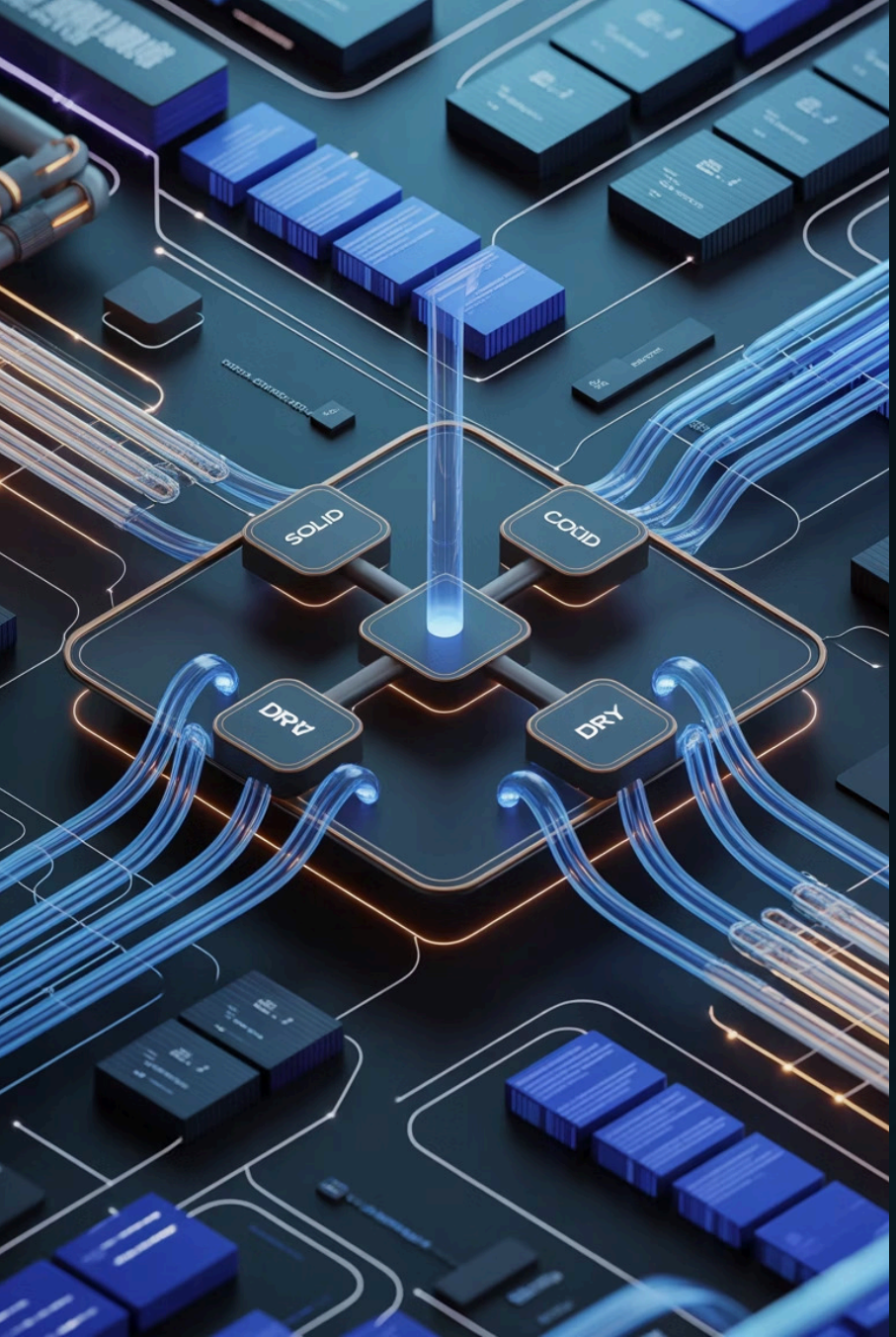



# SOLID Principles in Object-Oriented Design

SOLID is a framework for writing maintainable, flexible software. Created by Robert C. Martin, these principles form the cornerstone of effective object-oriented programming.

