SOLID Principles

By: Khawla ALareeqi

SOLID Definition:

SOLID is a popular set of design principles that are used in object-oriented software development.

SOLID is an acronym that stands for five key design principles:

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle.

All five are commonly used by software engineers and provide some important benefits for developers.

Single responsibility principle:

is a relatively basic principle that most developers are already utilizing to build code. It can be applied to classes, software components, and microservices.

Utilizing this principle makes code easier to test and maintain, it makes software easier to implement, and it helps to avoid unanticipated side-effects of future changes.

To ensure that you're following this principle in development, consider using an automated check on build to limit the scope of classes. This check is not a foolproof way to make sure that you're following single responsibility principle, but it can be a good way to make sure that classes are not violating this principle.

Open-Closed Principle

The idea of open-closed principle is that existing, well-tested classes will need to be modified when something needs to be added. Yet, changing classes can lead to problems or bugs. Instead of changing the class, you simply want to extend it. With that goal in mind, Martin summarizes this principle, "You should be able to extend a class's behavior without modifying it."

Following this principle is essential for writing code that is easy to maintain and revise. Your class complies with this principle if it is:

Open for extension, meaning that the class's behavior can be extended; and Closed for modification, meaning that the source code is set and cannot be changed.

Liskov Substitution Principle:

is perhaps the most difficult one to understand. Broadly, this principle simply requires that every derived class should be substitutable for its parent class. The principle is named for Barbara Liskov, who introduced this concept of behavioral subtyping in 1987. Liskov herself explains the principle by saying:

What is wanted here is something like the following substitution property: if for each object O1 of type S there is an object O2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O1 is substituted for O2 then S is a subtype of T.

While this can be a difficult principle to internalize, in a lot of ways it's simply an extension of open-closed principle, as it's a way of ensuring that derived classes extend the base class without changing behavior.

Following this principle helps to avoid unexpected consequences of changes and avoids having to open a closed class in order to make changes. It leads to easy extensions of software, and, while it might slow down the development process, following this principle during development can avoid lots of issues during updates and extensions.

Interface Segregation Principle:

The general idea of interface segregation principle is that it's better to have a lot of smaller interfaces than a few bigger ones. Martin explains this principle by advising, "Make fine grained interfaces that are client-specific. Clients should not be forced to implement interfaces they do not use."

For software engineers, this means that you don't want to just start with an existing interface and add new methods. Instead, start by building a new interface and then let your class implement multiple interfaces as needed. Smaller interfaces mean that developers should have a preference for composition over inheritance and for decoupling over coupling. According to this principle, engineers should work to have many client-specific interfaces, avoiding the temptation of having one big, general-purpose interface.

Dependency Inversion Principle:

This principle offers a way to decouple software modules. Simply put, dependency inversion principle means that developers should “depend on abstractions, not on concretions.” Martin further explains this principle by asserting that, “high level modules should not depend upon low level modules. Both should depend on abstractions.” Further, “abstractions should not depend on details. Details should depend upon abstractions.”

One popular way to comply with this principle is through the use of a dependency inversion pattern, although this method is not the only way to do so. Whatever method you choose to utilize, finding a way to utilize this principle will make your code more flexible, agile, and reusable.