LAB 1 - SOLID
Done by : Mohamed Karam Hassoun & Firas Saada

Exercice 1 :

The SRP states that a class should have only one reason to change, meaning it should have only one job or responsibility.

Original design issues :

In the original CarManager class, multiple responsibilities were bundled together:

- Managing a database of cars
- Formatting car information for display (getCarsNames)
- Selecting the "best" car based on some criteria (getBestCar)

This design violates the SRP because changes to how cars are formatted, how the best car is selected, or how cars are stored and retrieved would all require modifications to the CarManager class.

Modified design explanation :

- CarDatabase.java

Responsibility: Manages the storage and retrieval of Car objects.

Reason for Change: Changes are only expected if the way cars are stored or retrieved from the database changes.

This class is focused solely on the database aspect of the application, abstracting the storage mechanism away from other concerns.

- CarFormatter.java

Responsibility: Formats car information into a human-readable string.

Reason for Change: Changes are only expected if the presentation of the car data needs to be altered.

Separating the formatting logic into its own class allows changes to how cars are displayed without affecting database operations or the criteria for selecting the best car.

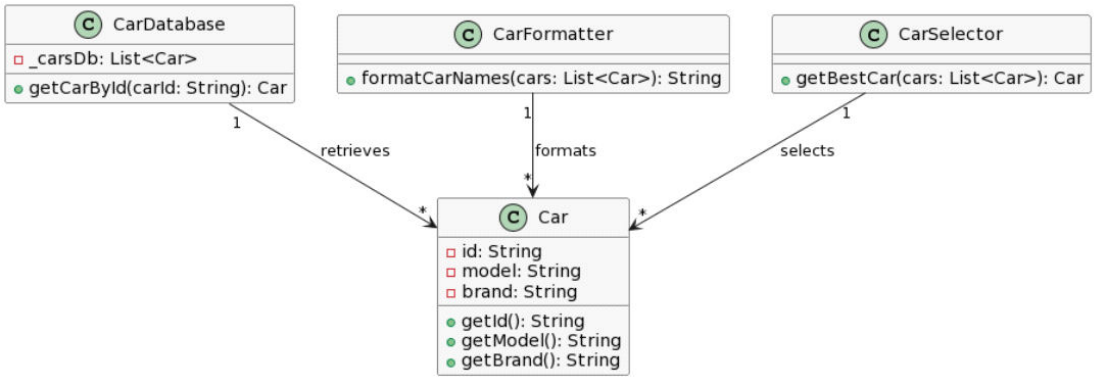
- CarSelector.java

Responsibility: Implements the logic to select the "best" car from a list of cars.

Reason for Change: Changes are only expected if the criteria for selecting the best car change.

This class encapsulates the decision-making process, allowing the criteria for selecting the best car to be modified independently of how cars are stored, retrieved, or formatted.

UML Diagram :



Exercice 2 :

The OCP states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

Original design issues :

In the original ResourceAllocator class, the addition of a new type of resource (beyond TIME_SLOT and SPACE_SLOT) would require modifying the allocate and free methods to include new cases in their switch statements.

This design violates the OCP because it necessitates changes to existing code for any extension (adding new resource types), increasing the risk of introducing bugs to existing functionalities.

Modified design explanation :

- Resource Interface

Role: Defines the contract for resource operations (allocate and free) that all resource types must adhere to.

Extension: New resource types can implement this interface to become part of the system without altering existing code.

The introduction of the Resource interface allows for the creation of new resource types by merely adding new classes that implement this interface, adhering to the OCP.

