TECHNICAL REPORT

User Stories

As a user, I want to register a account to that can access to the app.

As a user, I want to login to that use the specific functionalities according to my role.

As a user, I want to see all vehicles to that explore the available options.

As a user, I want to search vehicles for specific criteria to that find vehicles that fit my needs.

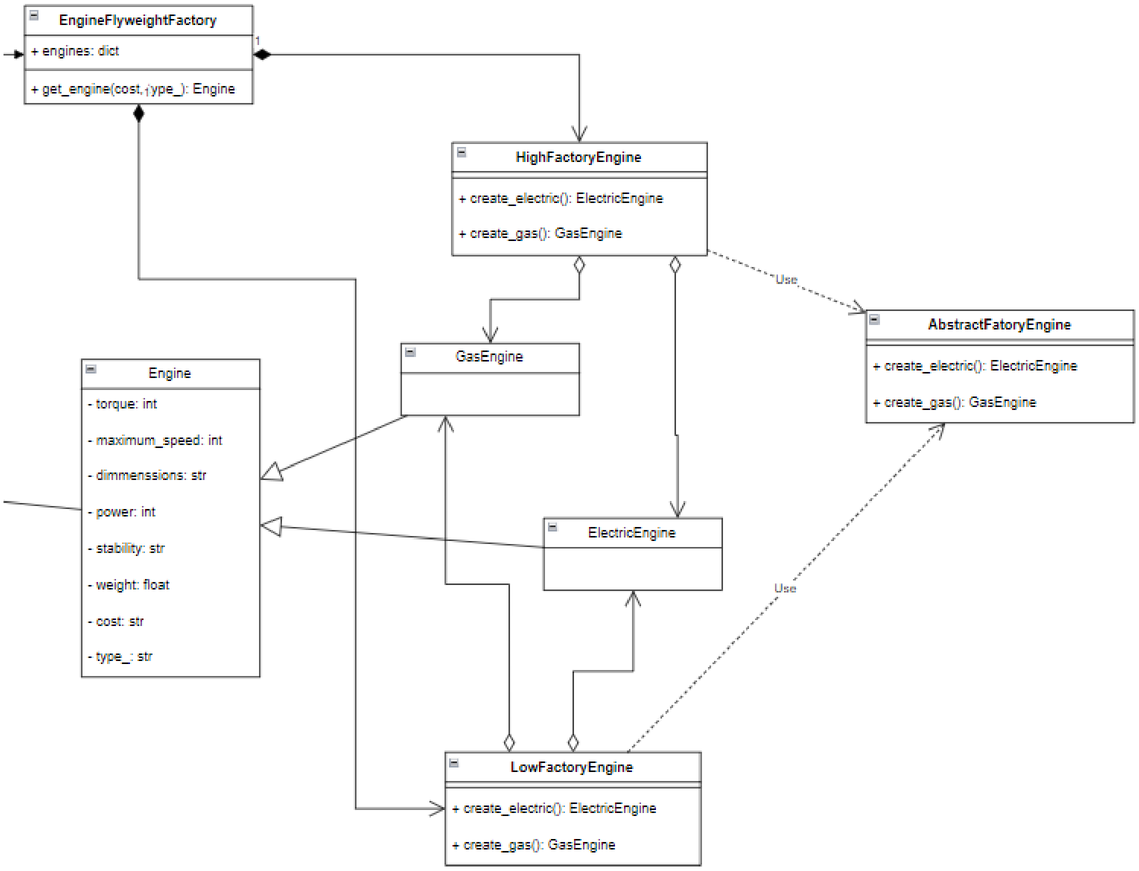
As a admin, I want to add new vehicles to that keep the catalog updated.

As a admin, i want to update the information of a vehicle to that fix errors or update details.

As a admin, i want to delete a vehicle to the catalog to that keep it updated.

