SOLID Principles

SOLID Principles is a coding standard that all developers should have a clear concept for developing software properly to avoid a bad design. It was promoted by Robert C Martin and is used across the object-oriented design spectrum. When applied properly it makes your code more extendable, logical, and easier to read.

When the developer builds software following a bad design, the code can become inflexible and more brittle. Small changes in the software can result in bugs. For these reasons, we should follow SOLID Principles.

Single Responsibility Principle:

A class should have one, and only one, reason to change.

One class should serve only one purpose. This does not imply that each class should have only one method, but they should all relate directly to the responsibility of the class. All the methods and properties should work towards the same goal. When a class serves multiple purposes or responsibilities, it should be made into a new class.

Open-closed Principle:

Entities should be open for extension, but closed for modification.

Software entities (classes, modules, functions, etc.) be extendable without actually changing the contents of the class you're extending. If we could follow this principle strongly enough, it is possible to then modify the behavior of our code without ever touching a piece of original code.

Liskov Substitution Principle:

The Liskov Substitution principle was introduced by Barbara Liskov in her conference keynote "Data abstraction" in 1987. Barbara Liskov and Jeannette Wing formulated the principle succinctly in a 1994 paper as follows:

Let φ(x) be a property provable about objects x of type T. Then φ(y) should be true for objects y of type S where S is a subtype of T.
The human-readable version repeats pretty much everything that Bertrand Meyer already has said, but it relies totally on a type-system:

1. Preconditions cannot be strengthened in a subtype.
2. Postconditions cannot be weakened in a subtype.
3. Invariants of the supertype must be preserved in a subtype.

Robert Martin made the definition sound more smoothly and concisely in 1996:

Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it.
Or simply: Subclass/derived class should be substitutable for their base/parent class.

It states that any implementation of an abstraction (interface) should be substitutable in any place that the abstraction is accepted. Basically, it takes care that while coding using interfaces in our code, we not only have a contract of input that the interface receives but also the output returned by different Classes implementing that interface; they should be of the same type.

Interface Segregation Principle:

A Client should not be forced to implement an interface that it doesn't use.
This rule means that we should break our interfaces in many smaller ones, so they better satisfy the exact needs of our clients.

Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to minimize the side consequences and repetition by dividing the software into multiple, independent parts.

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.

Or simply: Depend on Abstractions not on concretions

By applying the Dependency Inversion, the modules can be easily changed by other modules just changing the dependency module and High-level module will not be affected by any changes to the Low-level module.

Name: Manar Ahmed Abbas

IS / PM