SOLID Principles of Software EngineeringPage

Software Engineering

SOLID Principles of Software Engineering

Five enduring design principles that help teams build maintainable, scalable, and testable systems — from traditional enterprise services to modern AI-powered, agentic applications.

Introduction

Enterprise applications accumulate complexity rapidly. Without clear design principles, teams produce tightly coupled, monolithic classes that become expensive to modify and nearly impossible to test in isolation. SOLID provides a shared language and a practical framework that every engineer can apply from day one — regardless of team size or technology stack.

The SOLID Principles

S Single Responsibility Principle (SRP)

A class should have one, and only one, reason to change. Mixing concerns — e.g., business logic with data persistence — makes changes risky and testing difficult.

```
# X  Mixed concerns
class OrderService:
    def process(self, order): ...
    def send_email(self, order): ...   # unrelated concern

# OK  Separated concerns
class OrderService:
    def process(self, order): ...
class NotificationService:
    def send_email(self, order): ...
```

O Open / Closed Principle (OCP)

Software entities should be open for extension but closed for modification. Use abstraction and polymorphism to add new behaviour without touching existing, tested code.

```
from abc import ABC, abstractmethod
class Discount(ABC):
    @abstractmethod
    def apply(self, price: float) -> float: ...

class SeasonalDiscount(Discount):        # extend by adding new class
    def apply(self, price): return price * 0.9
class LoyaltyDiscount(Discount):
    def apply(self, price): return price * 0.85
```

L Liskov Substitution Principle (LSP)

Objects of a subclass must be replaceable for objects of the superclass without breaking correctness. Violating LSP produces unexpected runtime failures in polymorphic code.

```
class Bird:
    def fly(self): return 'flying'

# X  Ostrich cannot fly — LSP violated
class Ostrich(Bird):
    def fly(self): raise NotImplementedError
```

```
# OK  Redesign hierarchy around capability
class FlyingBird(Bird): ...
class Ostrich(Bird): ...   # omits fly()
```

I Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. Break large interfaces into smaller, role-specific ones to reduce coupling.

```
from abc import ABC, abstractmethod
# OK  Segregated interfaces
class Readable(ABC):
    @abstractmethod
    def read(self) -> str: ...
class Writable(ABC):
    @abstractmethod
    def write(self, data: str): ...
class FileManager(Readable, Writable):   # only what's needed
    def read(self): ...
    def write(self, data): ...
```

D Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. This enables swapping implementations (e.g., databases, APIs) without modifying business logic.

```
from abc import ABC, abstractmethod
class MessageBroker(ABC):               # abstraction
    @abstractmethod
    def publish(self, msg: str): ...

class KafkaBroker(MessageBroker):       # low-level
    def publish(self, msg): ...

class OrderPipeline:                    # high-level
    def __init__(self, broker: MessageBroker):
        self.broker = broker             # injected dependency
```

SOLID Principles — Quick Reference

Principle	Purpose	Key Benefit
S - Single Responsibility	Each class has one reason to change	Easier maintenance & debugging
O - Open/Closed	Open for extension, closed for modification	Safe feature addition without breakage
L - Liskov Substitution	Subtypes must be substitutable for base types	Reliable polymorphism & fewer runtime errors
I - Interface Segregation	Clients depend only on what they use	Reduced coupling & leaner interfaces
D - Dependency Inversion	Depend on abstractions, not concretions	Flexible, testable, mockable components

Practical Benefits in Enterprise Applications

- **Maintainability** — focused classes reduce the blast radius of change.
- **Scalability** — loosely coupled modules scale independently.
- **Testability** — abstractions and DI enable fast, isolated unit tests.
- **Extensibility** — OCP and DIP allow new features without regression risk.
- **Team collaboration** — clear boundaries reduce merge conflicts and ownership ambiguity.
- **Reduced technical debt** — coherent designs resist entropy over time.
- **AI & Agentic systems** — composable, interchangeable components simplify agent orchestration, tool integration, and model swapping.
- **Long-term sustainability** — systems remain comprehensible and adaptable as requirements evolve.

Common Anti-Patterns & Mistakes

Anti-Pattern	Problem	SOLID Violation
God Class	One class owns all business logic, making any change risky	SRP
Tight Coupling	Concrete dependencies hard-coded across layers	DIP
Large Interfaces	Clients forced to implement unused methods	ISP
Hard-Coded Deps	Infrastructure choices baked into business logic	DIP, OCP
Deep Inheritance	Fragile base class; subclasses break on parent changes	LSP, OCP

Common Mistakes to Avoid

- Over-engineering simple solutions with unnecessary abstractions.
- Misusing inheritance where composition is more appropriate.
- Ignoring dependency injection — leads to untestable, tightly coupled code.
- Designing large, bloated interfaces that violate ISP.
- Applying SOLID mechanically rather than purposefully.

Best Practices & Recommendations

- **Design for change** — assume requirements will evolve; build in flexibility.
- **Favour composition over inheritance** for flexibility and reduced coupling.
- **Keep classes focused** — if a class is hard to name, it probably does too much.
- **Program to abstractions** — use interfaces and ABCs at architectural boundaries.
- **Apply dependency injection consistently** — inject dependencies through constructors.
- **Review designs during code reviews** — SOLID violations are often visible at PR time.
- **Refactor continuously** — address violations incrementally rather than in big-bang rewrites.

Conclusion

The SOLID principles are among the most enduring and transferable concepts in software engineering. Adopted as living standards rather than one-time guidelines, they enable teams to confidently evolve complex systems from traditional enterprise services to modern AI-powered, agentic applications. Every engineer, regardless of seniority, should internalise these principles and apply them deliberately in day-to-day design decisions, code reviews, and architectural discussions.

SOLID Principles of Software Engineering

Revision #1
Created 2026-06-19 06:22:10 UTC by H Karthik P Nayak
Updated 2026-06-19 06:22:51 UTC by H Karthik P Nayak