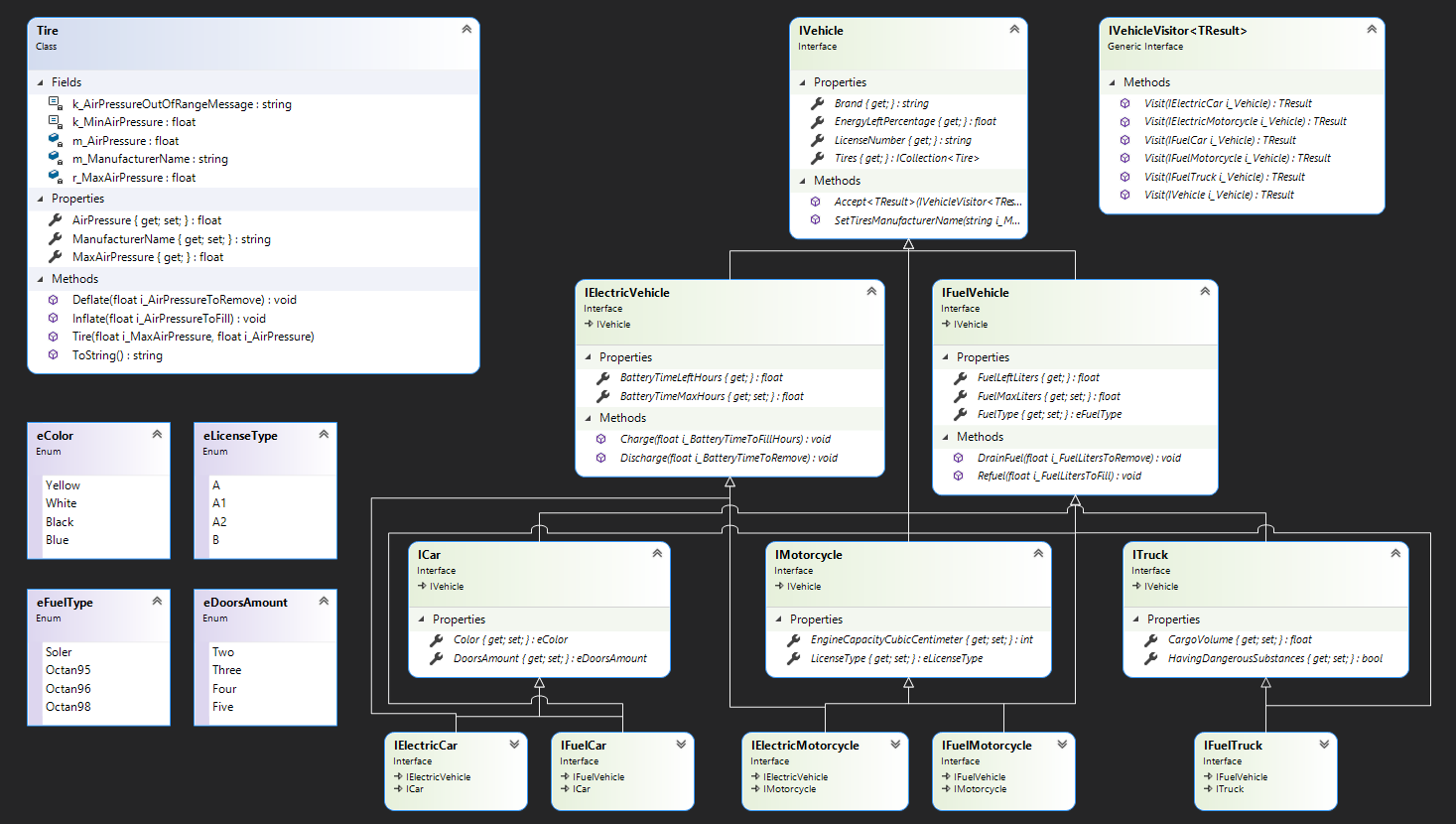
**MainMenu**

The main menu of the application is displayed until user selects to exit.

