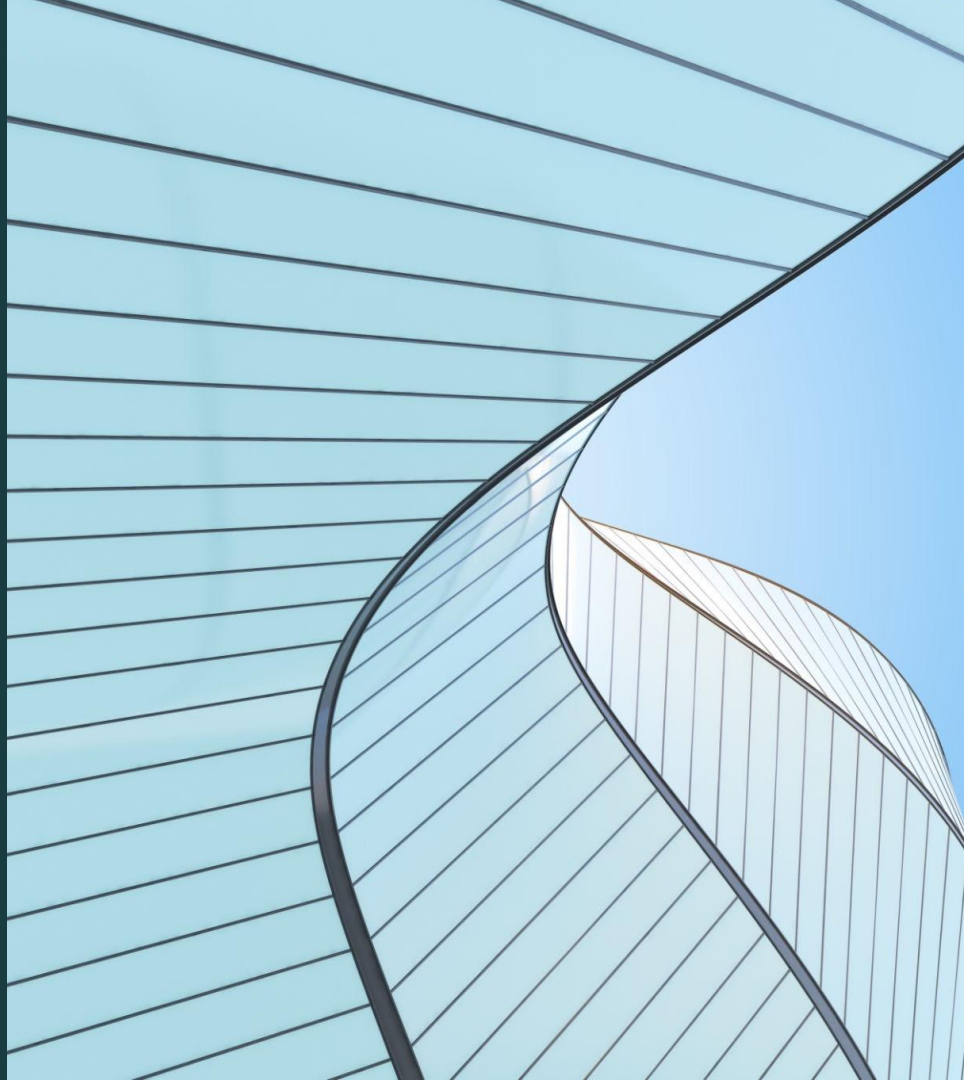
Akash Giri

Stop Writing "Spaghetti" Code. Start Building Systems

SOLID principles



SOLID

1

S-Single Responsibility (SRP)

2

O-Open/Closed (OCP)

3

L-Liskov Substitution (LSP)

4

I – Interface Segregation (ISP)

5

D – Dependency Inversion (DIP)

S – Single Responsibility (SRP)

Layman's Term

"One person, one job."

The Mess

Imagine a single `OrderService` class trying to do everything. It calculates taxes, saves data to the database, and sends out emails. If the email server changes or your tax rules update, you have to modify this giant "God Class," risking a crash for the entire order process

The Fix

Break it down into specialized workers:

- **OrderRepository:** Only talks to the database.
- **TaxCalculator:** Only handles the math.
- **EmailService:** Only handles sending messages

The Simple Retail Example

Think of it like a **Restaurant**. The **Chef** only cooks, the **Waiter** only takes orders, and the **Accountant** only handles the bill.