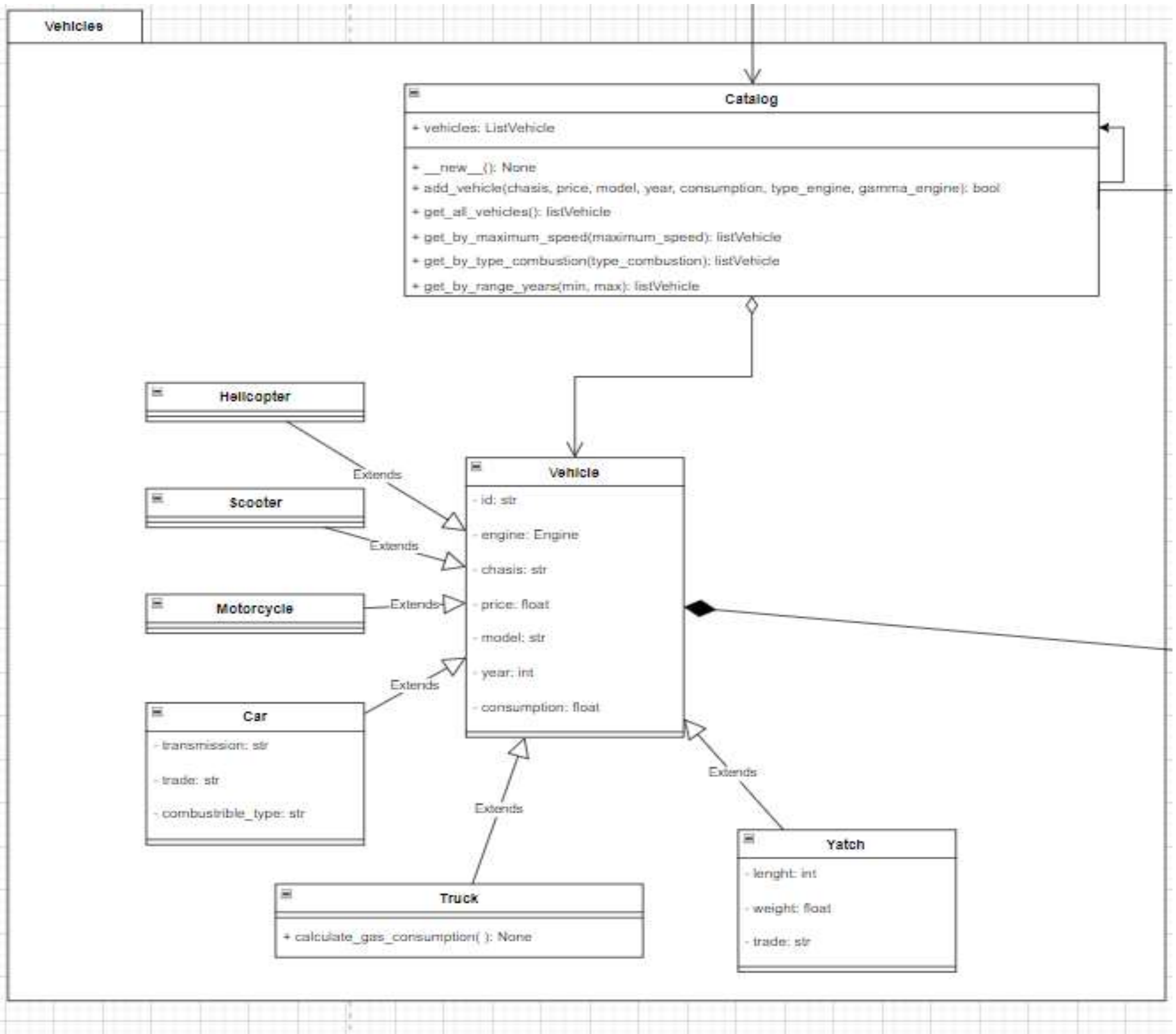
I divided the system into 7 components

1. Engines

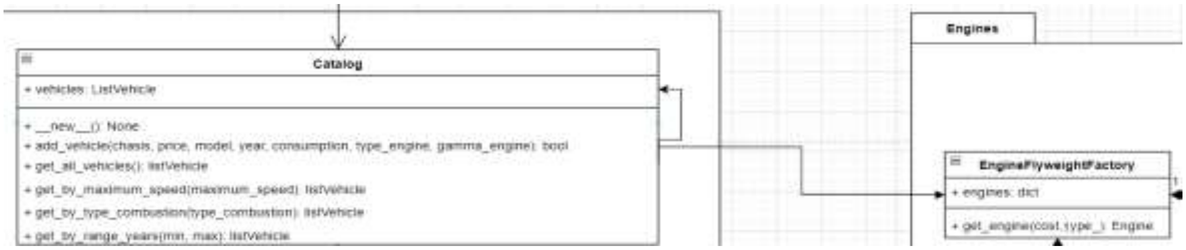


It is based on the model from the previous workshop, we continue managing the factories, an EngineFlyweightFactory class was created which will create the engines using the Flyweaight pattern to save memory, the engine object was also encapsulated to restrict access to the information and make it not modifiable.

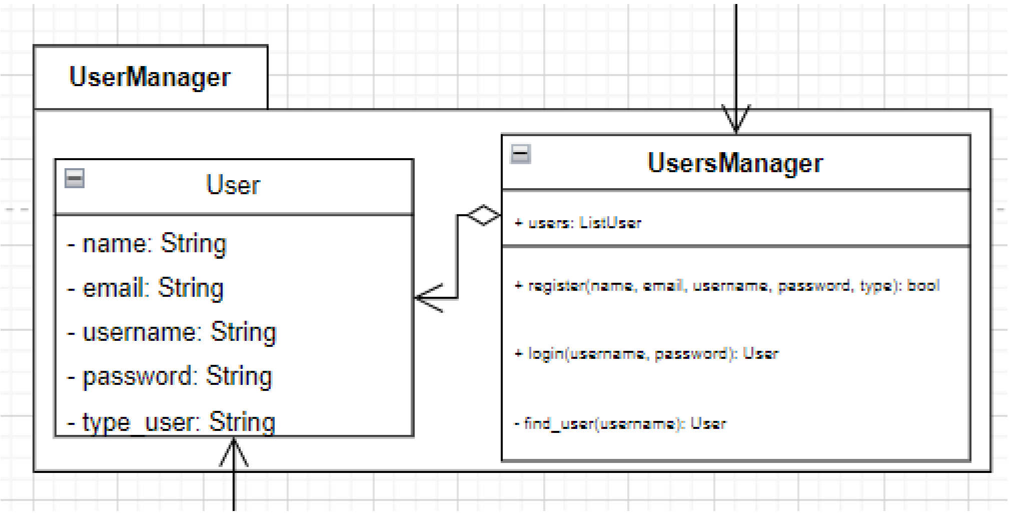
2. Vehicles



It is based on the model of the previous workshop, the only modification that was made is that methods were added to the