There are additional menus, but they are too simple to understand from the class diagram below.

Continue to next pages for class diagrams…

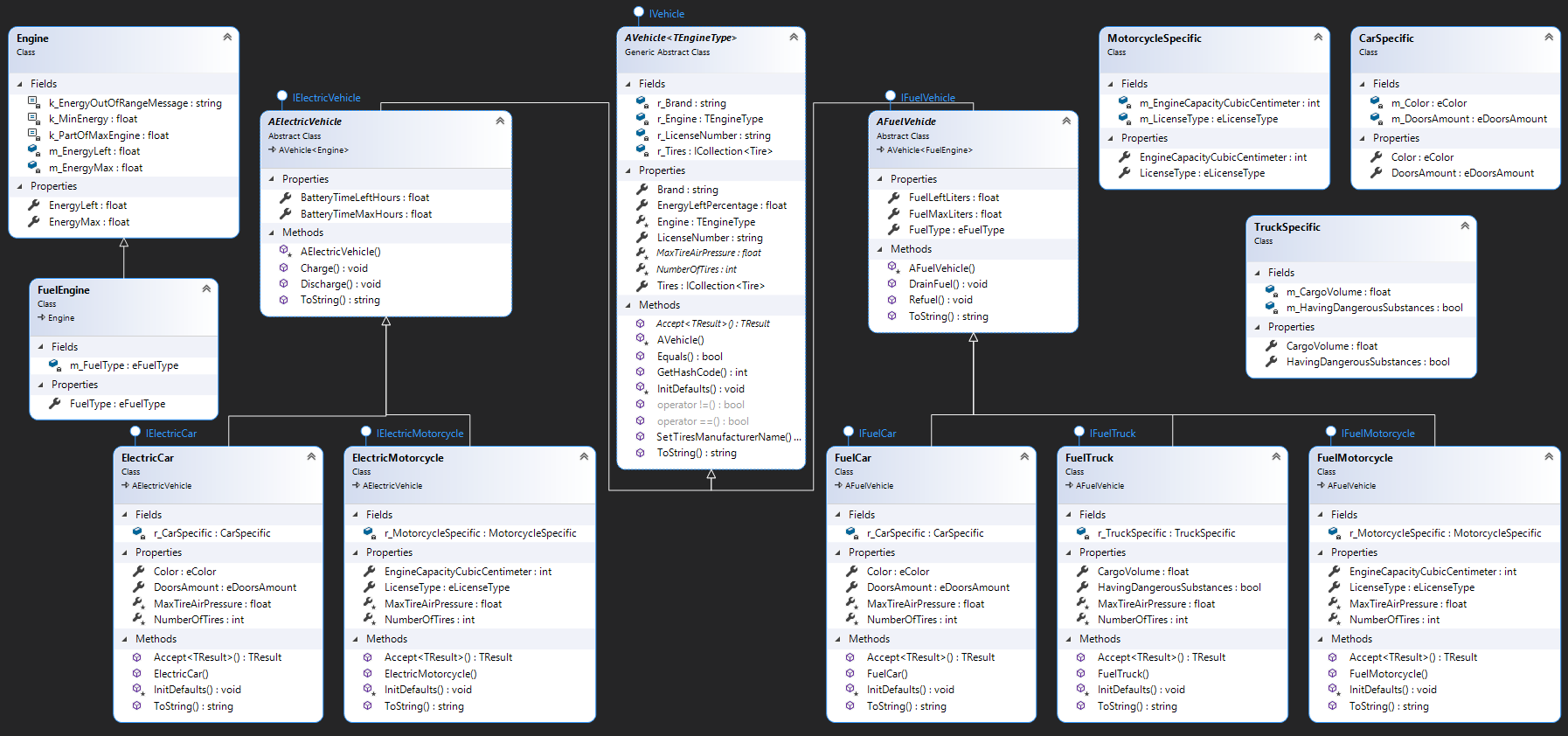
# **Interfaces created in order to expose functionality outside the Logic layer, and support “multiple” inheritance.**