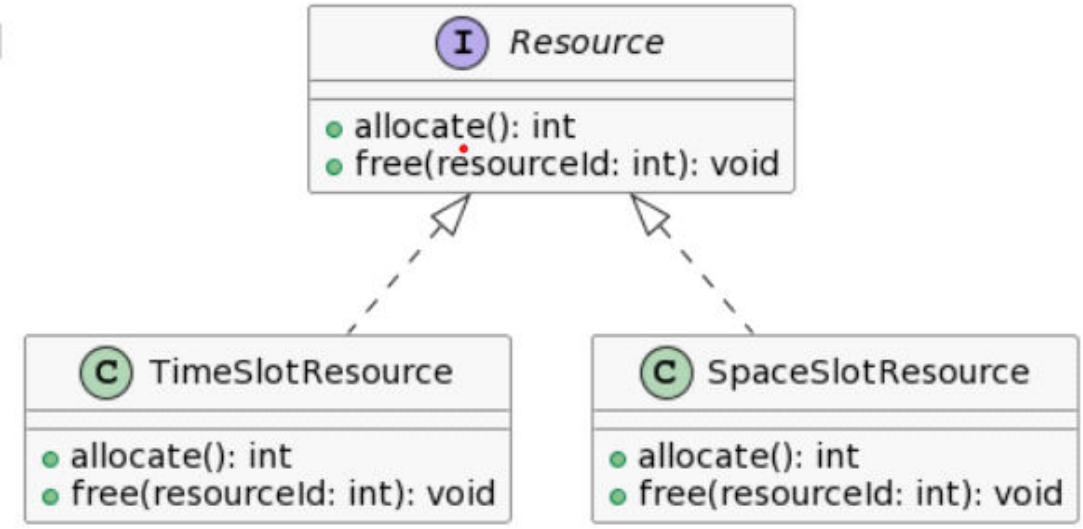
- SpaceSlotResource and TimeSlotResource Classes

Responsibility: Implements the Resource interface to manage space and time slots, respectively.

Extension: Each class encapsulates the logic specific to allocating and freeing its type of resource.

By encapsulating resource-specific logic in separate classes that implement a common interface, the system can easily be extended with new types of resources without modifying the existing resource allocator logic or the Resource interface.

UML Diagram :



Exercice 3 :

The LSP states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program. In simpler terms, subclasses should extend the behavior of their superclasses without changing their original functionality.

Original design issues :

- In the original ElectronicDuck class, overriding the quack and swim methods to throw exceptions when the electronic duck is turned off violates the LSP.
  The reason being, clients of the Duck class expect any duck, including an ElectronicDuck, to quack and swim without having to know about additional conditions like being turned on or off.

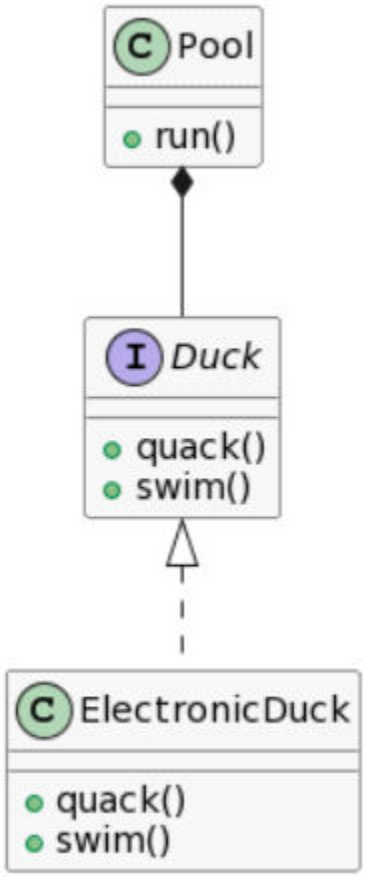
- This design introduces a tight coupling between the client code (e.g., Pool class) and the specific implementation details of ElectronicDuck, thereby breaking the LSP.
  ElectronicDuck is not a proper substitute for Duck because it alters the expected behavior of its methods under certain conditions.

Corrected design explanation :

- The corrected design we suggested is to remove the power state (_on attribute) and its associated behavior (checking if the duck is on before quacking or swimming) from the ElectronicDuck.

