**SOLID** Principles: It helps us to manage most of the software design problems. It also helps us to achieve the followings:

* Achieve reduction in complexity of code.
* Increase reusability, extensibility and maintenance.
* Reduce error and implement reusability.
* Achieve better testability.
* Reduce tight coupling.

**S- Single Responsibility Principle**:

It states that, a class should have only one reason to change or a class should have a Single Responsibility. A Class should have the Data & Behaviour which are related to a Single Responsibility only i.e. everything in the class should be related to that Single Responsibility. By using SRP, classes become smaller, cleaner and the code becomes more fragile.

**O- Open Closed Principle:**

It states that, A Class should be open for Extension (Only when a new requirement or a new Feature needed to be added) but closed for Modifications (Only when we found a Defect or a Bug). The simplest way to apply OCP is to implement the new functionality in the derived classes.

If we do not follow OCP then-

* We will end up testing the entire functionality.
* QA team need to test the entire flow.
* It may break the SRP.
* Maintenance overhead increases on the classes.

**L- Liskov Substitution Principle:**

It states that, we should be able to use any Derived Class instead of using the Base Class & have it behave in the same manner without any modification. It is a collection of guidelines for creating inheritance hierarchies in which a client can reliably use any class or subclass without compromising the expected behaviour. We need to make sure that the Derived Class should not alter the behaviour of the Base Class. A Derived Class should be substitutable for its Base Class.

Implementation guidelines:

* No new exceptions can be thrown by the new derived class.
* Clients should not know which specific subtype they are invoking.
* New derived classes just extend without replacing the functionality of the old classes.

**I- Interface Segregation Principle:**

It states that, we should not force the Client to implement the Interface / Interface Members they don’t use. We should not have a Single Fat Interface in our application which is being used throughout the application. We can create many small interfaces based on groups of methods, each one serving one submodule.

“Many client specific interfaces are better than one general purpose interface”.

**D- Dependency Inversion Principle:**

It states that, the Higher-level Modules/Classes should not depend on the Low-level Modules/Classes and vice versa. Instead they both should depend on the Abstractions. Secondly, Abstractions should not depend upon Details. Details should depend upon Abstractions.