S.O.L.I.D Principles :

`Sandra Mourali & Salma Ghabri`

GitHub - mouralisandra/S.O.L.I.D

1. SRP :

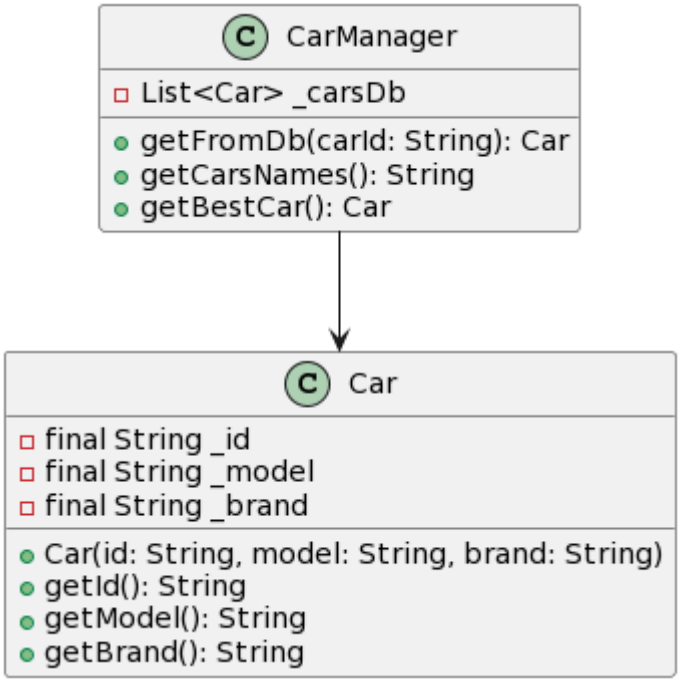
Single Responsibility Principle

Overview :

- A class should have only one reason to change.
- Each class/module should have only one responsibility.

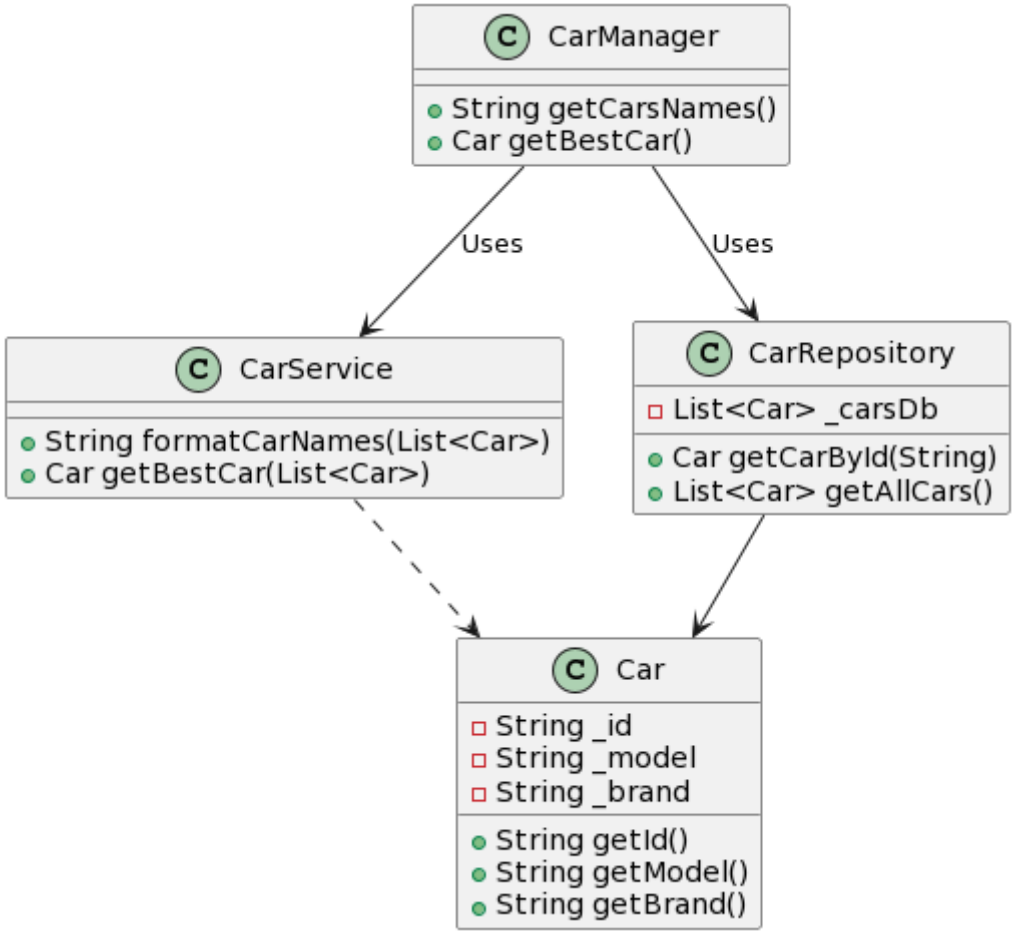
Before Refactoring :

```
In this example we notice a :
-Lack of clear separation of responsibilities within the Car Manager
class.
-Violation of the Single Responsibility Principle:  we can
distinguish 2 distinct treatment types within the carManager class.
-Potential code entanglement and increased complexity in the
carManager class, making it harder to understand, maintain, and
extend the codebase.
```