- By doing this, the ElectronicDuck class adheres to the LSP because it can now be used interchangeably with the Duck class without introducing unexpected behavior.

UML Diagram :



Exercice 4 :

The ISP suggests that no client should be forced to depend on methods it does not use. This principle promotes the divison of large interfaces into smaller, more specific ones so that clients only need to know about the methods that are of interest to them.

Original design issues :

In the initial design, the Door interface includes methods that are not universally applicable to all types of doors (timeOutCallback and proximityCallback).

This forces implementations of the Door interface (like SensingDoor and TimedDoor) to implement methods that are irrelevant to their specific type, which is a clear violation of the ISP.

As a result, both SensingDoor and TimedDoor include methods throwing NotImplementedException, indicating that these methods are not applicable to all door types.

Original design explanation :

- Segregated Interfaces (Door, ISensing, ITimed)

Door Interface: Now focuses solely on the basic door operations (lock, unlock, open, close), which are applicable to all door types.

ISensing Interface: Specifically for doors that have a proximity sensor feature, providing the proximityCallback method.

ITimed Interface: For doors that need to react to a timer, offering the timeOutCallback method.

- Implementation Classes (SensingDoor, TimedDoor)

SensingDoor Implements Door and ISensing: This class is concerned with doors that operate based on proximity sensing, thus it implements the methods from both Door and ISensing.

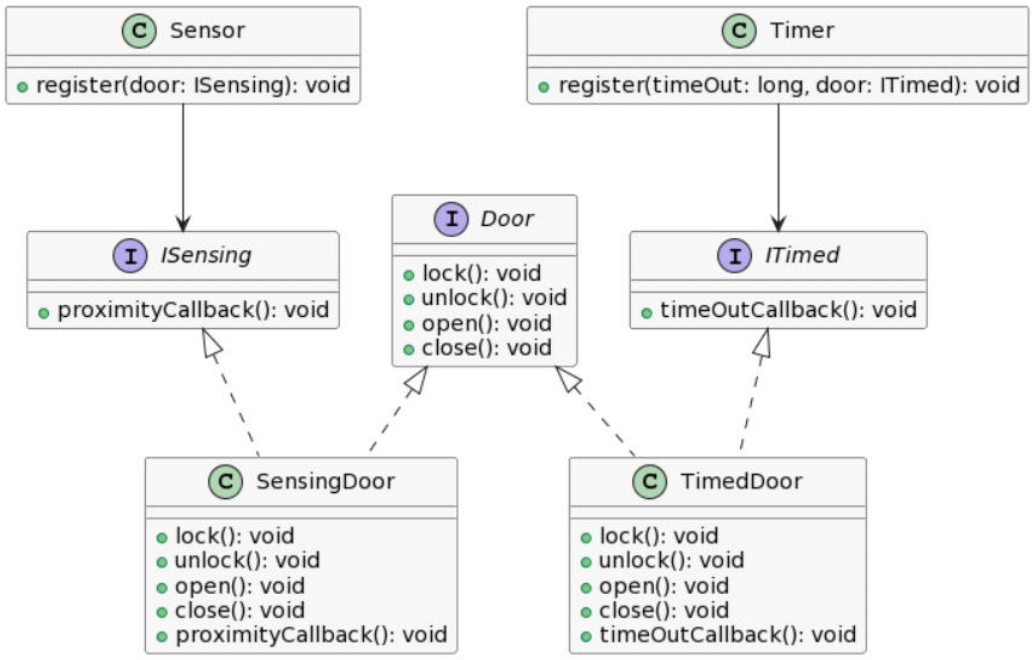
TimedDoor Implements Door and ITimed: Tailored for doors that lock after a certain time, it implements methods from both Door and ITimed.

- Dependency on Specific Interfaces

Sensor Registration: The Sensor class now registers objects that implement ISensing, ensuring that only those with the proximity sensor capability are interacted with.