N by Naresh Chaurasia


# What is SOLID?

## Acronym

Five design principles that form the foundation of clean code

## Maintainable

Makes code more understandable and easier to change

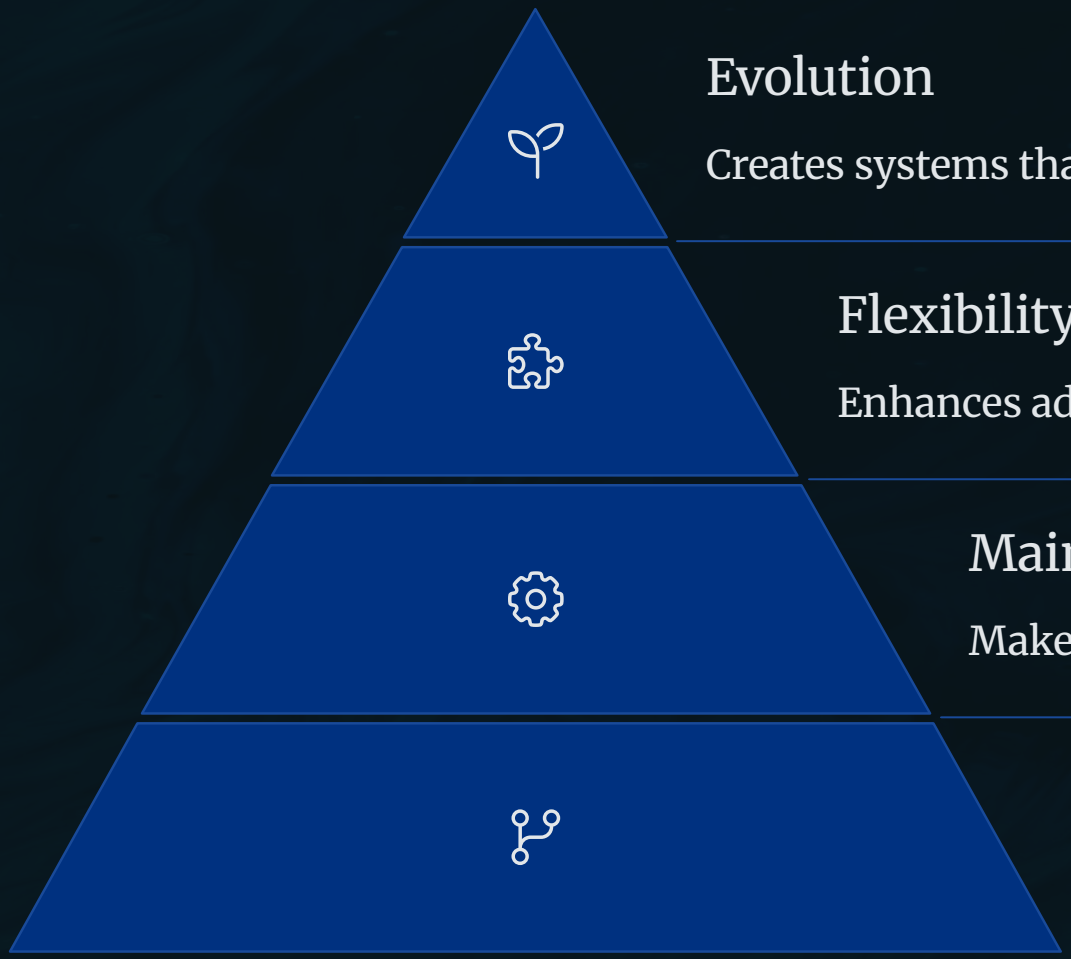
## Protective

Helps avoid "code smells" and accumulating technical debt

## Essential

Required knowledge for senior developers and architects

# Why SOLID Matters



## Evolution

Creates systems that can grow over time

## Flexibility

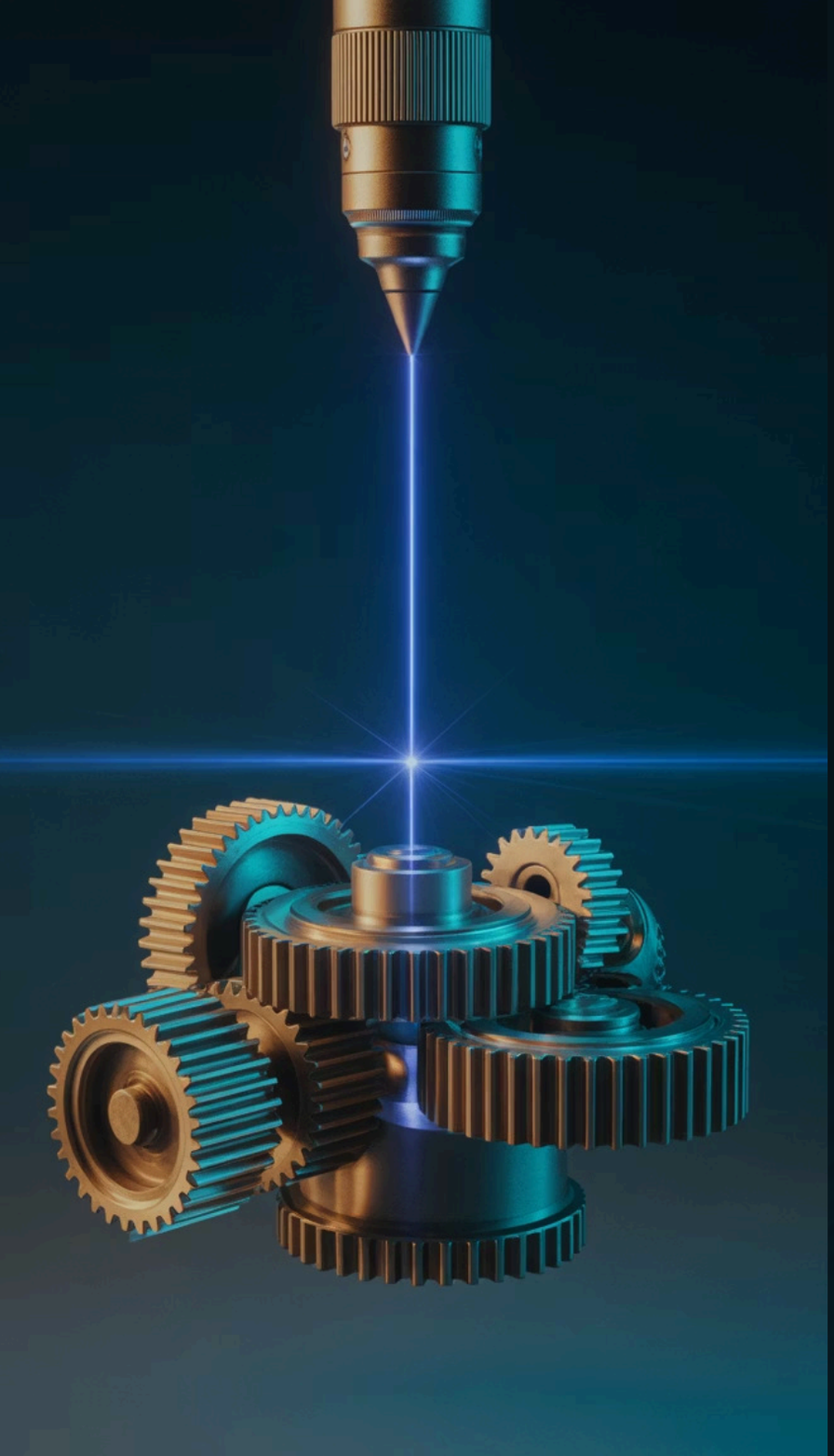
