TECHNICAL REPORT

New requirements:

1. Add a user subscription to a newsletter using their email:

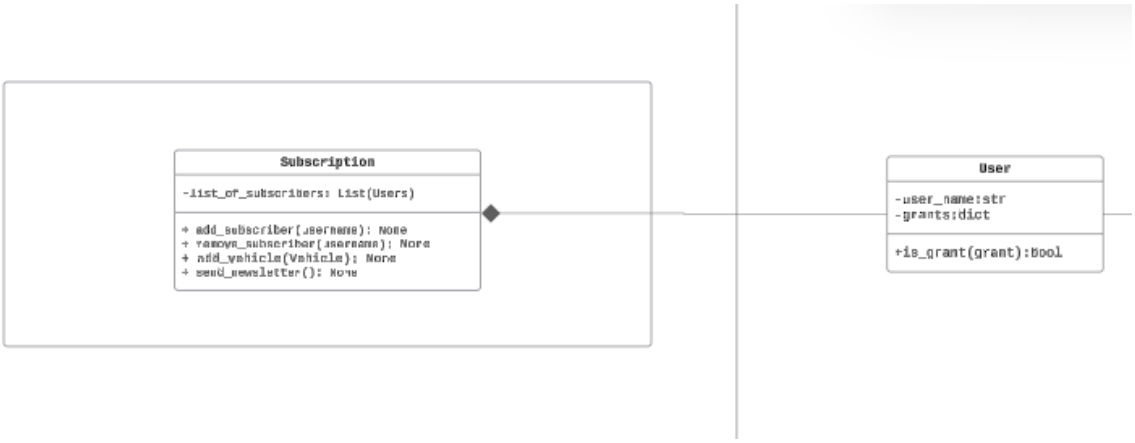
To meet this requirement, it has been decided to create or add a new class called Subscription to manage subscribers and newsletter shipments. This class will have the following attributes and methods:

Attributes:

- subscribers: A list of users without permissions or (customers) who are subscribed.
- newletter_vehicles: It is a list of the latest vehicles created.

Methods:

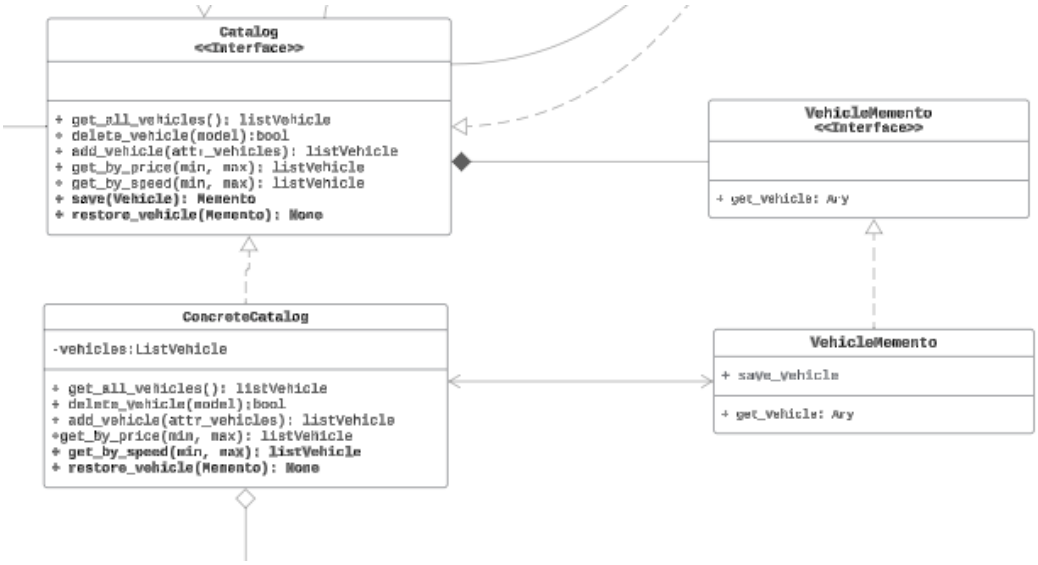
- add_subscriber(username): Adds a subscriber to the list. Receives the username as a parameter.
- remove_subscriber(username): Removes a subscriber from the list. Receives the username as a parameter.
- add_vehicle(Vehicle): Adds a newly created vehicle to the list of newletter_vehicles, receives a vehicle object.
- send_newsletter(): Send a newsletter to all subscribers. Determines the number of vehicles in newletter_vehicles, if it is less than 5, send the catalog list, and if it is greater, only send the newletter_vehicles list.



