**WEEK-1**

**Module-1**

**1.SOLID Principles:**

SOLID's aim is to create **maintainable, flexible, and easy-to-understand code.**

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

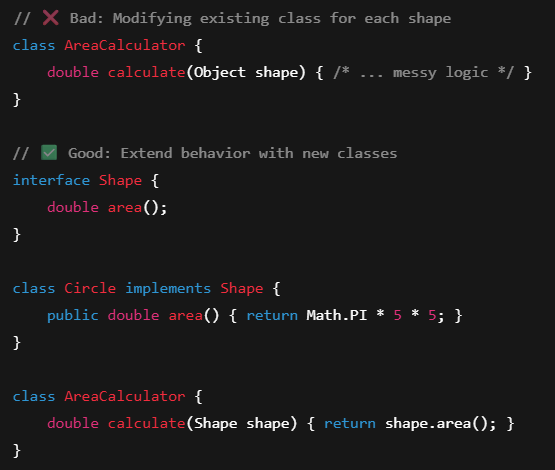
**S – Single Responsibility Principle**

One class should do one job only.  
If it has more than one job, changes to one job might break the other.



**O – Open/Closed Principle**

You can add new features, but don’t touch the old code.  
This avoids breaking things that already work.

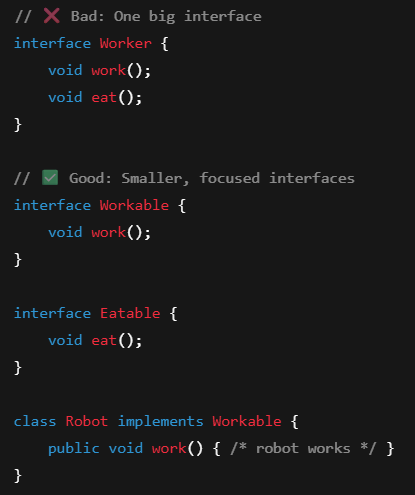


**L – Liskov Substitution Principle**

Child classes should be able to replace parent classes without causing issues.  
If your app breaks after swapping, you’re not following this.

**I – Interface Segregation Principle**

Don’t force people to use stuff they don’t need.  
Keep interfaces short and focused on specific tasks.



**D – Dependency Inversion Principle**

Rely on general rules (interfaces), not specific stuff (classes).  
This makes it easier to swap parts and do testing.

**2.** **Commonly used Design Patterns**

**Design Pattern**

A design pattern in programming is a reusable solution to a common problem that occurs during software design and development. It provides a structured approach to solving specific design or implementation issues, allowing developers to create more maintainable, flexible, and scalable code.

**Why Use Design Patterns?**

* Makes code **readable**, **maintainable**, and **reusable**
* Helps in **communicating ideas** between developers (common language)
* Avoids code smells and overcomplicated structures

Design patterns can be categorized into three main types:

1. **Creational Patterns – *"How objects are created"***

These patterns focus on object creation mechanisms, providing ways to create objects in a manner that is flexible, decoupled from their concrete implementations, and suited to the specific requirements of the application.

Creational patterns handle object creation in a **flexible**, **controlled**, and **decoupled** way. Instead of creating objects directly using new, these patterns provide ways to **abstract and manage the instantiation logic**, especially when objects are complex or dependent on various conditions

1. **Structural Patterns** – *"How things are structured and connected"*

These patterns deal with the composition of classes and objects to form larger structures while keeping them flexible and efficient.

They help ensure that components **fit together**, work efficiently, and can be **easily reused or changed**.

Think of a building. You can use different materials (bricks, steel, glass), but how you arrange and connect them (structure) is what matters for strength and design. Similarly, these patterns help you organize code blocks effectively.

**Common Use Cases:**

* Want to make two incompatible systems work together (Adapter)
* Want to add features dynamically without modifying original class (Decorator)
* Want to control access or behavior of another class (Proxy)

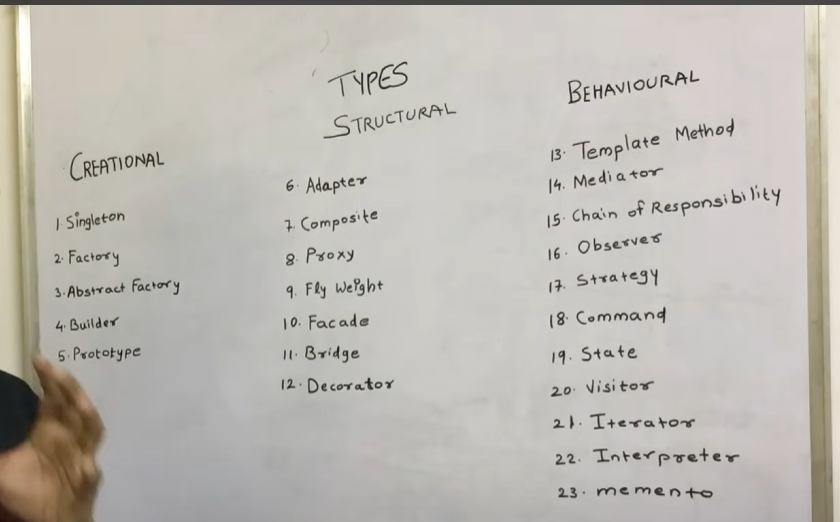
1. **Behavioral Patterns** – *"How objects interact and behave"*

Behavioral patterns define **how objects communicate**, **how responsibilities are assigned**, and **how control flows** between components. These patterns make it easier to manage complex object interactions and encapsulate algorithms or behavior logic.

**Real-world analogy:**  
Think of how a news channel works. The news station (Subject) notifies all its subscribers (Observers) whenever a new story breaks. Or think of a remote control (Command) sending commands to different devices.

**Common Use Cases:**

* Need to notify multiple objects about changes (Observer)
* Want to select algorithms at runtime (Strategy)
* Want to queue or log actions (Command)



**Mandatory Hands-ON:**

Exercise 1: Implementing the Singleton Pattern