After Refactoring :

The car manager class now respects the single responsibility prinicple as it now relies on 2 subclasses which are:
-CarRepository class who handles the responsibility of fetching data from the database.
-CarService class handles the **other operations** formatting of car names and getting the best car.
PS : we could have separated the CarService class into two classes, one for formatting the strings and one for getting the best car if the need calls for it.



2. OCP :

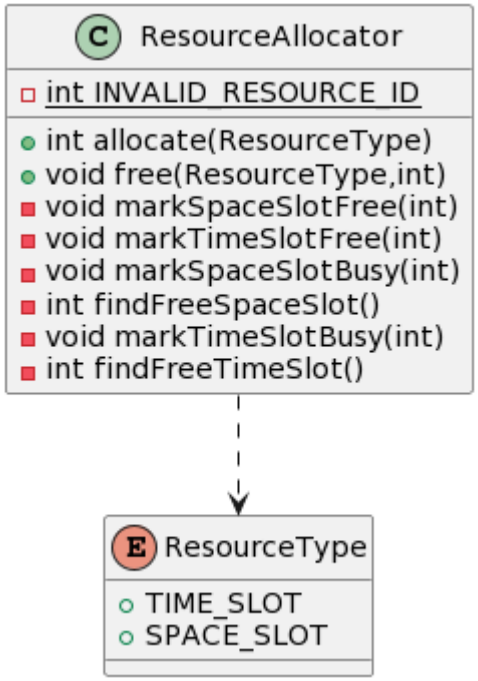
Open/Closed Principle

Overview :

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- Existing code should be open to extension without requiring modification.

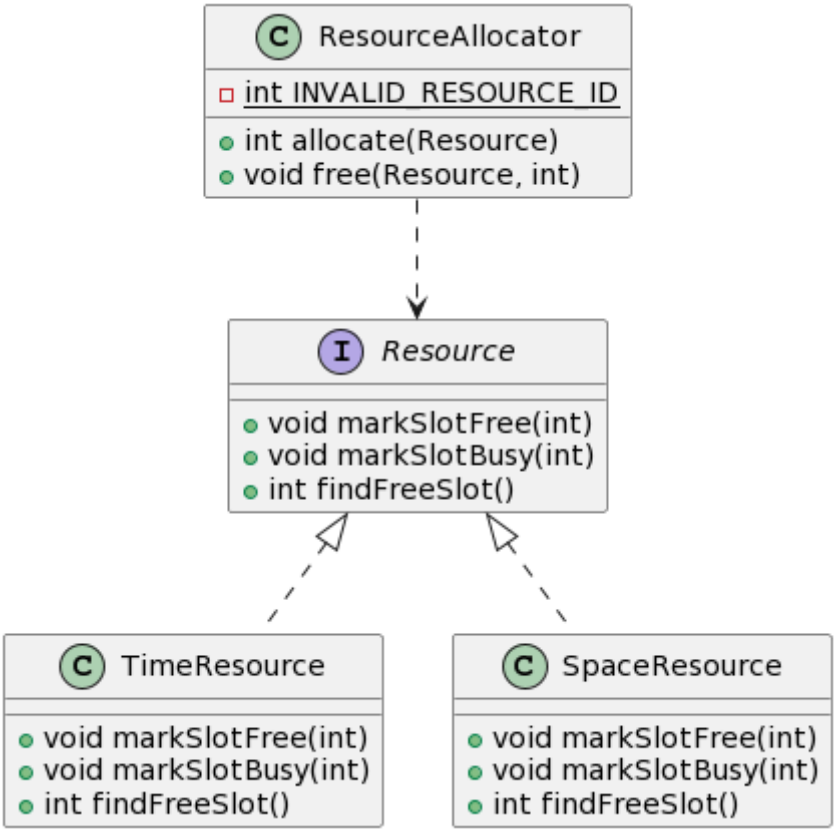
Before Refactoring :

In the Resource allocator exercice we notice that it doesn't follow the Open/Closed Principle and here's why :
-The responsibilities in the resource allocator class aren't clearly separated: we need to separate the functionalities for both Resource types.
-When we need to make changes (for eg. regarding a resource type) , we are obliged to modify the resource allocator class itself instead of extending it which leads to more complicated code.



