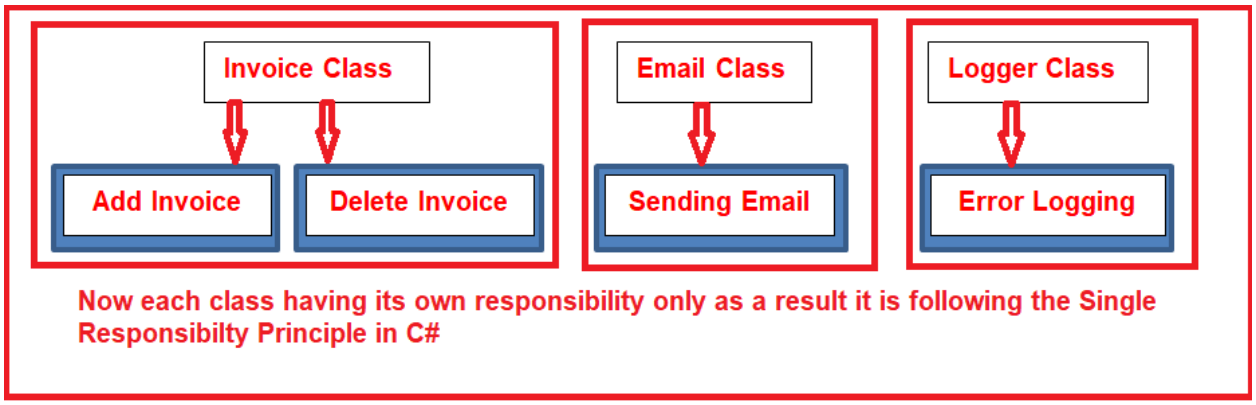
S:SRP

♦SOLID principles are the design principles that enable us to manage most of the software design problems. These principles provide us with ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly.

♦**In simple terms, a module or class should have a very small piece of responsibility in the entire application. Or as it states, a class/module should have not more than one reason to change.**
Example:SRP

Please have a look at the following diagram.



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. We can apply OCP by using interface, abstract class, abstract methods and virtual methods when you want to extend functionality.

Example:OCP

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. I will explain this with a real-world example that violates LSP.

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.

Example:LSP

I : Interface Segregation Principle

Let us break down the above definition into two parts.

1. First, no class should be forced to implement any method(s) of an interface they don't use.
2. Secondly, instead of creating a large or you can say fat interfaces, create multiple smaller interfaces with the aim that the clients should only think about the methods that are of interest to them.

Example:ISP

Dependency Inversion Principle (DIP)

This principle is about dependencies among components. The definition of DIP is given by Robert C. Martin is as follows:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Understanding

The principle says that high-level modules should depend on abstraction, not on the details, of low-level modules. In simple words, the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

The terms Dependency Injection (DI) and Inversion of Control (IoC) are generally used as interchangeably to express the same design pattern. The pattern was initially called IoC, but Martin Fowler (known for designing the enterprise software) anticipated the name as DI because all frameworks or runtime invert the control in some way and he wanted to know which aspect of control was being inverted.

Inversion of Control (IoC) is a technique to implement the Dependency Inversion Principle in C#. Inversion of control can be implemented using either an abstract class or interface. The rule is that the lower level entities should join the contract to a single interface and the higher-level entities will use only entities that are implementing the interface. This technique removes the dependency between the entities.

Example: **Dependency Injection**