**Design patterns & Solid Principles.**

**Solid Principles:-**

**S**ingle Responsibility Principle (SRP)

**O**pen/Closed Principle (OCP)

**L**iskov Substitution Principle (LSP)

**I**nterface Segregation Principle (ISP)

**D**ependency Inversion Principle (DIP)

**SRP-**

states that a class should have only **one reason to change**, meaning it should focus on a single responsibility or functionality. This principle ensures that the class is easy to maintain, test, and understand.

Imagine we are creating a program to manage employees. We have a class that handles employee details, saving employee data to a database, and generating reports.

Issues with This Design:

**Multiple Responsibilities**: The Employee class handles:

* Employee data management.
* Database operations.
* Report generation.

This violates SRP because:

* Changes in database logic will affect the Employee class.
* Changes in reporting logic will also affect the Employee class.

We fix this by refactor the code by separating responsibilities into different classes.

**(OCP)-**

is a design principle in object-oriented programming that states:  
**"Software entities (classes, modules, functions) should be open for extension but closed for modification."**

This means:

* You can add new functionality to an existing class without altering its existing code.
* This minimizes the risk of introducing bugs in the existing functionality and makes the system more maintainable.

we should wright the code in such a way that if we need to add a new functionality we shouldn't modify the code instead we should wright it in a separate class and just extend it.

**LSP-**

**"Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program."**

In simpler terms, any subclass should be able to stand in for its superclass without altering the desired behaviour of the program.

Eg: <https://github.com/MohammadAasimB/dp-sp/blob/main/Liskov%20Substitution%20Principle%20(LSP)>

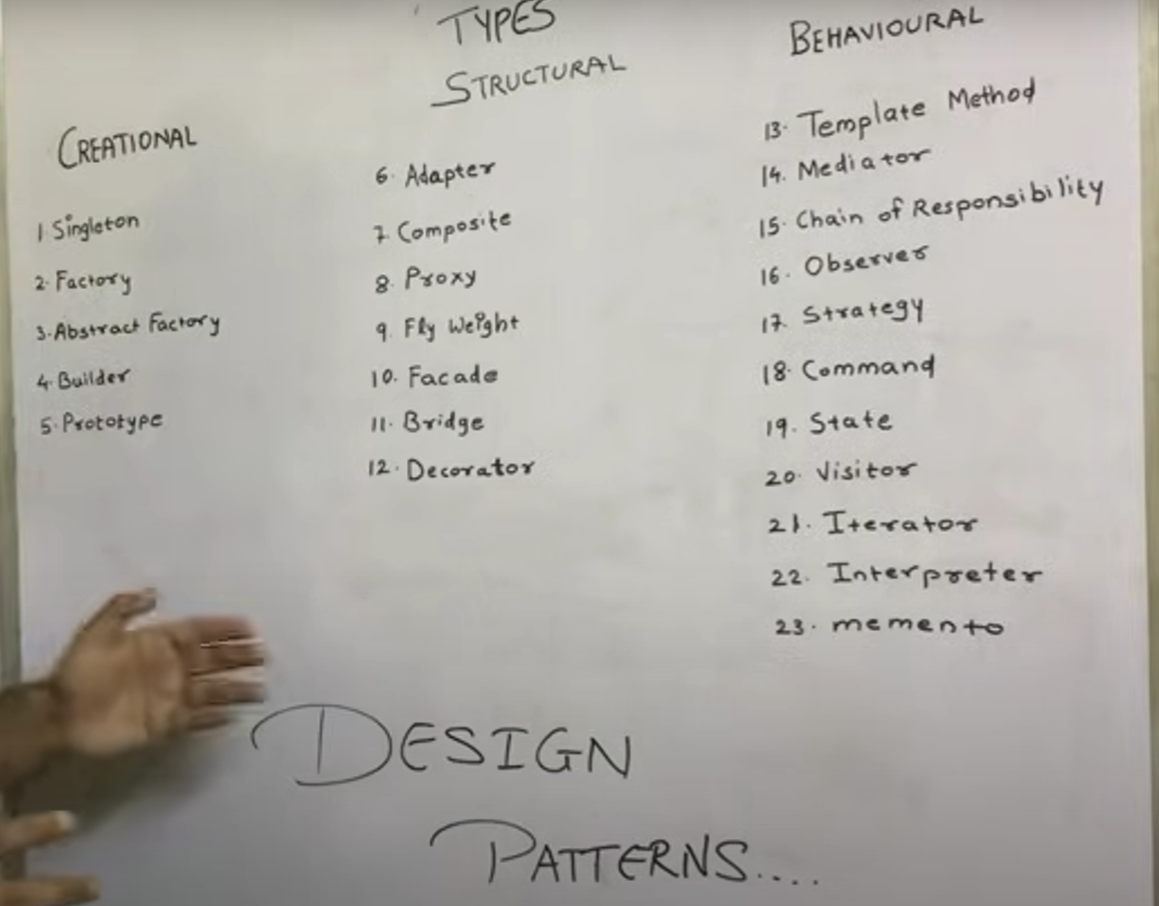
**ISP-**

In Interface Segregation Principle **"A class should not be forced to implement interfaces it does not use."**

In simpler terms, interfaces should be small and focused on specific tasks. If an interface has too many methods, classes that implement it may end up with unnecessary methods they don't need.

Eg: <https://github.com/MohammadAasimB/dp-sp/blob/main/Interface%20Segregation%20Principle>

**Design patterns: -**

****

**Creational-**

Creational patterns deal with the process of creation of the objects of classes.

**Structural-**

Deals with how classes and objects are arranged or composed.

**Behavioural-**

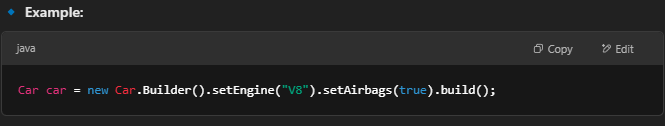
Describes how classes and objects interact and communicate with each other.

**Factory Pattern:-**

🔹 **Definition:** A **creational pattern** that provides a method to create objects without exposing the instantiation logic.  
🔹 **Use Case:** When object creation logic is complex or needs to be centralized.  
🔹 **Example:** CarFactory.getCar("Sedan") returns a Sedan object without using new.

**Builder Pattern:-**

🔹 **Definition:** A **creational pattern** that helps construct complex objects **step by step** while keeping the object **immutable**.  
🔹 **Use Case:** When an object has **many optional parameters** or **different configurations**.



**Singleton Pattern:-**

🔹 **Definition:** A **creational pattern** that ensures **only one instance** of a class exists and provides a **global access point** to it.  
🔹 **Use Case:** When only **one instance** is needed, like **logging, database connections, configuration managers**.

**Prototype Pattern:-**

🔹 **Definition:** A **creational pattern** that allows **cloning objects** instead of creating new ones from scratch.  
🔹 **Use Case:** When object creation is **expensive** or when you need **copies** of existing objects.

If we want to create new instance every time in spring boot:-

