Design Patterns

There are 8 patterns in total

1. Factory pattern :

If you **make a burger yourself**,
you buy ingredients, cook, assemble
you handle all object creation.

If you **order from a burger shop**, you just say
"Give me a cheeseburger"
The shop (Factory) handles the creation logic and you only get the final product.

But what if they add some shit inside the ingredients ?? 😭

That's why we have **Builder Pattern** 🔥

2. Builder Pattern :

We say, i need a burger
We can also be include ingredients that we want

- Factory → *"Give me a burger"* (predefined types, no deep customization).
- Builder → *"Give me a burger, but I'll specify exactly how it's made"* (step-by-step customization).
- You still **don't manually make the burger** — but you **control the build process** through a builder.

3. Singleton Pattern :

Ensure that a **class has only one instance** in the entire application and provide a **global point of access** to it.

Country has 1 prime minister, not multiple
If needed, refer the same one 👍

4. Observer Pattern :

- You have a YouTube channel (**Subject**).
- Viewers (**Observers**) subscribe.
- When a new video is uploaded, all subscribers get a notification.

Decouples publisher from subscribers — channel doesn't care who's watching, just notifies all.

5. Iterator Pattern :

Provide a way to **access elements of a collection sequentially** without exposing its internal structure.Think of a **YouTube playlist** 🎵:

- You don't care how the playlist is stored internally.
- You just **click Next** to move to the next video.
- The playlist gives you the videos one-by-one without revealing how it keeps them.

6. Strategy Pattern :

Strategy Pattern = **“Same goal, different ways to do it, and you can switch the way anytime.”**

You don’t change the main system — you just **swap out the method** used.

You want food 😭
Different **strategies** to get it:

- **Go to restaurant** 🚶
- **Order online** 📱
- **Cook at home** 👨‍🍳

The goal = “Get food” stays the same.
The method (strategy) changes 👍

7. Adapter Pattern :

Adapter Pattern = *“Make two things work together even if they don’t match by using a middleman.”*

It’s like a **translator** — one side speaks English, the other speaks Tamil, the adapter makes them understand each other.

You bought new earphones with a **3.5mm jack** 🎧.

Your new phone **only has a Type-C port** 🔌.

You **don’t** throw away your earphones or buy a new phone.

Instead, you use a **Type-C to 3.5mm convertor**🛠️.

Now your old earphones work perfectly with the new phone.

8. Facade Pattern :

Facade Pattern = “One simple door to access a big, complicated building.”
It gives you **a single, easy-to-use interface** that hides all the messy, complex stuff happening inside.

You want pizza 🍕.

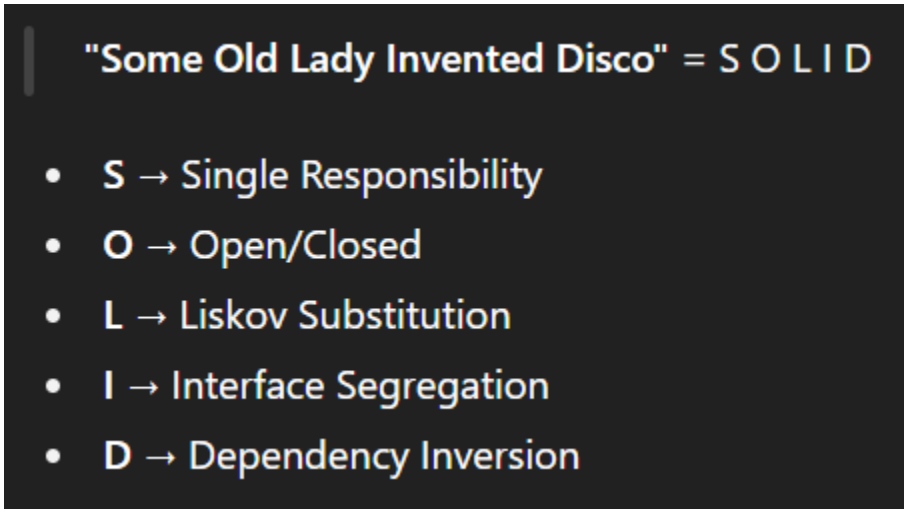
Inside Swiggy’s system there are **tons of steps**:

- Find nearby restaurants
- Check availability
- Process payment
- Assign delivery partner
- Track order

You **don’t** deal with all that.

You just **tap “Order”** and the app (facade) talks to all the complex systems for you.

Solid Principles



???? 😭

S — Single Responsibility Principle (SRP)

One class = One job ✅
Don't make a class handle 20 things — it'll turn into that one friend who *"does everything badly"*.
Example:

- **Good:** `InvoicePrinter` prints invoices, `InvoiceCalculator` calculates totals.
- **Bad:** One `InvoiceManager` that does *printing* + *calculating* + *emailing*.

O — Open/Closed Principle (OCP)

Open for extension, closed for modification 🚪
You should be able to add new features **without touching** existing working code.
Example:

- Add a **new payment method** without rewriting your existing payment logic.

- Use interfaces/abstract classes so new stuff plugs in easily.

📌 In code: Extend with new classes, don't rewrite working ones.

L — Liskov Substitution Principle

If it looks like a duck, it should quack like a duck 🦆
If your app expects a "Bike",

it should work whether it's a "Mountain Bike" or "Road Bike".

📌 In code: Subclasses must work in place of their parent class without surprises.

I — Interface Segregation Principle

Don’t force people to do things they don’t need
Like making a **delivery guy** also learn how to code — unnecessary.

📌 In code: Keep interfaces small, so classes only implement what they need.

For example, JPA repository gives some methods,
If it gives a method to print hello world ??? 😭
Unnecessary right??
So give only whats needed 👍

D — Dependency Inversion Principle

Work with roles, not specific people
If you need “a driver”, don’t say “I need John to drive” — what if John is sick? 🚗

📌 In code: Depend on abstractions (interfaces), not concrete classes.

- You’re building a **payment system** 💳.
- Instead of directly calling `new PaytmPayment()` inside your code, you say:

    - “I need something that can process a payment” → `PaymentProcessor` interface.

- Now, Paytm, GPay, Stripe… all just **plug in** without breaking your code.

- If tomorrow Paytm shuts down 😭, you swap in another payment class — no code chaos.

📌 Why?
Because high-level code depends on an **abstraction** (`PaymentProcessor`), not on a **specific class**.
It’s like saying *“I need a driver”*, not *“I need John”* — if John’s busy, someone else can drive.

Tightly coupled = Your code is glued to one specific class. If that class changes or disappears, everything breaks. 😭

Loosely coupled = Your code only knows about an **interface/abstract contract**, so you can swap implementations like changing SIM cards in a phone. 📱