# Solid Principles

SOLID principles are a set of five design principles that help in designing modular, maintainable, and scalable software systems. These principles were introduced by Robert C. Martin (also known as Uncle Bob) and are widely used in object-oriented programming.

The SOLID acronym stands for:

Single Responsibility Principle (SRP): A class should have only one reason to change. This principle states that a class should have only one responsibility or job, and it should be focused on performing that specific task. By adhering to SRP, classes become more cohesive and easier to understand, test, and maintain.

Open/Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This principle encourages developers to design systems that can be easily extended with new functionality without modifying the existing code. This is achieved through the use of abstractions, interfaces, and inheritance.

Liskov Substitution Principle (LSP): Subtypes must be substitutable for their base types. This principle emphasizes that objects of a superclass should be able to be replaced with objects of its subclasses without affecting the correctness of the program. In other words, subclasses should adhere to the contract and behavior defined by their superclass.

Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use. This principle states that interfaces should be specific and focused on the needs of the clients that use them. It promotes the segregation of interfaces into smaller and more cohesive units, reducing the impact of changes and making the system more maintainable.

**Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions. This principle promotes loose coupling between modules by introducing abstractions and relying on dependency injection. It states that the high-level modules should depend on abstractions (interfaces), rather than concrete implementations, allowing for flexibility, extensibility, and easier testing.

# When to use Solid principles

**SOLID** principles should be applied during the design and development phase of software systems. They provide guidelines and best practices for creating maintainable, modular, and scalable code. Here are some situations where SOLID principles can be beneficial:

**When Designing Classes and Components:** As you design the architecture of your software, you should consider applying SOLID principles to ensure that your classes and components are organized in a way that promotes maintainability and flexibility.

**When Refactoring Existing Code:** If you're working on a legacy codebase or need to make significant changes to an existing system, applying SOLID principles can help you refactor the code to make it more modular and easier to work with.

**When Developing New Features:** When adding new features to your software, consider how they fit within the existing architecture and whether they adhere to SOLID principles. This will help prevent introducing unnecessary complexity or tightly coupled code.

**When Collaborating in a Team:** SOLID principles provide a common set of guidelines that can improve communication and collaboration within a development team. When all team members follow these principles, it becomes easier to understand and maintain each other's code.

**When Working on Object-Oriented Projects:** SOLID principles are particularly applicable in object-oriented programming environments. If your project follows an object-oriented paradigm, these principles can greatly enhance the quality and maintainability of your code.

**When Developing Frameworks or Libraries:** If you're creating software that other developers will use, adhering to SOLID principles can make your codebase more user-friendly and extensible. Well-designed frameworks or libraries based on SOLID principles are easier for other developers to understand and use effectively.

**When Planning for Long-Term Maintenance:** If you anticipate that your software will be maintained and extended over a longer period, applying SOLID principles can save a lot of time and effort in the long run. A well-structured codebase is less likely to become a maintenance nightmare.

**When Focusing on Quality and Maintainability:** If you prioritize code quality, maintainability, and scalability in your project, SOLID principles can guide you in making design decisions that align with these goals.

SOLID principles provide valuable guidance, they are not strict rules that must be followed in every situation. Use your judgment to strike the right balance between adhering to SOLID principles and practicality within your specific project and requirements.