SOLID Design Principles are the design principles that help us solve most software design problems. These design principles provide multiple ways to remove the tightly coupled code between the software components (between classes), making the software designs more understandable, flexible, and maintainable.

SOLID Principles SOLID stands for Single Responsibility Principle (SRP), Open closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP).

Singleton is a creational design pattern, which ensures that only one object of its kind exists and provides a single point of access to it for any other code.

Single Responsibility Principle (SRP)- 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. Everything in that class should be related to a single purpose.

Open/closed Principle (OCP)- A software module/class is open for extension and closed for modification. Open for extension" means we must design our module/class so 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.

Liskov Substitution Principle (LSP)- 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, a derived class must be substitutable for its base class.
This principle is just an extension of the Open Closed Principle, and we must ensure that newly derived classes extend the base classes without changing their behavior.

Interface Segregation Principle (ISP)- 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 serving one submodule."
An interface should be more closely related to the code that uses it than the 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 they don't use.

Dependency Inversion Principle (DIP)- High-level modules/classes should not depend on low-level modules/classes. First, both should depend upon abstractions. Secondly, abstractions should not rely upon details. Finally, details should depend upon abstractions.