What are SOLID Design Principles?

SOLID design principles represent a set of guidelines that helps us to avoid a bad design when designing and developing software. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are 3 important characteristics of a bad design that should be avoided:

- Rigidity - It is hard to change because every change affects too many other parts of the system.
- Fragility - When you make a change, unexpected parts of the system break.
- Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

S.O.L.I.D - is a acronym based on name of the design principles in the collection:

- **S**ingle Responsibility Principle
- **O**pen Close Principle
- **L**iskov's Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- *A class should have only one reason to change.*

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

Single Responsibility Principle was introduced Tom DeMarco in his book Structured Analysis and Systems Specification, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

Open Close Principle

- *Software entities like classes, modules and functions should be **open for extension** but **closed for modifications.***

OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don't have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there are many reasons for which you'll prefer to extend it without changing the code that was already written (backward compatibility, regression testing, '). This is why we have to make sure our modules follow Open Closed Principle.

When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.

Liskov's Substitution Principle

- *Derived types must be completely substitutable for their base types.*

This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle was introduced by Barbara Liskov in a 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy

Interface Segregation Principle

- *Clients should not be forced to depend upon interfaces that they don't use.*

This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example if

we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.

Dependency Inversion Principle

- *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.*

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Further more it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.

Dependency Inversion or Inversion of Control are better know terms referring to the way in which the dependencies are realized. In the classical way when a software module(class, framework, ') need some other module, it initializes and holds a direct reference to it. This will make the 2 modules tight coupled. In order to decouple them the first module will provide a hook(a property, parameter, ') and an external module controlling the dependencies will inject the reference to the second one.

By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module. Factories and Abstract Factories can be used as dependency frameworks, but there are specialized frameworks for that, known as Inversion of Control Container.