Enhances adaptability and scalability

## Maintainability

Makes code easier to test and maintain

## Simplicity

Reduces complexity as software grows



# S: Single Responsibility Principle

1

## One Reason to Change

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

2

## Focused Problem-Solving

Each class solves exactly one problem

3

## Simplified Testing

Makes unit testing straightforward and comprehensive

4

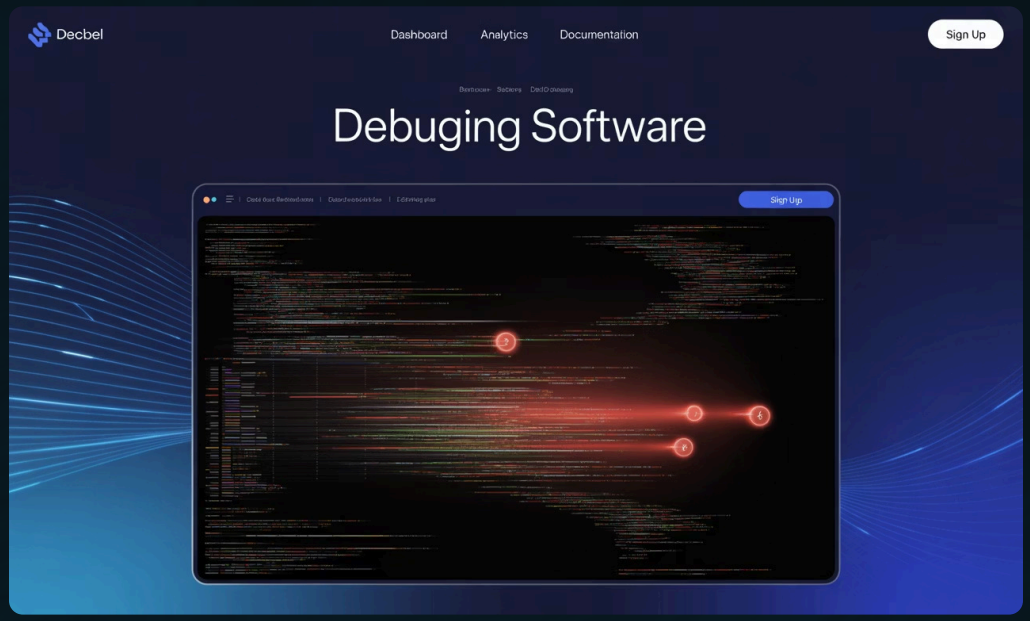
## Broad Application

Works for classes, components, and microservices

