**Solid Principles**

Solid principles are the design principles which helps to manage most of the software design principles. These are intended to make the design more flexible, understandable and maintainable.

Solid is acronym for five 5 key design principles :

S - Single Responsibility Principle.

O – Open Close Principle.

L – Liskov Substitution Principle.

I – Interface Segregation Principle.

D – Dependency Inversion Principle.

These are applied to any object oriented programming languages. There are many advantages of following solid principles in writing code. They are listed below:

1. Achieve reduction in complexity of code.
2. Increases maintainability, readability and extensibility.
3. Reduces errors.
4. Implement Reusability.
5. Achieve better Testability.
6. Reduce tight coupling.
7. **Single Responsibility principle:**

“This principle states that a class or module should have only one responsibility.”

Requirements keep changing day by day. When a class possess many responsibilities, it makes difficult to change them frequently and no longer the modules would be independent of each other.

By using Single Responsibility Principle it makes code to understand much easier to explain, understand and implement. By this we can detect and fix bugs easily.

1. **Open Close Principle :**

“The Open/Closed Principle states that code entities should be open for extension, but closed for modification.”

One should write the class in such a way that it should closed for modification, but it can extended by creating instance, inheriting etc. If a file is not closed for modification and we keep on changing the file, it has an impact on the clients who are using this file.

1. **Liskov Substitution Principle :**

“The LSP quotes that any child type of a parent type should be able to stand in for that parent without things blowing up.”

Liskov Substitution Principle is a particular definition of subtyping relation, called behavioural subtyping and this is an extension of open close principle.

If the subtype is not replaceable for the supertype reference, then in order to support the subtype instances as well we go ahead and make changes to the existing code and add the support. This is a clear violation of OCP.

1. **Interface Segregation Principle :**

The Interface Segregation Principle (ISP) says that one should favor many, smaller, client-specific interfaces over one larger, more monolithic interface.

If this isn’t followed we will be forcing a client to implement all methods present in a big interface which is a burden to client. Instead of it we can have smaller interfaces which has only those methods which a client can relate.

1. **Dependency Inversion Principle :**

“This principle suggests to write our code basing on abstractions rather than on concrete details.”

So we need to define abstract interfaces for low level components and resolve dependencies upon object construction.

**Other Principles :**

1. **Law of Demeter :**

This is a design guideline for developing software, particularly object-oriented programs. Law od Demeter is a specific case of loose coupling.

Law of Demeter says that :

* Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
* Each unit should only talk to its friends; don't talk to strangers.
* Only talk to your immediate friends.

As objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers.

1. **Design by Contract :**

It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

Design by contract does not replace regular testing strategies, such as [unit testing](https://en.wikipedia.org/wiki/Unit_testing), [integration testing](https://en.wikipedia.org/wiki/Integration_testing) and [system testing](https://en.wikipedia.org/wiki/System_testing). Rather, it complements external testing with internal self-tests that can be activated both for isolated tests and in production code during a test-phase. The advantage of internal self-tests is that they can detect errors before they manifest themselves as invalid results observed by the client. This leads to earlier and more specific error detection.

1. **KISS :**

“Keep It Simple.”

This principle says that instead of dealing with bigger codes, deal with smaller ones by breaking them into smaller pieces. This increases the understandability of codes and reduces the frequency of bugs.

**Benefits :**

* Problems can be solved in a small time frame.
* You will be able to produce code to solve complex problems in fewer lines of code.
* Higher quality code can be achieved.
* Easier to maintain.
* Code base will be more flexible, easier to extend, modify or refactor when new requirements arrive.

1. **DRY :**

Don’t repeat yourself.

**Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.**

It explains that if you need to write same piece of code in many places, instead of copy-pasting it, make it as a seperate method and call that method whereever it is required. This is applying DRY at the code level.

1. **Yagni :**

**“**You Aren’t Gonna Need It.**”**

Always implement things when you actually need them, never when you just foresee that you need them.

Two best practices of Yagni :

* Stop building features which may or may not be useful to our application at the present level.
* Save that time and build the present features effectively.