After Refactoring :

To solve the ResourceAllocator problem, we defined a contract that separated the treatments for the two resource types: Time and space Resource.
- Thanks to the contract (Resource interface), we establish clear boundaries for how each resource type is handled within the system.
- This separation allows for more modular and focused development, adhering to the Single Responsibility Principle by assigning specific tasks to each resource type.



3. LSP :

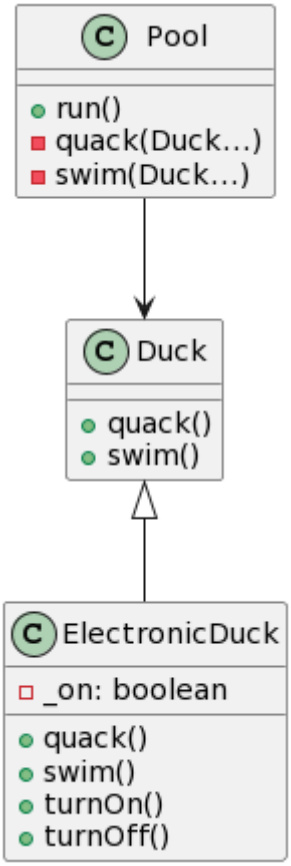
Liskov Substitution Principle

Overview :

- Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.
- Subtypes must be substitutable for their base types.

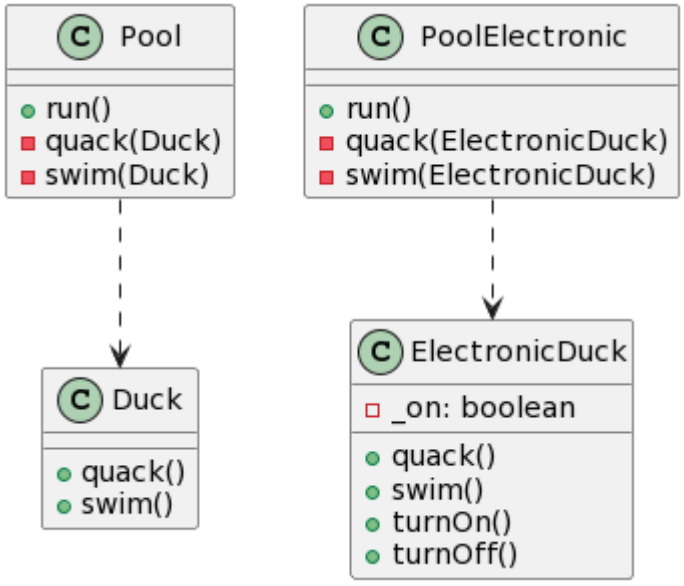
Before Refactoring :

```
In the Pool Duck example we notice that :
-The subclass(electronic duck) doesn't comply with the contract of its
superclass (duck).
-The behavior of an ElectronicDuck object is not stable across
different states (turned on or off). This violates the principle of
having predictable behavior in subclasses, which could lead to
exceptions in client code.
```



After Refactoring :

By refactoring the Duck Pool with the Liskov principle we were able to :
-Remove the inheritance relationship between the electornic duck and duck since the two objects do not behave the same way.
-Get a stable behavior as when **we separate Pool and PoolElectronic** classes it helps to keep the behavior of Duck and DuckElectronic ducks steady, even when they're in different situations. So, whether a DuckElectronic is turned on or off, it acts the way we'd expect based on the Duck rules. This makes the system more reliable.
-Consistent Interactions as objects will always act the same way, no matter which subclass they belong to.
=> This way our design is more solid, and there's less chance of things going wrong or causing errors in the code.