# Single Responsibility: Example

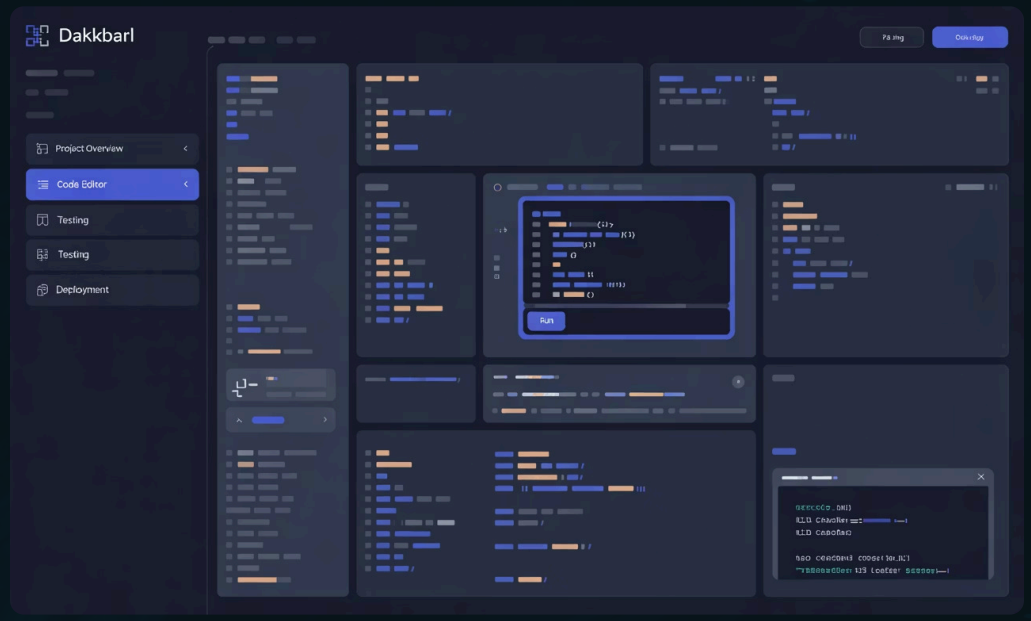
## Bad Practice

UserManager handling authentication, profile updates and notifications




## Good Practice

Separate classes for UserAuthentication, ProfileManager, and NotificationService



# O: Open-Closed Principle

## The Principle

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

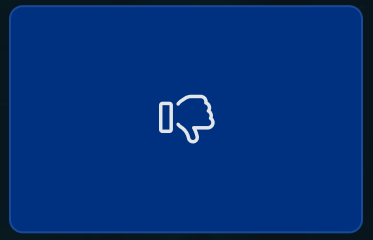
## The Approach

Extend functionality without changing existing code through abstractions and polymorphism