query catalog and the ability to update and delete a vehicle is no longer related to the engine class, now it is related to the EngineFlyweightFactory class which creates the engines

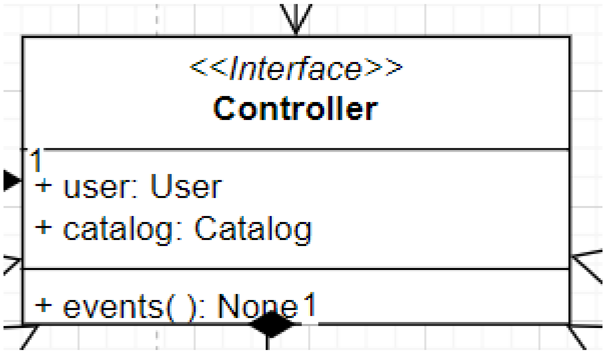

Catalog
+ vehicles: ListVehicle
+ __new__(): None
+ add_vehicle(chasis, price, model, year, consumption, type_engine, gamma_engine): bool
+ get_all_vehicles(): listVehicle
+ get_by_maximum_speed(maximum_speed): listVehicle
+ get_by_type_combustion(type_combustion): listVehicle
+ get_by_range_years(min, max): listVehicle
Engines
EngineFlyweightFactory
+ engines: dict
+ get_engine(cost,type_): Engine
1

