SOLID principles

The **SOLID** principles are a set of five design principles that were introduced by Robert C. Martin (also known as "Uncle Bob") in the early 2000s. They are a set of guidelines that help developers design software that is easy to maintain, extend, and modify over time.

SOLID is an acronym for the following principles:

- Single Responsibility Principle **(SRP)**
- Open/Closed Principle **(OCP)**
- Liskov Substitution Principle **(LSP)**
- Interface Segregation Principle **(ISP)**
- Dependency Inversion Principle **(DIP)**

Each of these principles has its own purpose and guidelines for implementation, which we will discuss below.

1. **Single Responsibility Principle (SRP):** The **SRP** states that a class should have only one reason to change. This means that a class should have only one responsibility or job to do. If a class has multiple responsibilities, it becomes difficult to maintain and modify over time. This principle encourages developers to break down complex classes into smaller, more focused classes that each have a single responsibility.

   For example, consider a class that is responsible for both handling user authentication and sending emails. This class violates the **SRP** because it has two responsibilities. Instead, we could create two separate classes: one for user authentication and one for sending emails. This would make our code easier to maintain and modify.

2. **Open/Closed Principle (OCP):** The **OCP** states that a class should be open for extension but closed for modification. This means that we should be able to add new functionality to a class without modifying its existing code. This principle encourages developers to design their code in a way that allows for easy extension without breaking existing functionality.

   For example, consider a class that is responsible for calculating shipping costs. If we want to add support for a new shipping method, we should be able to do so without modifying the existing code. We could achieve this by creating a new subclass that extends the existing class and adds support for the new shipping method.

3. **Liskov Substitution Principle (LSP):** The **LSP** states that subclasses should be able to be substituted for their base classes without affecting the correctness of the program. This means that if we have a class hierarchy, we should be able to use any subclass in place of its parent class without causing any unexpected behavior.

   For example, consider a class hierarchy that includes a Square class and a Rectangle class. Since a square is a type of rectangle, we might be tempted to make Square a subclass of Rectangle. However, this would violate the **LSP** because the properties of a square (all sides are equal) are different from the properties of a rectangle (only opposite sides are equal). Instead, we could make both Square and Rectangle subclasses of a common Shape class.

4. **Interface Segregation Principle (ISP):** The **ISP** states that clients should not be forced to depend on interfaces they do not use. This means that we should design our interfaces in a way that allows clients to use only the functionality they need.

   For example, consider a class that implements a large, complex interface that includes many methods. If a client only needs to use a subset of those methods, they will still be forced to depend on the entire interface. This violates the **ISP** because the client is being forced to depend on functionality they do not need. Instead, we could break down the complex interface into smaller, more focused interfaces that clients can use as needed.

5. **Dependency Inversion Principle (DIP):** The **DIP** states that high-level modules should not depend on low-level modules. Instead, both should depend on abstractions. This means that we should design our code in a way that allows for maximum flexibility and modularity.

   For example, consider a class that depends on a low-level database access class. If we want to switch to a different database, we would need to modify the existing class. This violates the DIP because the high-level class is depending on a low-level implementation detail. Instead, we could create an abstraction (such as an interface) that represents the database access functionality. The high-level class would depend on this abstraction, and the low-level implementation would depend on the same abstraction. This would allow us to switch to a different database implementation without modifying the high-level class.

In summary, the **SOLID** principles are a set of guidelines that help developers design software that is easy to maintain, extend, and modify over time. By following these principles, we can create code that is more flexible, modular, and adaptable to changing requirements.