4. ISP :

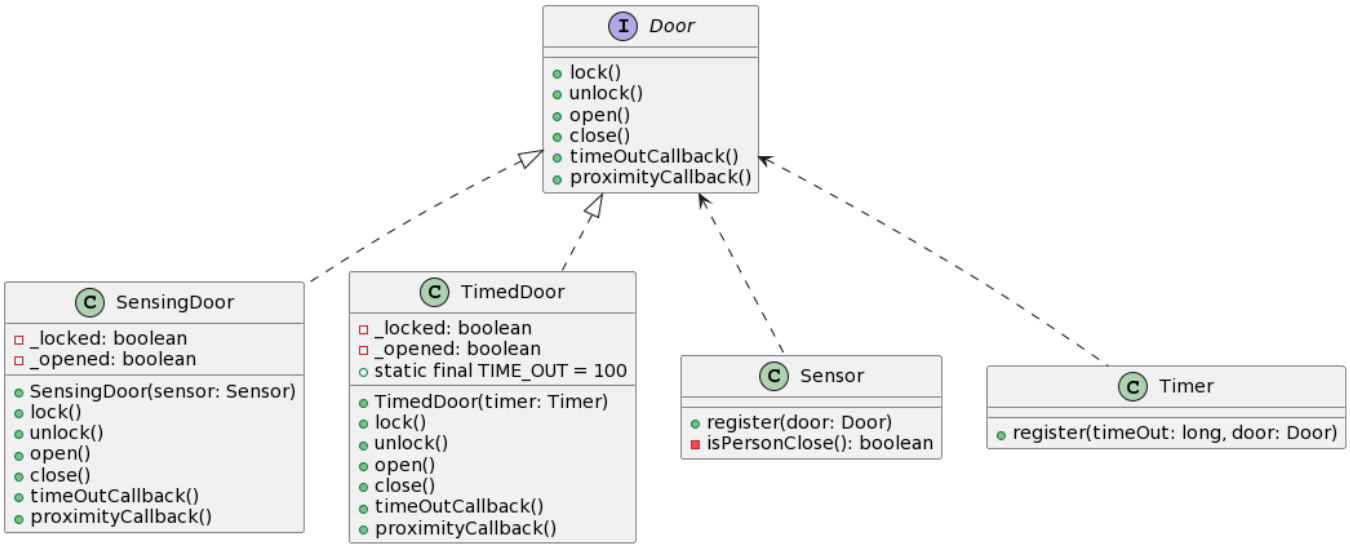
Interface Segregation Principle

Overview :

- Clients should not be forced to depend on interfaces they do not use.
- Make fine-grained interfaces that are client-specific.

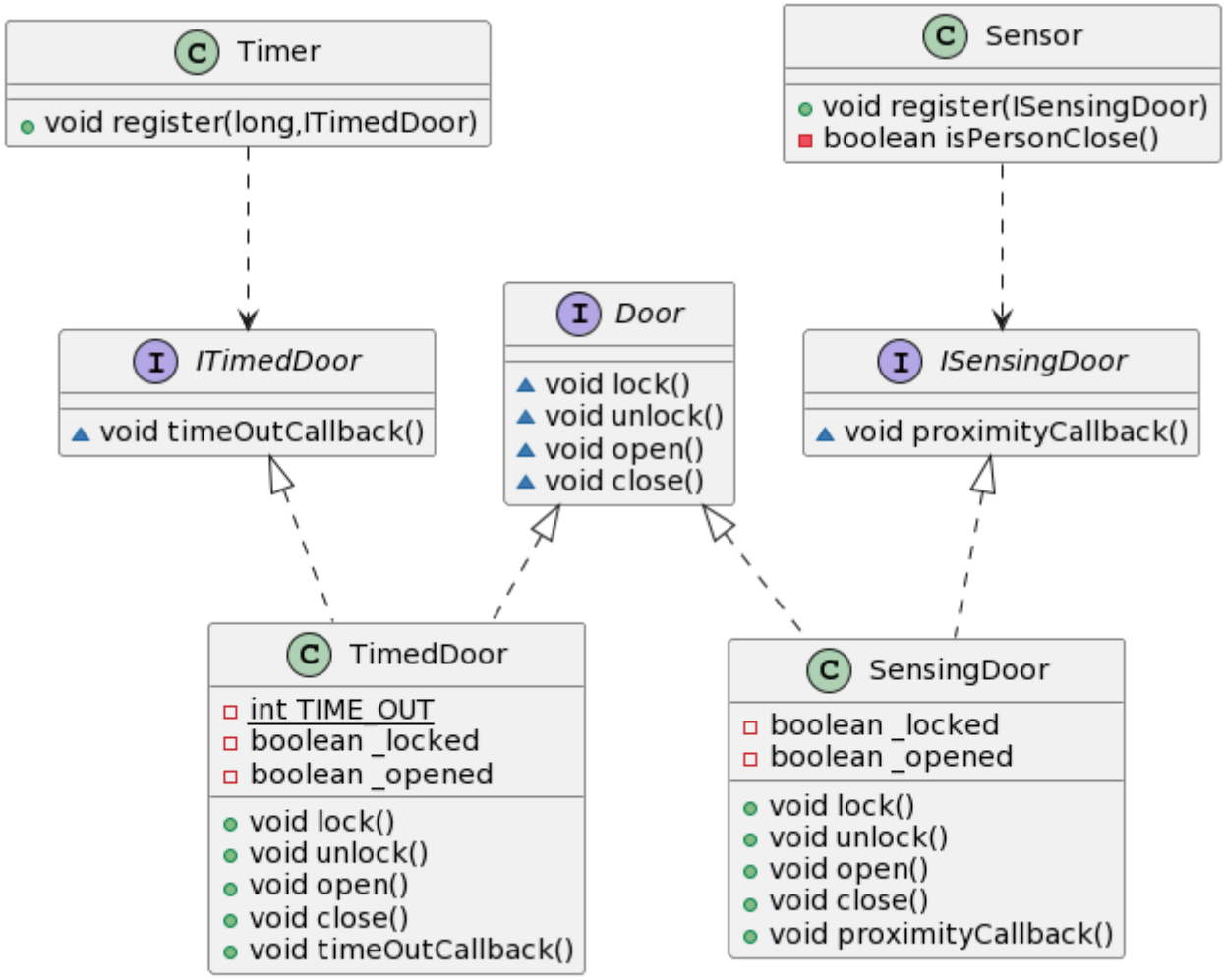
Before Refactoring :

```
- The ISP violation occurs because not all implementations of Door need
to handle timeout callbacks (timeOutCallback) or proximity callbacks
(proximityCallback).
-SensingDoor:
It only needs to handle proximity callbacks, but it's forced to
implement timeOutCallback as well, even though it doesn't use it.
- TimedDoor:
It only needs to handle timeout callbacks, but it's forced to implement
proximityCallback, which it doesn't use.
```