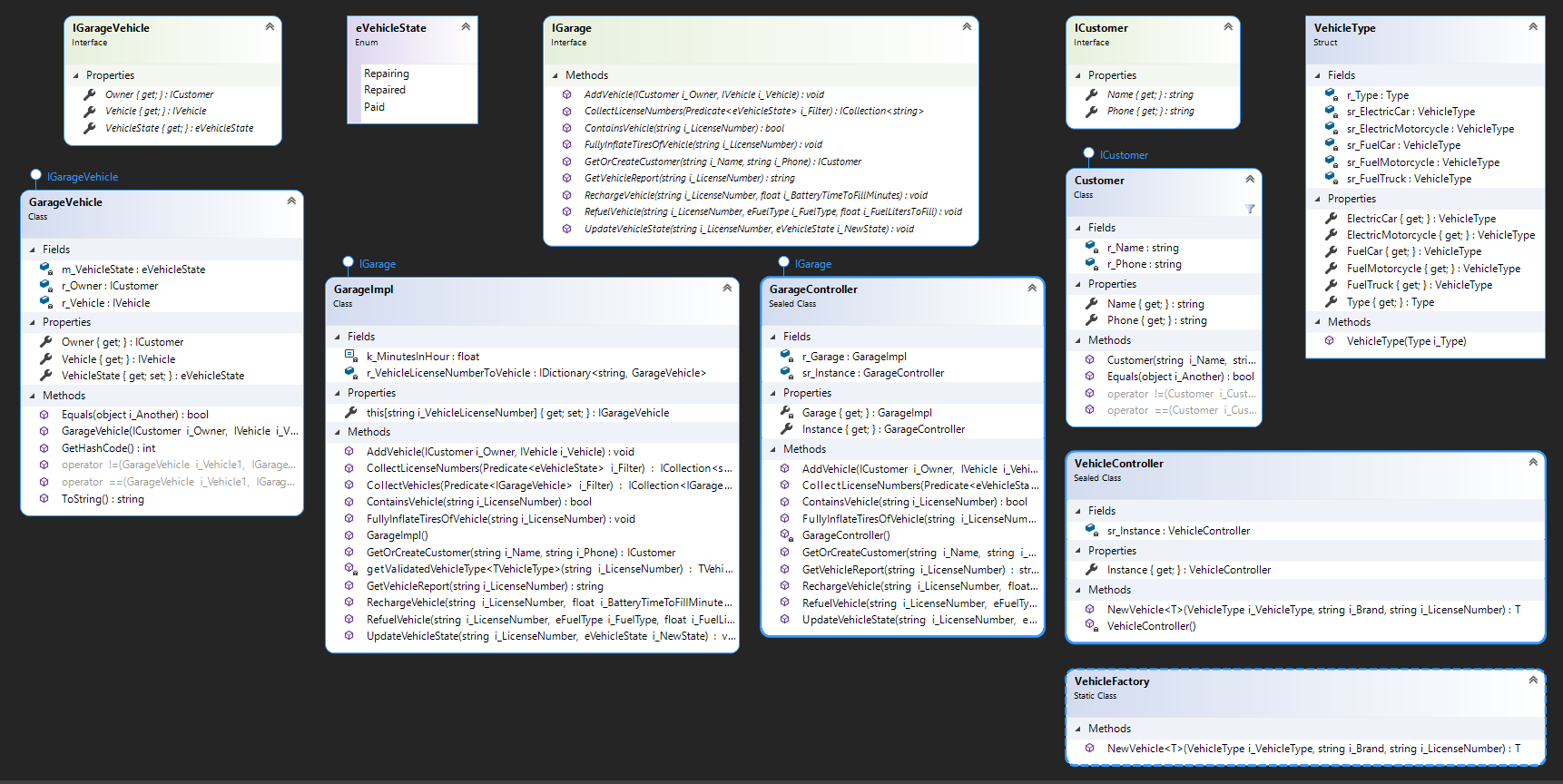
# **Implementation of the interfaces (internal to Logic layer)**

