**Solid Clean Architecture**



Clean code is the foundation of a clean architecture. For components to be clean, classes must be clean. Clean components are required for clean systems. SOLID principles guide clean code.

The SOLID Principles are as follows

* **S – Single Responsibility principle**
* **O – Open-Closed principle**
* **L – Liskov Substitution principle**
* **I – Interface Segregation principle**
* **D – Dependency Inversion principle**

**Single Responsibility principle (SRP)**

**According to Rober C. Martin, the Single Responsibility principle states that "a class or modules should have one, and only one, reason to change," or in other words, "a class or module should have no more than one reason to modify."**

**Example: An EmployeeReviewer class is only responsible for counting an employee's appraisal score based on the company's performance criteria. This class should only be changed if the company's criteria change.**

**Open-Closed principle (OCP)**

"A software artifact should be open for extension but closed for modification," according to the OCP. It suggests that a software structure should be built in such a way that each new release adds new capability rather than changing the prior one. In other words, a software artifact's functionality should be expandable without requiring it to be modified.

Example: Skyscraper a multi-story building is akin to software development. So, instead of altering the previous one, the design should be such that if we want to add a new feature, we can just add some new functions/methods or classes. If we want to build a house that already has a room, then to add a kitchen, we will have to modify the already existing rooms, then the design for building the house is obviously incorrect. The software system should be designed according to the same rules.

**Liskov Substitution principle (LSP)**

**"Derived classes must be substitutable for their base classes," according to LSP. "What is wanted here is something along the lines of 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 remains unchanged when O1 is substituted for O2, then S is a subtype of T," wrote Barbara Liskov. "Objects in a program should be replaceable with instances of their subtypes without affecting the program's correctness," to put it another way.**

**For example, you have a Bike instance that our software utilizes to conduct a drive action. If Honda is a subclass of Bike, this instance might be replaced with an instance of Honda.**

**Interface Segregation principle (ISP)**

"Many client-specific interfaces are preferable than one general-purpose interface," according to ISP. The core of the principle is straightforward. Build distinct interfaces for each client and multiply them into the class if you have a class with several customers. Instead of loading the class with all the methods that the clients require, create specific interfaces for each client and multiply them into the class.

Assume we have a firm management system with many types of jobs, such as admin, HR, and employee. Instead of creating a single class that can do all of these types' actions, we'd construct individual classes for each type that implement their specific actions based on our use case.

**Dependency Inversion principle (DIP)**

According to DIP, "Abstractions are required. Don't put your faith in concretions." The primary mechanism is stated in the DIP. Dependency The strategy of relying on interfaces or abstract functions and classes rather than real functions and classes is known as inversion. This construction begins at the top and descends to the details. High-level modules are reliant on lower-level modules, which in turn are reliant on even lower-level modules.

Example: Assume we have a class distributor that can distribute a blog post across many channels, such as Twitter and Facebook. The distributor employs a composition of various instances, such as a Twitter Share Action and a Facebook Share Action, according to the Interface Segregation Principle. The purpose of the Dependency Inversion Principle now is to avoid relying on actual share action class methods such as share Post on Twitter and share Post On Facebook. Instead, the Distributor class would specify a Sharing interface, which Twitter Share Action and Facebook Share Action would implement. It declares the share Post abstract method. The Distributor class does not need to be aware of the specifics of the share operations. It simply invokes the share Post function.

Code complexity is decreased by being more explicit and straightforward; readability is considerably improved; coupling is generally minimized; and your code has a greater chance of growing cleanly when you use the SOLID Principles.