## The Benefits

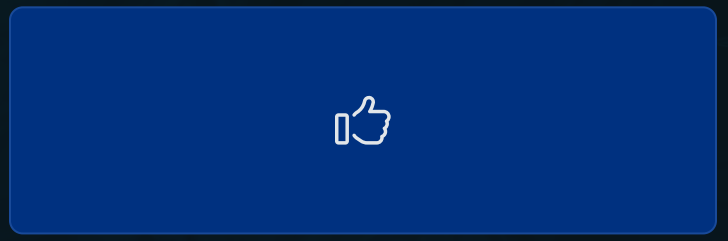
Minimizes risk when adding features and reduces regression bugs

# Open-Closed: Example



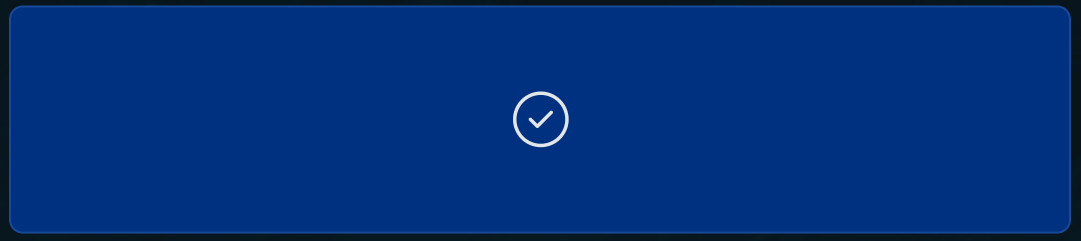
## Bad Practice

Adding if/else statements to Shape.calculateArea() for each new shape



## Good Practice

Abstract Shape class with calculateArea() implemented by each shape



## Benefits

Add new shapes without modifying existing code

# L: Liskov Substitution Principle

## Core Concept

"Subtypes must be substitutable for their base types"

## Contract Adherence

Derived classes must honor base class contracts

## Consistent Behavior

Ensure reliable behavior when using polymorphism

## Clean Code

Avoid type checking and special case handling

# Liskov Substitution: Example

| | |
|---|---|
| Bad Practice | Rectangle and Square inheritance that violates width/height expectations |
| Good Practice | Shape with area() method implemented by both Rectangle and Square |
| Benefits | Code using Shape works correctly with all shape subtypes |

# I: Interface Segregation Principle

### Focused Interfaces

Create specific, targeted interfaces rather than general ones

### Reduce Coupling

Promotes cohesion and minimizes dependencies

### Avoid Bloat

Prevent "fat" interfaces with too many methods

### Implement Only What's Needed

Classes shouldn't implement methods they don't use


# Interface Segregation: Example

## Bad Practice

Worker interface with work(), eat(), and sleep() methods forcing all classes to implement everything

## Good Practice

Separate interfaces for Workable, Eatable, and Sleepable letting classes choose what they need

## Benefits

Classes implement only relevant behaviors, creating cleaner, more focused code


# D: Dependency Inversion Principle

## Abstraction Over Implementation

"High-level modules shouldn't depend on low-level modules"