Timer Registration: Similarly, the Timer class registers objects that implement ITimed, focusing on those that have a timed locking feature.

UML Diagram :



Exercice 5 :

The Dependency Inversion Principle (DIP) mandates that high-level functionality should interact with abstractions rather than directly with low-level implementations, promoting a design where both high-level and low-level modules depend on shared interfaces.

Original design issues :

The original EncodingModule directly interacts with both file and network resources, as well as a specific

database implementation (MyDatabase).

This design tightly couples the high-level logic of encoding data with the low-level details of how that data is stored and retrieved.

Changing the source of data or the method of storage would require modifications to the EncodingModule, violating the DIP.

Corrected design explanation :

We have introduced an abstraction layer for data storage operations through the DataStorage interface. This interface declares a write method that any storage mechanism can implement, allowing the EncodingModule to save encoded data without needing to know the details of how this is achieved.

- DataStorage Interface

Purpose: Defines a common interface for writing data, allowing different storage mechanisms to be used interchangeably.

- DatabaseDataStorage and FileDataStorage Classes

Role: Implement the DataStorage interface to write data to a database and a file, respectively.

Benefit: Allows changing or adding new data storage mechanisms without altering the encoding logic.

- EncodingModule Class

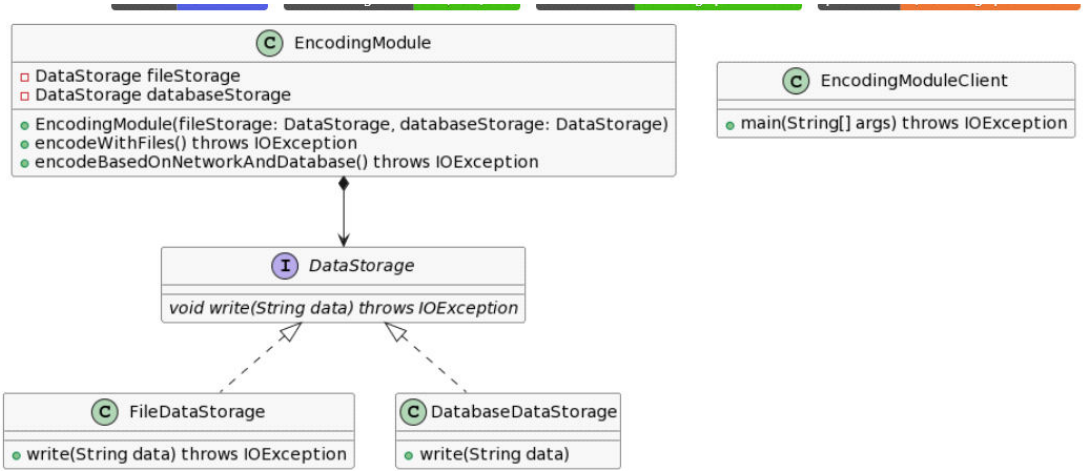
Modification: Depends on the DataStorage abstraction rather than concrete implementations for file and database storage. It means the module is now decoupled from the specifics of data storage and retrieval mechanisms.

Construction: Receives concrete instances of DataStorage for file and database operations through its constructor, allowing for dynamic substitution of storage mechanisms.

- EncodingModuleClient Class

Function: Creates specific instances of DataStorage (for files and databases) and passes them to an EncodingModule instance. This setup exemplifies the inversion of dependencies, where high-level policy (encoding data) is separated from low-level details (how and where data is stored).

UML Diagram :


EncodingModule
DataStorage fileStorage
DataStorage databaseStorage
EncodingModule(fileStorage: DataStorage, databaseStorage: DataStorage)
encodeWithFiles() throws IOException
encodeBasedOnNetworkAndDatabase() throws IOException
EncodingModuleClient
main(String[] args) throws IOException
DataStorage
void write(String data) throws IOException
FileDataStorage
write(String data) throws IOException
DatabaseDataStorage
write(String data)