Does your class diagram respect or violate SOLID principles?

Our project indeed respects the SOLID principles as it’s shown:

# **Single responsibility principle**

In our class diagram, we implemented each class to fulfill the SRP.

We divided our controller class so that it could handle each responsibility individually by dividing the work done by the system. as seen in all of the classes.

# **Open/closed principle**

We handled this principle by enabling the classes to be open for extensions but closed for modification. By making implementing interfaces that ease this process. Any modification can be done to our existing system without complicating the understanding and implementation of our system. as seen in the entity classes and controller classes.

**Interface segregation principle**

In programming, the interface segregation principle states that no client should be forced to depend on methods it does not use. We did not add additional functionality to our system.

Instead, We created a new interface and let our class implement multiple interfaces if needed. So each class has its own needed interface without being forced to implement non-needed functions. As seen in all of the classes.

Does your class diagram contain any design pattern(s), if yes name it and list the names of the classes involved in such pattern(s)?

We applied the following design patterns in our code :

the strategy pattern is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use. This design can be reflected clearly in the “Slotassigner” class.

the adapter pattern is a software design pattern that allows the interface of an existing class to be used as another interface. It is often used to make existing classes work with others without modifying their source code. This design can be reflected clearly in the “GarageController” class.