## Depend on Interfaces

Both high and low-level should depend on abstractions

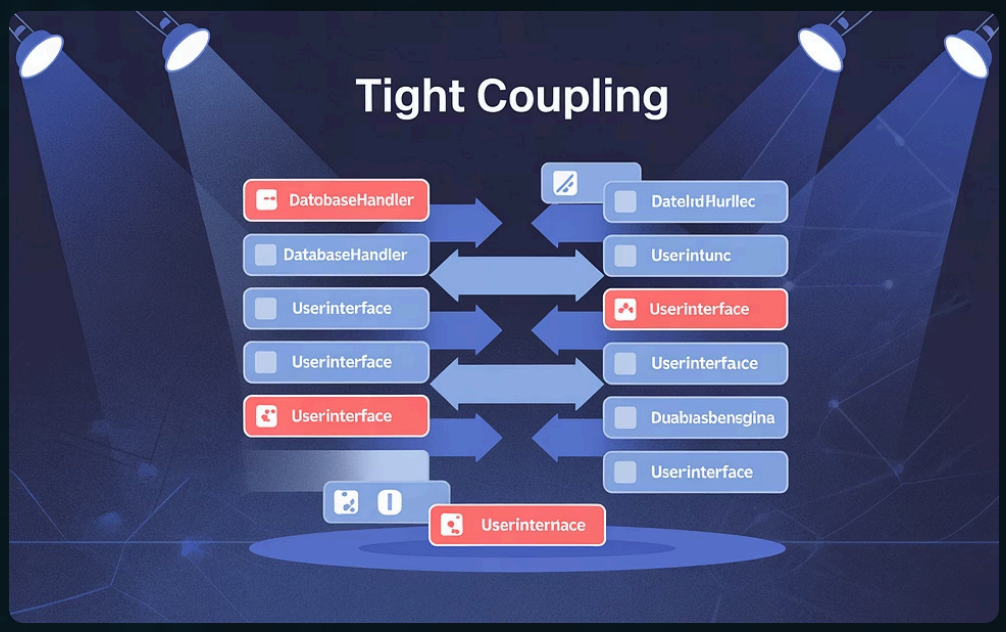
## Decoupling

Separates implementation details from usage

## Flexibility

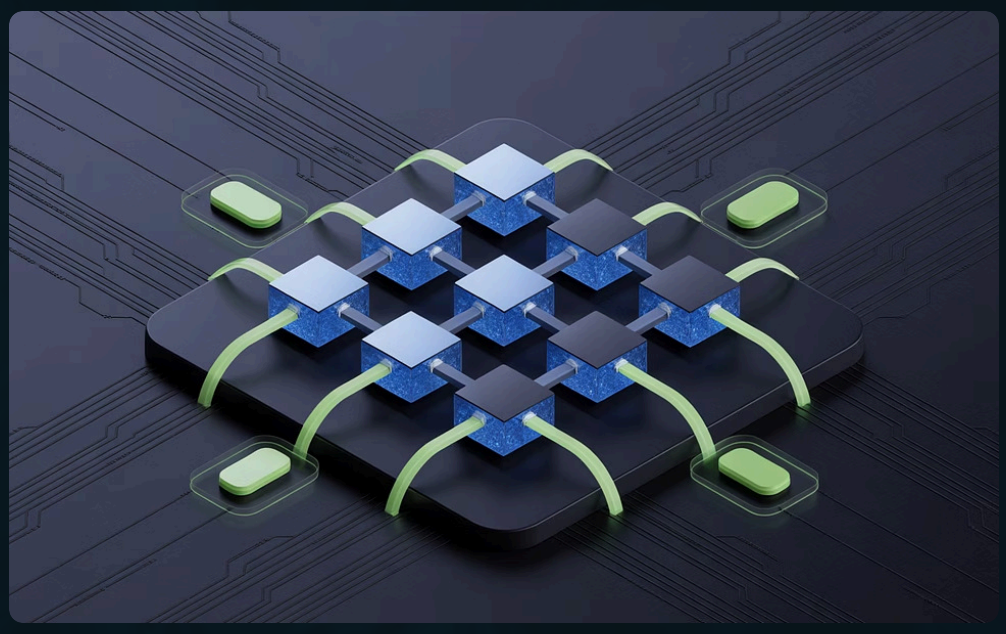
Enables swapping components without changing consumers