If the Chef burns a dish, the Waiter can still talk to customers and the Accountant can still process payments. They aren't "stuck" together.

In Code: Changing the "Christmas Theme" on your **Invoice PDF** should never have the power to break your **Credit Card Payment** logic. They should live in different rooms!

O – Open/Closed (OCP)

Layman's Term

"Plug-and-Play."

The Mess

Imagine you have a checkout system that only accepts Credit Cards. When you want to add **Google Pay**, you have to "open up" your existing code and add a messy `if-else` block. Every time you touch that old code to add a new feature, you risk breaking the features that were already working perfectly!

The Fix

Use an **Interface**.

- `interface PaymentGateway { void process(); }`
- Instead of changing the engine, you just "plug in" a new module.

The Simple Banking Example

TThink of it like a **Universal Power Socket**. The socket is **"Closed"**—you don't pull the wires out of the wall every time you buy a new device. It is **"Open"** to any device (Phone, Laptop, Lamp) as long as the device has the right **Plug**.

To add **UPI Payments**, I just created a new "Plug" (the `UPIPayment` class). The main "Socket" (the `PaymentOrchestrator`) didn't change at all.

In Code: You extend the system by adding new classes, not by rewriting your old, tested logic. This keeps your core system safe and stable

L – Liskov Substitution (LSP)

Layman's Term

"A promise is a promise."

The Mess

Your interface promised: *"Every item can be executed."*
You add **E-books** to your system. Since they don't need a warehouse, you make your `DigitalFulfillment` class throw an `UnsupportedOperationException`
Your system processes a "Shirt" and an "E-book" together. When it hits the E-book, the unexpected error **crashes the entire service**.

- **What broke?** You broke the "promise" that every child could stand in for the parent. You forced the system to handle a "surprise" it wasn't built for

The Fix

Subclasses must be true substitutes. Even if "executing" an E-book just means sending an email link, it **must** complete the action without crashing the orchestrator

The Simple Logistics Example

Think of it like a **Battery**. A remote control doesn't care if you use **Duracell** or **Energizer**; both follow the same "contract" (shape/voltage).

In Code: My Orchestrator doesn't ask "Are you a truck or a bike?" It just calls `execute()`. Both handle it differently, but **neither crashes the system**.

I – Interface Segregation (ISP)

Layman's Term

"Don't force me to do things I don't need."."

The Mess

Imagine a giant `OrderOperations` interface that includes `processPayment()`, `shipPhysicalItem()`, and `disburseBankLoan()`.

- **The Problem:** Your **Logistics (Courier) Service** only needs to ship items, but it's forced to implement `disburseBankLoan()`.
- **The Result:** Your courier code is now cluttered with "dummy" banking methods that just throw errors. If you change a Banking rule, you accidentally force the Logistics team to re-deploy their code!

The Fix

Break "Fat" interfaces into smaller, specialized ones. Only give a class the methods it actually needs to perform its job.

The Simple E-Commerce Example

Think of a **Smart TV Remote**. It doesn't have a **Microwave "Popcorn" button** on it. Even though both are "Home Electronics," you don't force the TV remote to carry kitchen features.

In Code: I split my system into `CustomerActions` (Track, Cancel) and `WarehouseActions` (Pack, Ship).

The Benefit: The Warehouse app only "sees" shipping tools. It isn't forced to carry (or accidentally break) the Customer's "Update Profile" logic. It keeps the system secure and modular.

D – Dependency Inversion (DIP)

Layman's Term

"Depend on the blueprint, not the brick."

The Mess

I Your `NotificationService` is hardcoded to talk directly to **WhatsApp API**.

- **The Problem:** One day, the company decides to switch to **Email** or **SMS**. Because your code is "tightly coupled" to WhatsApp, you have to rewrite your entire business logic to swap them out.

The Fix

High-level logic should depend on **Interfaces (Abstractions)**, not concrete tools.

The Simple Cloud Example

Think of **Wall Power Outlets**. The outlet doesn't care if you plug in a Samsung phone or an Apple laptop. It provides a standard "Interface" (the socket). You can swap devices without changing the wiring in your house.

In Code: My engine depends on a `NotificationProvider` interface.

The Benefit: I can swap between **Kafka**, **Gmail**, or **Twilio** by simply changing one line in a configuration file. The core business logic never has to change!