2. Implement an option to recover the last vehicle removed.

To fulfill this second requirement I have decided to implement the Memento behavior pattern, since this pattern allows us to create a copy of the vehicle before deleting it, storing it in one of its attributes. And in this way, allowing us, if we wish, to recover the vehicle later.

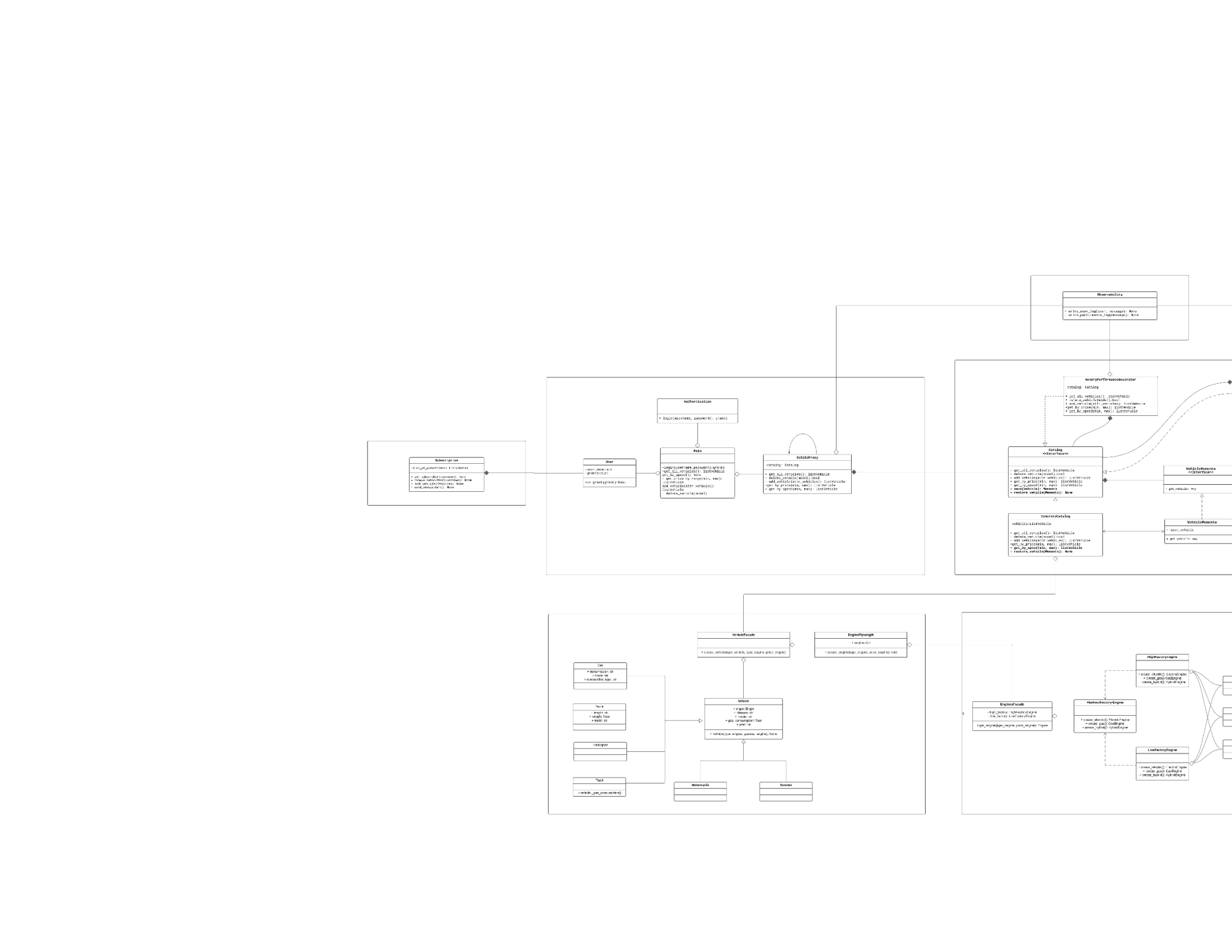
Concern: I was not very clear in which class the Memento should be implemented, because it would be used to store the last deleted vehicle, so I first thought about the vehicle class, but it as such does not have the state or instance of the last one. It was deleted since this is to create them. So, I decided to do it for the catalog that despite having a whole list of vehicles, this is the one in charge of eliminating them, this being the action that before the vehicle occurs, it must save a copy of the vehicle that will be eliminated, that is, that this is the originating class. Additionally, when it is restored, it must be added to the vehicle list again, which must be done from the catalog.