After Refactoring :

-To adhere to ISP, we split the Door interface into smaller, more specific interfaces, each catering to a specific set of functionalities.
-The Door interface handles basic operation for any generic door (locking/unlocking, opening/closing)
-The ISensingDoor interface handles proximityCallbacks for sensing doors.
-The ITimedDoor interface handles timeOutCallbacks for timed doors.



5. DIP :

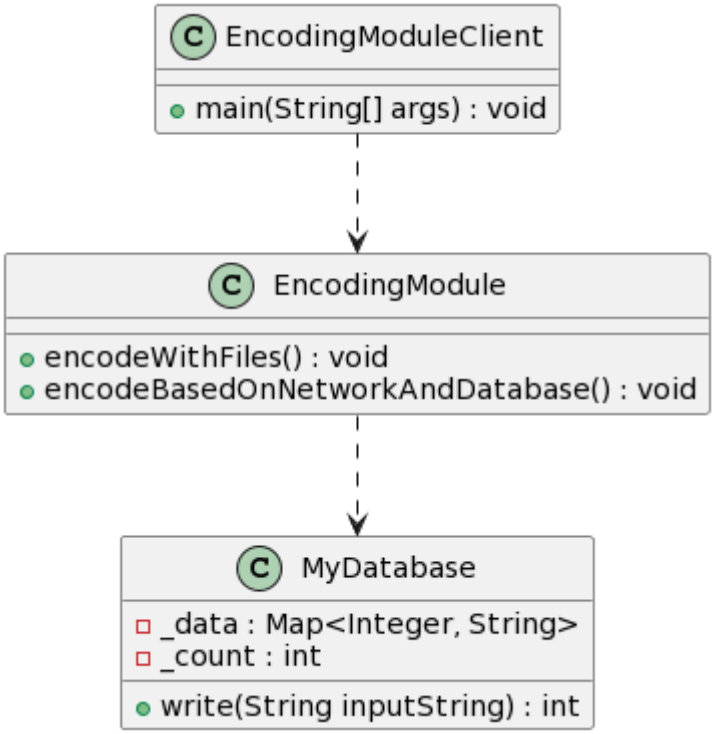
Dependency Inversion Principle

Overview :

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Before Refactoring :

```
the EncodingModule class violates the Dependency Inversion Principle
because it directly depends on concrete implementations such as
BufferedReader, BufferedWriter, FileReader, FileWriter, URL,
InputStream, InputStreamReader
```



After Refactoring:

```
- We noticed that for every encoding type there must be a reader (
either a network or a file) and a writer (database or a file).
-EncodingModule depends now on abstractions ( interfaces) that
represent reading and writing operations, allowing for different
implementations to be provided from files, network and database.
```