**SOLID Principles in Programming: Understand With Real Life Examples**

Last Updated : 15 Mar, 2024

In software development, [Object-Oriented Design](https://www.geeksforgeeks.org/object-oriented-analysis-and-design/) plays a crucial role when it comes to writing flexible, scalable, maintainable, and reusable code. There are so many benefits of using OOD but every developer should also know the SOLID principle for good object-oriented design in programming. The SOLID principle was introduced by Robert C. Martin, also known as Uncle Bob and it is a coding standard in programming. This principle is an acronym of the five principles which are given below:

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

The SOLID principle helps in reducing tight coupling. Tight coupling means a group of classes are highly dependent on one another which you should avoid in your code.

* Opposite of tight coupling is loose coupling and your code is considered as a good code when it has loosely-coupled classes.
* Loosely coupled classes minimize changes in your code, helps in making code more reusable, maintainable, flexible and stable. Now let’s discuss one by one these principles…

**1. Single Responsibility Principle**

This principle states that “**A class should have only one reason to change**” which means every class should have a single responsibility or single job or single purpose. In other words, a class should have only one job or purpose within the software system.

Let’s understand Single Responsibility Principle using an example:

*Imagine a baker who is responsible for baking bread. The baker’s role is to focus on the task of baking bread, ensuring that the bread is of high quality, properly baked, and meets the bakery’s standards.*

* However, if the baker is also responsible for managing the inventory, ordering supplies, serving customers, and cleaning the bakery, this would violate the SRP.
* Each of these tasks represents a separate responsibility, and by combining them, the baker’s focus and effectiveness in baking bread could be compromised.
* To adhere to the SRP, the bakery could assign different roles to different individuals or teams. For example, there could be a separate person or team responsible for managing the inventory, another for ordering supplies, another for serving customers, and another for cleaning the bakery.

**2. Open/Closed Principle**

This principle states that “**Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification**” which means you should be able to extend a class behavior, without modifying it.

Let’s understand Open/Closed Principle using an example:

*Imagine you have a class called PaymentProcessor that processes payments for an online store. Initially, the PaymentProcessor class only supports processing payments using credit cards. However, you want to extend its functionality to also support processing payments using PayPal.*

Instead of modifying the existing PaymentProcessor class to add PayPal support, you can create a new class called PayPalPaymentProcessor that extends the PaymentProcessor class. This way, the PaymentProcessor class remains closed for modification but open for extension, adhering to the Open-Closed Principle

**3. Liskov’s Substitution Principle**

The principle was introduced by Barbara Liskov in 1987 and according to this principle “**Derived or child classes must be substitutable for their base or parent classes**“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

Let’s understand Liskov’s Substitution Principle using an example:

*One of the classic examples of this principle is a rectangle having four sides. A rectangle’s height can be any value and width can be any value. A square is a rectangle with equal width and height. So we can say that we can extend the properties of the rectangle class into square class.*

In order to do that you need to swap the child (square) class with parent (rectangle) class to fit the definition of a square having four equal sides but a derived class does not affect the behavior of the parent class so if you will do that it will violate the Liskov Substitution Principle.

**4. Interface Segregation Principle**

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “**do not force any client to implement an interface which is irrelevant to them**“. Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces. You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.

Let’s understand Interface Segregation Principle using an example:

*Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets.*

* In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don’t eat in your food. Here the menu should be different for different types of customers.
* The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

**5. Dependency Inversion Principle**

The Dependency Inversion Principle (DIP) is a principle in object-oriented design that states that “**High-level modules should not depend on low-level modules. Both should depend on abstractions**“. Additionally, abstractions should not depend on details. Details should depend on abstractions.

* In simpler terms, the DIP suggests that classes should rely on abstractions (e.g., interfaces or abstract classes) rather than concrete implementations.
* This allows for more flexible and decoupled code, making it easier to change implementations without affecting other parts of the codebase.

Let’s understand Dependency Inversion Principle using an example:

*In a software development team, developers depend on an abstract version control system (e.g., Git) to manage and track changes to the codebase. They don’t depend on specific details of how Git works internally.*

This allows developers to focus on writing code without needing to understand the intricacies of version control implementation.