SOLID Principles
Complete Study Guide

Author: Study Guide for Software Engineering Principles
Date: November 2025
Purpose: Comprehensive guide to SOLID, DRY, KISS, and YAGNI principles

Introduction

SOLID is an acronym for five design principles intended to make software designs more **understandable**, **flexible**, and **maintainable**. These principles were introduced by Robert C. Martin (Uncle Bob) in the early 2000s and have become the foundation of professional object-oriented software development.

The Five SOLID Principles

Principle	Acronym	Key Point
Single Responsibility	SRP	One class, one job
Open/Closed	OCP	Open for extension, closed for modification
Liskov Substitution	LSP	Subtypes must be substitutable
Interface Segregation	ISP	Small, focused interfaces
Dependency Inversion	DIP	Depend on abstractions, not concretions

1. Single Responsibility Principle (SRP)

"A class should have one, and only one, reason to change."

Each class or module should have **only one job** or **responsibility**. This makes the code easier to understand, maintain, and test. When a class has multiple responsibilities, changes to one responsibility can affect the others, making the code fragile and hard to maintain.

Why It Matters:
• Changes to one responsibility don't affect others
• Easier to understand and maintain
• Easier to test in isolation
• Easier to reuse

2. Open/Closed Principle (OCP)

"Software entities should be open for extension, but closed for modification."

You should be able to **add new functionality** without **changing existing code**. This reduces the risk of breaking existing functionality and makes your codebase more maintainable as it grows.

3. Liskov Substitution Principle (LSP)

"Objects of a superclass should be replaceable with objects of a subclass without breaking the application."

If class B is a subtype of class A, you should be able to replace A with B **without the program breaking**. This ensures that inheritance is used correctly and prevents unexpected behavior.

4. Interface Segregation Principle (ISP)

"Clients should not be forced to depend on interfaces they don't use."

Create **small, focused interfaces** instead of large, "fat" interfaces. Classes should only implement the methods they actually need, not be forced to implement methods they don't use.

5. Dependency Inversion Principle (DIP)

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

Depend on **interfaces** or **abstract classes**, not concrete implementations. Inject dependencies instead of creating them. This makes code more flexible and easier to test.

Additional Principles

Principle	Acronym	Key Point
Don't Repeat Yourself	DRY	One source of truth
Keep It Simple, Stupid	KISS	Simple is better than complex
You Aren't Gonna Need It	YAGNI	Build only what's needed

DRY - Don't Repeat Yourself

Every piece of knowledge should have a **single, unambiguous representation** in the system. Don't duplicate code or logic. If you write the same code twice, extract it into a reusable function or class.

KISS - Keep It Simple, Stupid

Most systems work best if they are kept **simple rather than complicated**. Avoid unnecessary complexity. Simple solutions are better than clever ones. If it can be done simply, do it simply.

YAGNI - You Aren't Gonna Need It

Don't implement something **until it is necessary**. Don't add features "just in case". Build for current requirements, not hypothetical future ones. Add complexity only when needed.

Benefits of Following These Principles

✓ **Maintainable** - Easy to modify and extend

✓ **Testable** - Easy to write unit tests

✓ **Flexible** - Easy to adapt to changes

✓ **Scalable** - Can grow without breaking

✓ **Professional** - Industry-standard practices

✓ **Understandable** - Clear responsibilities and abstractions

✓ **Reliable** - Fewer bugs and side effects

Recommended Reading

"Clean Code" by Robert C. Martin
The classic on software craftsmanship

"Clean Architecture" by Robert C. Martin
Deep dive into SOLID principles

"Design Patterns" by Gang of Four
Classic patterns with SOLID principles

"Head First Design Patterns"
More approachable introduction

"Refactoring" by Martin Fowler
How to improve existing code

Conclusion

SOLID principles are the **foundation of professional software development**. They help you write code that is easy to understand, change, test, and extend. Start applying these principles today and you'll see immediate improvements in your code quality!

Next Steps:
1. Review your current project and identify violations
1. Refactor one violation at a time
1. Practice with simple examples
1. Read "Clean Code" by Robert C. Martin
1. Apply these principles in all future projects

Happy coding! ■