**StarLabs 2022 - Documentation**

**SOLID Principles in C#**

***SOLID design principles in C# are basic design principles. SOLID stands for***

1. Single Responsibility Principle (SRP),
2. Open closed Principle (OSP),
3. Liskov substitution Principle (LSP),
4. Interface Segregation Principle (ISP),
5. Dependency Inversion Principle (DIP).

***S: Single Responsibility Principle (*SRP*)***

**SRP** says "Every software module should have only one reason to change".

This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose. Our class should not be like a Swiss knife wherein if one of them needs to be changed then the entire tool needs to be altered. It does not mean that your classes should only contain one method or property. There may be many members as long as they relate to single responsibility. The Single Responsibility Principle gives us a good way of identifying classes at the design phase of an application and it makes you think of all the ways a class can change. A good separation of responsibilities is done only when we have the full picture of how the application should work.

***O: Open/Closed Principle***

**The Open/closed Principle** says "A software module/class is open for extension and closed for modification".

Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this.

***L: Liskov Substitution Principle***

**The Liskov Substitution Principle (LSP**) states that "you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification". It ensures that a derived class does not affect the behavior of the parent class, in other words,, that a derived class must be substitutable for its base class. This principle is just an extension of the Open Closed Principle and it means that we must ensure that new derived classes extend the base classes without changing their behavior.

***Real world example***

*A father is a doctor whereas his son wants to become a cricketer. So here the son can't replace his father even though they both belong to the same family hierarchy.*

***I: Interface Segregation Principle (ISP)***

**The Interface Segregation Principle** states "that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.".

We can define it in another way. An interface should be more closely related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs rather than which methods the class implements. So clients should not be forced to depend upon interfaces that they don't use. Like classes, each interface should have a specific purpose/responsibility (refer to 1). You shouldn't be forced to implement an interface when your object doesn't share that purpose. The larger the interface, the more likely it includes methods that not all implementers can do. That's the essence of the Interface Segregation Principle.

***D: Dependency Inversion Principle***

The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.

High-level modules/classes implement business rules or logic in a system (application). Low-level modules/classes deal with more detailed operations; in other words they may deal with writing information to databases or passing messages to the operating system or services. A high-level module/class that has a dependency on low-level modules/classes or some other class and knows a lot about the other classes it interacts with is said to be tightly coupled. When a class knows explicitly about the design and implementation of another class, it raises the risk that changes to one class will break the other class. So we must keep these high-level and low-level modules/classes loosely coupled as much as we can. To do that, we need to make both of them dependent on abstractions instead of knowing each other.