3. UserManager



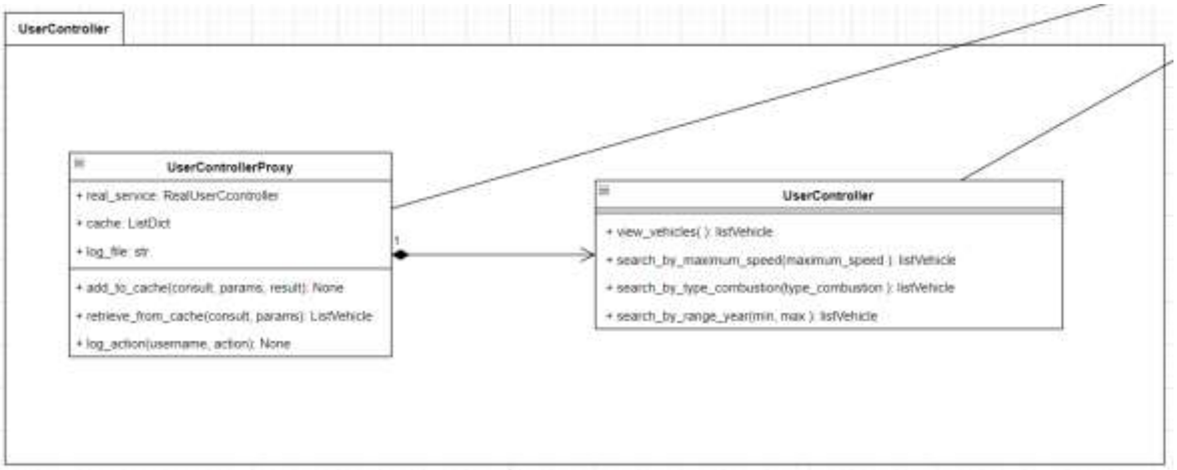
Create the user class that represents the attributes of a user, on the other hand there is the usersManager class which contains the list of registered users, it also contains the methods for registration and login validation of users

4. Controller:



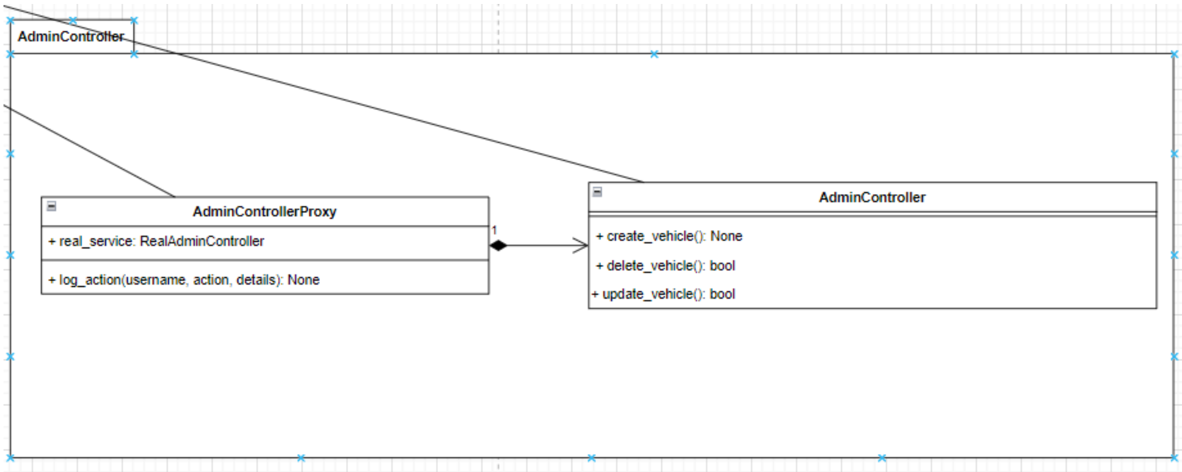
I created a controller interface to manage the actions of users and administrators