Please note the “XXXSpecific” is a part that we compose in order to share common stuff between Electric and Fuel implementations

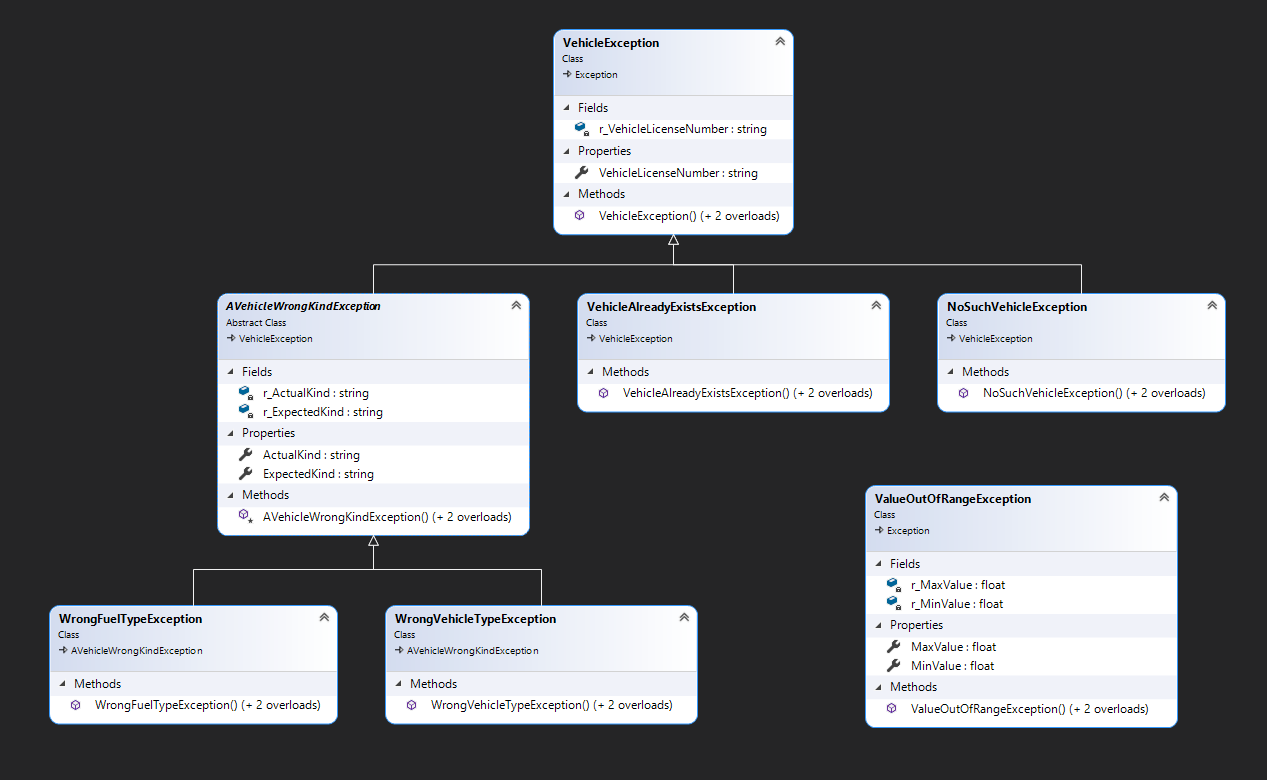
So is the Engine, which contains energy details that we compose inside AVehicle.



# **Garage and Controllers (public API to let other layers to use the Logic layer)**



# **Exceptions**



# **UI**

