SOLID

S - Single responsibility
O - Open - closed
L - Liskov substitution
I - Interface segregation
D - Dependency Inversion

S- Singe responsibility principle
->One class should be responsible for only one job.

In order to have a proper design we have to reduce the responsibilities of one class to a logical minimum. Otherwise it might happen that we ~~encapsulate include~~ in a class some functionalities that block further extensions of the project.

Ex: I saw an example in which the task was to calculate of different geometrical shapes. Two classes, Circle and Square have their own properties, AreaCalculater calculates it. If we include the printing method in AreaCalculater, it blocks the possibility of printing the result in different formats, for example.

~~This way combining the software elements~~

-> One class should only change one part of the software

The more functionalities we include in one class, the more often we have to change it; different functionalities

change at different times

O- Open- closed principle

→ Open Objects or entities should be open for extension but closed for modification

This means we should be able to extend the ~~source code~~ behavior without modifying the source code. By extension we mean adding field or functionalities to a class.

If we want to make sure that we have a design that is easily extensible a good solution would be to include the implementation of interfaces. So if we want to make sure that all the classes of the same type (eg: geometric shapes) have the same new functionality (area calculation), we add the method to the interface and all the classes implementing the interface need to implement the new method as well.

To obtain for an entity ~~that~~ to be closed to modifications we need to have stable, well-defined methods and when we want to modify our software we shouldn't modify the existing code (that might have been already tested and considered well-functioning) or do as less modifications as possible; we should rather ADD new functionalities; we might say that the existing ~~functionalities~~ methods should have only the logically minimal functionalities

L - Liskov substitution principle

-> $q(x)$ - property provable about objects of x of type T

=> $q(y)$ - should be provable for objects y of type S where S is a subtype of T

From this we have to understand that in every case we have to be able to substitute a superclass with any of its subclasses, without risking the correctness of the program -> objects of T may be substituted by objects of S. When an entity extends another it inherits all its attributes and methods and it only adds up to it some others. This way it has all the necessary functionalities that the superclass has => we can substitute the superclass' object with one of the subclass'.

I- Interface segregation principle

-> No client should be forced to depend on methods it does not use or implement an interface that is also not used.

In case of interfaces we can assume that if we want the classes to implement it, we should only apply it in case of classes that implement all of its methods and make use of them. To obtain this we should rather design "smaller" interfaces rather that huge, robust ones with a lot of unimplemented

methods and a lot of classes implementing this interface without even making use of ~~the~~ all of the methods — Clients only have to know about methods that are in their interest.

D- Dependency Inversion principle
→ One should depend on abstractions not concretions

A. High-level modules should not depend on low-level modules. Both should depend on abstractions.

B. Abstractions should not depend on details. Details should depend on abstractions.

The idea behind all this is again to ~~ease the maint~~ increase the maintainability of a program.

If a higher-level module depends on a concrete class (eg: PasswordVerification on a concrete MySql object) if we want to change this we need to rewrite the code (Open-closed principle violation).

We should rather use an interface that the lower level modules implement and making the higher-level module depend on this type (interface) This way we do not ~~more~~ reduce the reusability of any module.