# Dependency Inversion: Example



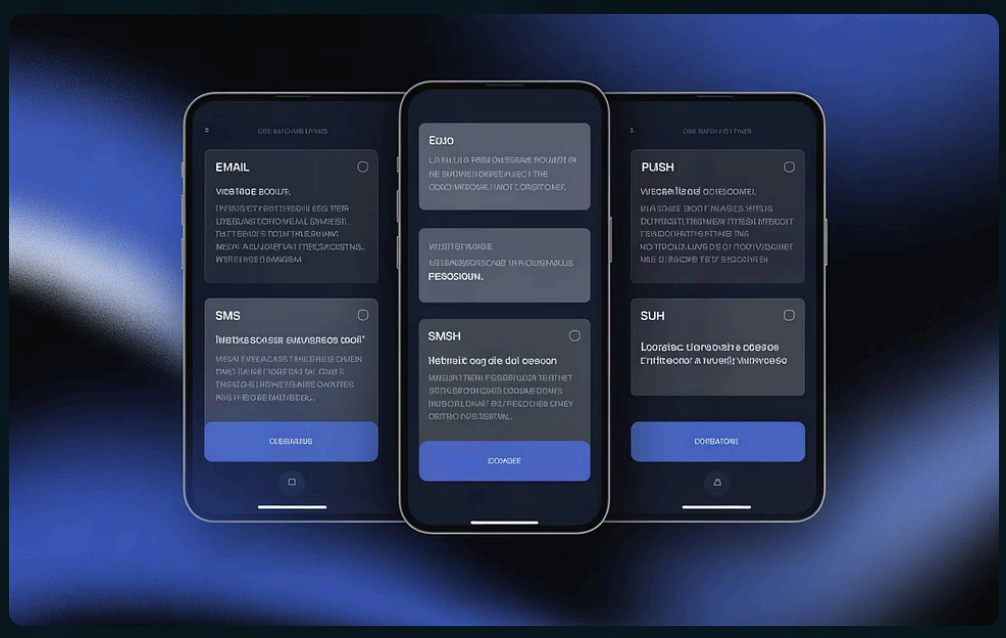

## Bad Practice

NotificationService directly using EmailSender class, creating tight coupling



## Good Practice

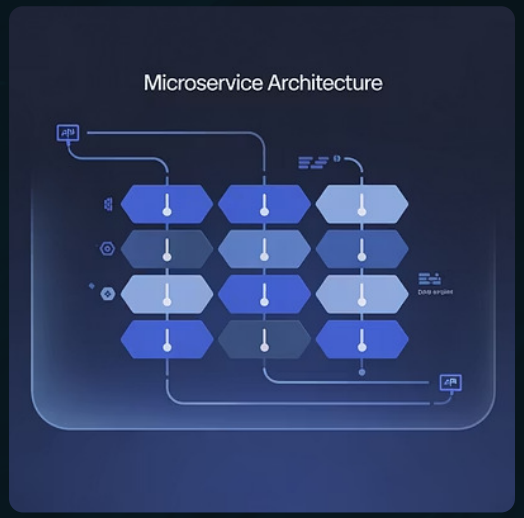
NotificationService depending on MessageSender interface, allowing flexibility



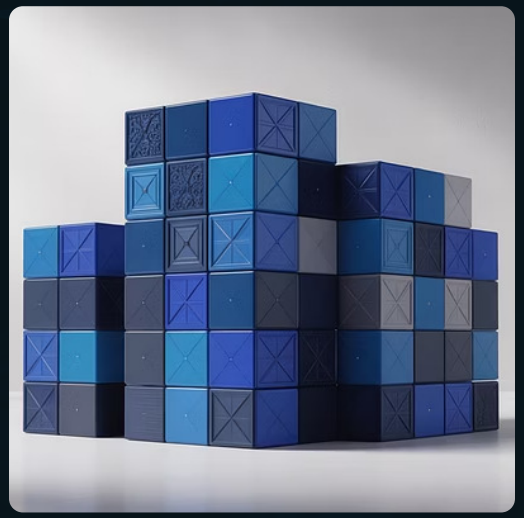

## Benefits

Easily switch between email, SMS, or push notifications

# SOLID in Modern Development








SOLID principles form the foundation for design patterns and microservice architecture. They're essential for maintaining large codebases and are industry standard for quality code.

# Implementing SOLID: Next Steps

1

## Start Small

Begin with Single Responsibility on new code

2

## Refactor

Gradually improve existing code to follow principles

3

## Enforce

Use code reviews to maintain SOLID standards

4

## Balance

Find the middle ground between perfect design and practicality