SOLID - Five Design principles for better software designs

Single responsibility principle

- A class / module should have only one reason to change
- E.g consider a module that compiles and prints a report. Imagine such a module can be changed for two reasons. First, the content of the report could change. Second, the format of the report could change.
- These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules.

Open–closed principle

- software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification

- Able to add new functionality without changing the existing code. This prevents situations in which a change to one of the classes also requires adaptation to all depending classes.

Open–closed principle

- It can be achieved through
  - Inheritance
  - Polymorphism

Liskov Substitution Principle

- if S is a subtype of T, then objects of type T may be *replaced* with objects of type S (i.e. an object of type T may be *substituted* with any object of a subtype S) without altering any of the desirable properties of the program.

- An overridden method of a subclass needs to accept the same input parameter values as the method of the superclass.The return value of a method of the subclass needs to comply with the same rules as the return value of the method of the superclass.

Liskov Substitution Principle

- the behavior of your subclass should also be similar to that base class.
- Don’t implement any stricter validation rules on input parameters than implemented by the parent class.
- Behavioral subtyping

Interface segregation principle

- the interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.

- ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.

- Many client-specific interfaces are better than one general-purpose interface

Dependency inversion principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).

- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.