What is SOLID:

- SOLID is a mnemonic (something helps you remember something) for five design principles in object-oriented programming (and software design in general).
- It was popularized by Robert C. Martin (aka “Uncle Bob”), and the acronym was coined by Michael Feathers.
- The goal: make code more maintainable, flexible, modular, and easier to evolve without unintended side-effects.

The Five Principles:

1. Single Responsibility Principle (SRP):

A class (or module) should have exactly one reason to change — i.e. one responsibility or job.

- This keeps classes focused and small. It makes them easier to understand, test, and maintain.
- If a class does too many unrelated tasks (e.g. handles business logic *and* user interface *and* data persistence), changes become risky: a change for one reason might break something else.

Summary:

A class should do only one thing.

Example: Don’t mix saving data with displaying it.

2. Open/Closed Principle (OCP):

“Software entities … should be open for extension, but closed for modification.”

This means once a class/module works, you should be able to extend its behavior (e.g. add features) without modifying its source.

- Achieved via abstractions, inheritance, interfaces or composition. New behavior → new subclass or module rather than editing existing code.
- Benefit: reduces risk of breaking existing functionality when extending the system.

Classes should be open for extension, closed for modification**.**

Example: Add new features via new classes, not by changing old ones.

3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of its subclasses without altering correct behavior of the program.

- In other words: derived classes must behave in ways expected by users of the base class — they shouldn't surprise the caller.
- Violating LSP often leads to subtle bugs, because code that relies on the base class can get unexpected behavior from a subclass.

Subclasses should work anywhere the parent class works**.**

Example: A Bird subclass should still behave like a Bird.

4. Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they don't use. Prefer many small, specific interfaces rather than one large, general-purpose interface.

- This avoids "fat" interfaces where a class must implement methods it doesn't need — leading to unnecessary code and complexity
- It promotes decoupling, and makes code more modular and easier to modify or extend

Don't force classes to implement methods they **don't use**.

Example: Break big interfaces into smaller, specific ones.

5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level concrete modules. Both should depend on abstractions (e.g. interfaces or abstract classes). Also, abstractions should not depend on details — details should depend on abstractions.

- This decouples components. Changing a low-level implementation (e.g. a database, a logger, a network module) shouldn't force changes in high-level business logic.
- Facilitates substitutability (e.g. mocking, swapping modules), reusability, and easier testing.

High-level modules should **depend on abstractions**, not on concrete details.

Example: Use interfaces instead of hard-coding dependencies.

The Benefits:

- Code becomes easier to understand, maintain, and reason about — each part does one thing.
- Easier to extend or adapt: you can add new features without modifying existing, stable code (thanks to OCP, DIP).
- Less coupling, more modular — which means lower risk of bugs when changing parts of the system.
- Better for testing: small, decoupled components are easy to unit-test in isolation, and abstractions help to swap implementations (for mocks, stubs, etc.).

⚠️ Limitations & Criticisms

- Over-abstraction: following SOLID strictly can lead to many small classes/interfaces, which may make the codebase harder to navigate or understand.

- For some domain models (especially business/domain entities), following OCP or DIP can feel overkill — frequent changes might make inheritance hierarchies rigid or cumbersome.
- Using too many interfaces and abstractions can make the design more complex than necessary.
- Principles remain guidelines, not strict rules. They should be applied with judgment. Blindly applying SOLID everywhere may backfire.

🔨 Practical Advice on Using SOLID

- Use SOLID as **guidelines**, not dogma("A rule you must believe, even without questioning it."). Evaluate when a principle brings real benefit vs when it introduces unnecessary complexity.
- Combine SOLID with other design ideas (e.g. composition over inheritance, separation of concerns, modular design) for flexible, maintainable architecture.
- Refactor incrementally: extract responsibilities, define abstractions/interfaces, decouple dependencies. That way you improve design without rewriting everything.
- In small/simple projects — heavy abstraction may be overkill. For large, evolving systems — SOLID tends to pay off.