3. Add a new type of engine, the hybrid engine, which should have the attribute: electric_power

To meet this requirement, you must add a new class called hybrid_engine, in the AbstracFactoryEngies, making it also inherit from the Engine class, which is the generic engine and adding its own attribute, then add the method to create it in the two factories , low and high cost, just like in the AbstractFactory:

Diagram UML


Observability
Subscription
User
Authentication
Main
CatalogProxy
MemoryPerformanceDecorator
Catalog
VehicleMemento
ConcreteCatalog
VehicleFacade
EngineFlyweigh
Car
Truck
Helicopter
Truck
Vehicle
Motorcycle
Scooter
EnginesFacade
AbstractFactoryEngine
HighFactoryEngine
LowFactoryEngine

For the development of this workshop, after developing the UML diagram with the new requirements, we proceeded to implement it in the code. Additionally, all documentation was completed and improved, and some unit tests for the subsystems were added. During this review, the following poor practice was identified:

IF nested: Because these are problematic in the code due to their negative impact on readability, maintainability and debugging ability. As the nesting depth increases, the code becomes more difficult to understand and modify, further violating the single responsibility principle. To improve this, it is advisable to refactor nested code blocks into separate functions or use alternative control structures, such as loops or dictionaries, to keep the code cleaner and easier to maintain in the long term.

```python
class EnginesFacade:
    def get_engine(type_engine: str, price_engine: str) -> Engine:

        Args:
            type_engine (str): The type of the engine.
            price_engine (str): The price of the engine.

        Returns:
            An engine based on parameters.
        """
        if price_engine == "high":
            if type_engine == "electric":
                engine = EnginesFacade.high_factory.create_electric_engine()
            elif type_engine == "gas":
                engine = EnginesFacade.high_factory.create_gas_engine()
            elif type_engine == "hybrid":
                engine = EnginesFacade.high_factory.create_hybrid_engine()
            else:
                raise ValueError("Invalid engine type")
        elif price_engine == "low":
            if type_engine == "electric":
                engine = EnginesFacade.low_factory.create_electric_engine()
            elif type_engine == "gas":
                engine = EnginesFacade.low_factory.create_gas_engine()
            elif type_engine == "hybrid":
                engine = EnginesFacade.low_factory.create_hybrid_engine()
            else:
                raise ValueError("Invalid engine type")
        else:
            raise ValueError("Invalid price value")

        return engine
```

Solution: To not use nested ifs, an alternative was looked for where dictionaries were used

```python
class EnginesFacade:
    def get_engine(type_engine: str, price_engine: str) -> Engine:
        """
            type_engine (str): The type of the engine.
            price_engine (str): The price of the engine.

        Returns:
            An engine based on parameters.
        """

        factories = {
            "high": EnginesFacade.high_factory,
            "low": EnginesFacade.low_factory,
        }

        engine_creators = {
            "electric": lambda factory: factory.create_electric_engine(),
            "gas": lambda factory: factory.create_gas_engine(),
            "hybrid": lambda factory: factory.create_hybrid_engine(),
        }

        if price_engine not in factories:
            raise ValueError("Invalid price value")

        if type_engine not in engine_creators:
            raise ValueError("Invalid engine type")

        factory = factories[price_engine]
        engine = engine_creators[type_engine](factory)

        return engine
```