SOLID Principles in C#

By
Narasimha Rao T
Corporate Trainer and Mentor
Professional Development Trainer
tnrao.trainer@gmail.com

SOLID Principles

SOLID Principles Overview

Definition:

SOLID is an acronym representing **five key object-oriented design principles** introduced by Robert C. Martin (Uncle Bob) to make software **more maintainable, scalable, and robust**.

Purpose:

- Encourage **clean architecture**.
- Reduce **technical debt**.
- Promote **reusability and flexibility**.

Disadvantages

- **More Classes** – Each strategy is a separate class, which increases code count.
- **Indirection Overhead** – Slight performance cost due to delegation.
- **Client Awareness** – The client must understand the differences between strategies to choose the right one.

Real-World Analogy

Think of a **navigation app**:

- You can choose different route strategies: *Fastest Route*, *Shortest Distance*, *Avoid Tolls*.
- The app (context) doesn't care how the route is calculated — it just uses the selected strategy.

5. SOLID Principles in C#

1. **S – Single Responsibility Principle**
   A class should have only one reason to change.
2. **O – Open/Closed Principle**
   Software entities should be open for extension but closed for modification.
3. **L – Liskov Substitution Principle**
   Subtypes must be substitutable for their base types.
4. **I – Interface Segregation Principle**
   No client should be forced to depend on methods it does not use.
5. **D – Dependency Inversion Principle**
   Depend on abstractions, not concretions.

1. S – Single Responsibility Principle (SRP)

Definition:
A class should have only **one reason to change**, meaning it should have only **one job or responsibility**.

Why We Use It:

- To make classes **focused and understandable**.
- To reduce complexity and avoid tightly coupled responsibilities.

Advantages:

- Easier to maintain and test.
- Clearer separation of concerns.
- Reduces merge conflicts in team environments.

Disadvantages:

- May result in more classes in the codebase.
- Over-separation can make code harder to navigate.

Real-World Analogy:
A chef in a restaurant cooks food but doesn't also take payments or manage deliveries.

Example Use Case in C#:
Separate `InvoiceCalculator` (calculations) and `InvoicePrinter` (printing) instead of mixing them in one class.

2. O – Open/Closed Principle (OCP)

Definition:

Software entities should be **open for extension but closed for modification**.

Why We Use It:

- To add new functionality **without changing existing code**.
- To reduce the risk of breaking existing features.

Advantages:

- Safer code modifications.
- Supports plugin-like architectures.

Disadvantages:

- May require abstraction layers (slightly more complex).
- Initial setup can take more time.

Real-World Analogy:
A power strip allows you to plug in new devices without rewiring your house.

Example Use Case in C#:
Adding new payment methods by implementing a `IPaymentMethod` interface instead of modifying existing payment code.

3. L – Liskov Substitution Principle (LSP)

Definition:

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

Why We Use It:

- To ensure subclass behavior is consistent with the base class.
- To maintain **polymorphic integrity**.

Advantages:

- Fewer unexpected bugs when substituting derived classes.
- More reliable and predictable behavior.

Disadvantages:

- Misuse of inheritance can easily violate this principle.
- Requires careful design of base class contracts.

Real-World Analogy:

If a driver can drive a car, they should also be able to drive a sports car without learning a new skill set.

Example Use Case in C#:

If `Bird` has a method `Fly()`, a subclass `Penguin` shouldn't exist unless you change the abstraction so flight isn't mandatory.

4. I – Interface Segregation Principle (ISP)

Definition:

No client should be forced to depend on methods it **does not use**.

Why We Use It:

- To keep interfaces **small and specific**.
- To avoid bloated "God interfaces."

Advantages:

- Reduces unused code dependencies.
- Improves code readability and testability.

Disadvantages:

- More interfaces to manage.
- Requires careful thought during design.

Real-World Analogy:

A smartphone app giving you only the permissions it needs (camera-only app shouldn't request microphone access).

Example Use Case in C#:

Instead of one `IPrinter` interface with `Print()` and `Scan()`, separate into `IPrinter` and `IScanner`.

5. D – Dependency Inversion Principle (DIP)

Definition:

High-level modules should **not depend** on low-level modules; both should depend on **abstractions**.

Why We Use It:

- To decouple modules.
- To make systems easier to change or replace parts.

Advantages:

- Greater flexibility.
- Easier testing (mocking dependencies).
- Promotes inversion of control (IoC).

Disadvantages:

- More interfaces/abstractions to maintain.
- Can be overkill for very small applications.

Real-World Analogy:
Instead of a lamp depending on a specific power source, it depends on a plug socket interface — allowing any power source to be used.

Example Use Case in C#:
A `ReportService` depends on `IReportRepository` instead of directly on `SqlReportRepository`.

Comparison Table

Principle	Core Idea	Goal	Example
SRP	One reason to change	Focused classes	Separate invoice printing and calculation
OCP	Extend, don't modify	Safe feature addition	Add new payment type without changing core code
LSP	Subclass must be substitutable	Correct polymorphism	`Dog` can replace `Animal` without breaking
ISP	Small, focused interfaces	Reduce unused dependencies	Separate `IPrinter` and `IScanner`
DIP	Depend on abstractions	Decoupling	Service depends on interface, not concrete DB

Quiz Time

1. What are the three main categories of design patterns?
2. How does the Factory pattern differ from the Abstract Factory?
3. Can Singleton be thread-safe? How?
4. Explain a real-world scenario for the Builder pattern.
5. What is the Open/Closed Principle, and how does it apply in C#?
6. How would you implement the Strategy pattern in a shopping cart application?
7. Why might overusing design patterns be harmful?