5. UserController:


UserController
UserControllerProxy
+ real_service: RealUserCcontroller
+ cache: ListDict
+ log_file: str
+ add_to_cache(consult, params, result): None
+ retrieve_from_cache(consult, params): ListVehicle
+ log_action(username, action): None
1
UserController
+ view_vehicles( ): listVehicle
+ search_by_maximum_speed(maximum_speed): listVehicle
+ search_by_type_combustion(type_combustion): listVehicle
+ search_by_range_year(min, max): listVehicle

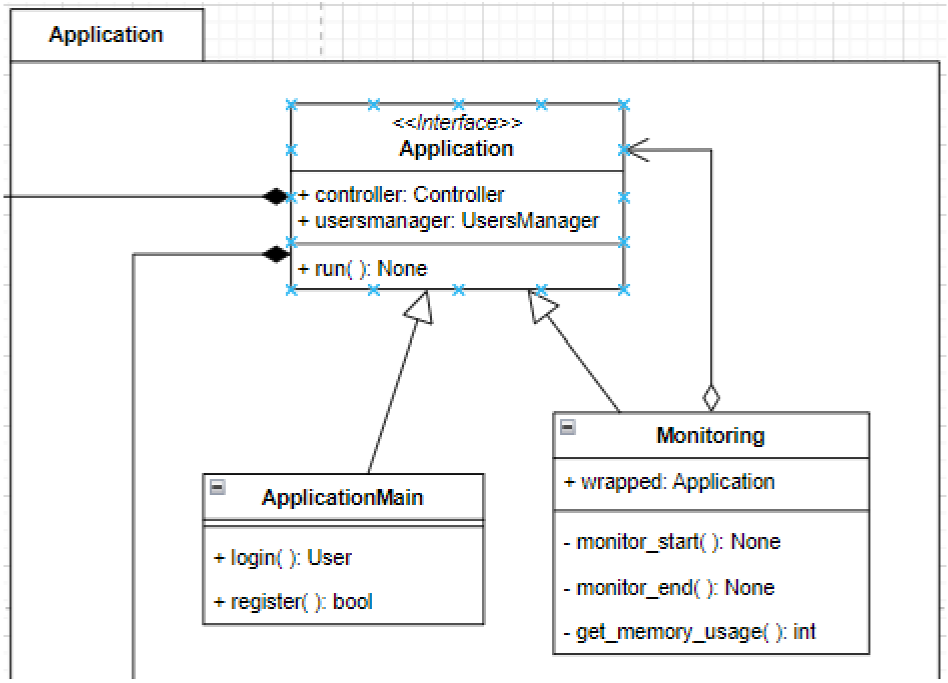
For this component use the proxy pattern, the UserController and UserControllerProxy class extend the Controller interface, this component is responsible for handling the actions and events that the user makes in the application, the proxy will save each query that the user makes in the log file , it will also cache the results of the last five queries made to speed up future searches.

6. AdminController:



For this component use the proxy pattern, the AdminController class and AdminControllerProxy extend the Controller interface, this component is responsible for handling the actions and events that the admin does in the application, the proxy will save each action that the admin does in the log file , how to create, delete or update a vehicle.

7. Application



The application module will be in charge of running the app, it will use the usersManager to request the registration or login of a user who wants to use the services, it will also have a controller to which UserControllerProxy or AdminControllerProxy is assigned depending on the type of user who logs in, the decorator pattern was also used to monitor the application by measuring the execution time and memory consumption, for this we created the Monitoring class